Human Factors in the Design of Parallel Program Performance Tuning Tools

Anna Hondroudakis

Doctor of Philosophy

The University of Edinburgh

1997

Στο Λαμπρο *και στη* Νικη:

αλλη μια φορα τα λογια ειναι φτωχα ...

Declaration

This thesis was composed by myself and the work reported herein is my own except where indicated.

Anna Hondroudakis

Acknowledgements

I would like to thank my supervisor Dr. Rob Procter for his encouragement and the benefit of his scientific experience.

I would also like to thank all the people who contributed to this research by sharing with me information about the ways they develop parallel programs. I am grateful to Edinburgh Parallel Computing Centre (EPCC) for providing me with opportunities to collaborate and speak with "real world" parallel programmers. Thanks to the Engineering and Physical Sciences Research Council for providing financial support during the project.

Thanks to all my friends who have made the bad times bearable and the good times great fun.

Special thanks to my husband Pavlo for his constant encouragement throughout the last years.

This work is dedicated to my parents Lambro and Niki.

Abstract

Parallel program design and development is inherently more difficult than its sequential counterpart. People who have traditionally programmed on sequential machines are now called upon to deal with new and complex issues when trying to control a number of entities executing concurrently. In the quest to produce application programs that take advantage of the power of the underlying hardware, usable and efficient tools are required to assist with the task of performance analysis and tuning. A large number of tools are described in the literature, but the results of tool use are disappointing. The problem can be attributed to the fact that tool design is not informed about users and their tasks. This thesis presents an investigation of tuners' practices and analyses its implications for tool support.

An initial investigation by means of interviews and questionnaires provided the basis of a framework for the analysis of tuning. A number of problems with existing tuning tools were recorded and tuner requirements were captured. Some of these requirements were investigated further with a smaller number of local tuners who participated in the design of VISPAT, a tool for visualising the performance of parallel programs running on clusters of workstations.

Tuning was also examined in the wider context of requirements and work practices within multi-people development projects. An environment is proposed – the Tuner's Workbench – which addresses issues which arise from the repetitive nature of tuning, namely, its management and documentation in the context of real world parallel code development.

Table of Contents

1. Introduction **1**

2. Parallel Program Performance Analysis Tools **5**

2.1 Performance data collection 7

2.1.1 Implementation level 7

2.1.2 Triggering mechanism 8

2.1.3 Instrumentation techniques 9

2.2 Performance data analysis and filtering 12

2.2.1 Data analysis 13

2.2.2 Selection 13

2.2.3 Source code reference 15

2.3 Performance visualisation 17

2.3.1 Animation of communications 19

2.3.2 Program displays 21

2.3.3 Aggregated metrics 23

2.3.4 Animated system views 24

2.3.5 Alternative representations 26

2.4 Automatic performance diagnosis 27

2.5 Tool integration 29

2.6 A Design Agenda 30

3. A framework for studying tuning **34**

3.1 Studies of programming 36

4. Tuning in the small **42**

4.1 Difficulties of tuning 43

4.2 Cause/effect chasm 49

4.3 Faulty assumption/model 52

4.4 Inadequate tools 56

4.4.1 Tool requirements 61

4.4.2 Case study I: tool evaluation 62

4.5 Change 66

4.6 Training in Tuning 68

4.7 Conclusions 69

5. Tuning in the large **77**

5.1 Model of tuning in the large 80

5.2 Tuning problem solving 84

5.2.1 Experimentation techniques 85

5.2.2 Duration of tuning 88

5.2.3 Size of the parameter set 89

5.2.4 Case study I: ad-hoc tuning management 90

5.3 Documentation methods 92

5.3.1 Case study III: tuning notes of informant 17 97

5.3.2 Reasons for documenting tuning 100

5.4 Case study IV: porting a program across many different machines . 101

5.5 Tool requirements 104

5.6 Conclusions 105

6. Tuning across machines, people and organisations. 110

6.1 Different attitudes to tuning 111

6.2 Case study I: the LIBRA project 113

6.3 Case study II: development in X 115

6.4 Tuning within the group 119

6.4.1 Somebody else's code 122

6.5 Case study III: EPCC User Support Service 124

6.6 Conclusions 130

7. VISPAT: a VISualisation and Performance Analysis Tool 134

7.1 Overview of VISPAT 135

7.2 The concept of phases 136

7.3 Trace data format 138

7.4 Data filtering 141

7.5 Visualisation 145

7.5.1 Navigation display 146

7.5.2 Communication display 150

7.5.3 Membership Matrix Display 153

7.5.4 Statistics display 154

7.5.5 Profile display 155

7.6 An example of VISPAT use 156

7.7 Conclusions 160

8. The participative design process of VISPAT **162**

8.1 Participative design of VISPAT 163

8.1.1 First year meetings' overview 164

8.1.2 Communicating the design 168

8.1.3 The designer's role 168

8.2 Evaluation of VISPAT 170

8.2.1 Quantitative and qualitative findings 172

8.2.2 Evaluation tasks 174

8.3 Summary and conclusions 178

8.3.1 Future plans for VISPAT 179

8.3.2 Summary 179

9. The Tuner's Workbench **180**

9.1 Preliminary specification of TWB 182

9.1.1 Tuning record creation and management 185

9.1.2 Assessing tuning progress 188

9.1.3 Sharing tuning knowledge 189

9.2 Conclusions and future work 190

10. Conclusions **194**

10.1 Future work 198

A. Tuning in the small **200**

A.1 Difficulties of tuning 200

A.2 Cause/effect chasm 202

A.3 Faulty assumption/model 202

A.4 Inapplicable tools 204

A.4.1 Tool requirements 211

A.4.2 Case study I: tool evaluation 213

A.5 Change 217

A.6 Training in tuning 220

B. Tuning in the large **222**

B.1 Problem solving 222

B.1.1 Experimentation techniques 225

B.1.2 Case study I: verbal protocol analysis 229

B.2 Documentation methods 231

B.2.1 Reasons for documenting tuning 243

B.3 Tool requirements 244

C. Tuning across organisations, people and machines **248**

C.1 Different attitudes to tuning 248

C.2 Case study I: the LIBRA project 251

C.3 Case study II: development in X 252

C.4 Tuning within the group 254

C.4.1 Somebody else's code 256

C.5 Case-study III: EPCC User Support Service 258

D. Findings of VISPAT's empirical testing **264**

D.1 List of tasks 264

D.2 Qualitative and quantitative findings 265

D.2.1 Task 1 265

D.2.2 Task 3 267

D.2.3 Task 7 270

D.2.4 Task 9 272

E. Publications **275**

Bibliography **276**

List of Figures

1–1 Structure of investigation 3

2–1 Pablo analysis environment 14

2–2 Barrier visualisation in Gthreads 19

2–3 A collection of Paragraph displays 20

2–4 The Source code display of VT 21

2–5 The Navigation display in MPP Apprentice 24

2–6 Data presentation model of Maritxu 25

4–1 Profile of respondents 42

4–2 Tool requirements 63

4–3 Tool requirements expressed by participants in the design meetings of VISPAT (Chapter Seven) 64

4–4 Role of tuning tools changes 73

4–5 Model and processes of tool development 74

5–1 Profile of respondents 79

5–2 Resolution of a performance problem 81

5–3 Tasks associated with tuning in the large 82

5–4 Tuning notes of Informant 46 ([e626]) 95

5–5 Excerpt from tuning notes 99

6–1 What tuners need to know 131

7–1 Overview of VISPAT 135

7–2 Phase hierarchy in a CHIMP/PUL program 137

7–3 A phase node 142

7–4 A phase tree 142

7–5 Communication event list 143

7–6 Navigation display 145

7–7 The central role of the Navigation display 146

7–8 Graphical encoding of MPI point-to-point communications 152

7–9 Components of the Communication display 154

7–10 Membership matrix display 155

7–11 Two instances of the Profile display 156

7–12 Unfolded phase hierarchy of predator-prey program. 157

7–13 Top level phases of predator prey program. 158

7–14 Phases Mesh, Land and Comm have been filtered out. 158

7–15 Phase Loop has been expanded to its subphases. 158

7–16 Phase hierarchy of predator-prey program. 159

7–17 Phase hierarchy of predator-prey program (version two) 160

8–1 Meeting dates in a period of four months. 165

8–2 Basic tool functionalities. 167

8–3 A model of system acceptability 171

8–4 Animation control buttons 174

9–1 Components of TWB 183

9–2 Tuning in the large 184

9–3 Tuning record creation and management 185

9–4 Tuning experiment (as HTA) 187

A–1 Communication display in Paragraph 206

A–2 Length of message queue per node 210

A–3 Communication map 210

List of Tables

4–1 Sources of difficulties in tuning 44

4–2 Revised framework 49

4–3 Tuning tool use 56

4–4 Tuning training and access to documentation 69

5–1 Duration of tuning 89

5–2 Size of parameter set 90

5–3 Documentation methods 94

5–4 [Be627] Diary of porting IFS to CS-2 96

5–5 Informant 45's notes on a series of tuning sessions [e625] 97

5–6 Reasons for documenting tuning 101

6–1 What tuners ask 128

7–1 MPI collective operations and corresponding labels 153

8–1 Ways of interacting with the interface features of the tool. 173

8–2 Different kinds of utterances. 173

B–1 [Be627] Diary of porting IFS to CS-2 247

D–1 Ways of interacting with the tool. 265

D–2 Different kinds of utterances. 265

D–3 Quantitative findings from Task 1. 266

D–4 Quantitative findings from task 3. 267

D–5 Quantitative findings from task 7. 270

D–6 Quantitative findings from task 9. 273

Chapter 1

Introduction

Parallel programming is inherently more complex than its sequential counterpart. People who have previously developed sequential codes have now to apply data and work decomposition techniques, be concerned with synchronisation and communication issues and often know a lot about the operation of parallel machines in order to produce efficient parallel code. To this end, performance analysis and tuning is often undertaken to ensure that the power of the underlying hardware is fully utilised. Self evidently, performance is a key requirement for parallel software.

Crucial to the success of tuning is the availability of adequate support tools and considerable effort and expense have been devoted to their design. A large number of performance analysis tools have been developed which claim to assist the tuner in the tuning task. One common application of performance analysis tools is in the visualisation of performance data resulting from program execution monitoring, with the aim of helping the tuner to discover and analyse areas of poor performance. However, the results of tool use are disappointing. Users complain that tuning tools are hard to learn and use and do not provide the information they really need. Pancake reports that 64% of 448 participants of Supercomputing '93 had never used an externally developed tuning tool [133]. The root of the problem lies in the lack of a systematic study of tuners and of how tuning is performed.

The work described in this thesis is an investigation of tuners and tuning. Questions that need to be asked are how parallel users tune their codes and what their requirements of tool support are. Most empirical studies of programmers leave a huge gap when it comes to observing and recording the practices of real programmers and more specifically of real parallel programmers. The majority of these studies have used students as their subjects and examined the cognitive issues involved in cases of sequential program understanding or debugging. Parallel programming introduces a number of new different issues which need to be investigated. By observing tuners and their work practices, this project attempts to answer some of the above questions and reveal others.

First, Chapter Two reviews a number of state-of-the-art approaches to tool design and the issues that tool developers have to address.

Chapter Three draws on some empirical studies of programmers to create a framework for studying tuning.

An initial, informal investigation was conducted by interviewing local tuners and by soliciting views of the wider tuning community via the medium of Internet bulletin boards, the World Wide Web and electronic mail. This highlighted a number of issues which were later explored more systematically through questionnaires and interviews. These findings are presented and analysed in Chapter Four and in two subsequent chapters. More specifically, Chapter Four takes a closer look at how the task is performed currently. The problems in relation to the use of existing tuning tools along with tuner requirements of current and future tools were collected and are presented. The impact that hardware and software support may have on tuning is also discussed.

Discussions with local and non-local tuners revealed an aspect of the tuning task which until recently was almost completely neglected by tool developers. Most of the tools were concerned with supporting the tuning unit cycle i.e., the effort to change the values of performance determining parameters in order to improve performance on the basis of an observation of the program behaviour (tuning in the small). What this work demonstrates is that an equal level of attention should be paid to the implications that arise from having to repeat the tuning unit cycle a number of times in order to reach an acceptable performance (tuning in the large). Current tuning practices in relation to the management of the process and its results were investigated and are presented in Chapter Five. Chapter Six examines tuning practices in the context of collaborative work environments. In particular, the communication and documentation needs of tuners are investigated in the context of program development and tuning performed by many people within or even across organisations. Figure 1–1 shows the structure of the investigation presented in Chapters Four, Five and Six.

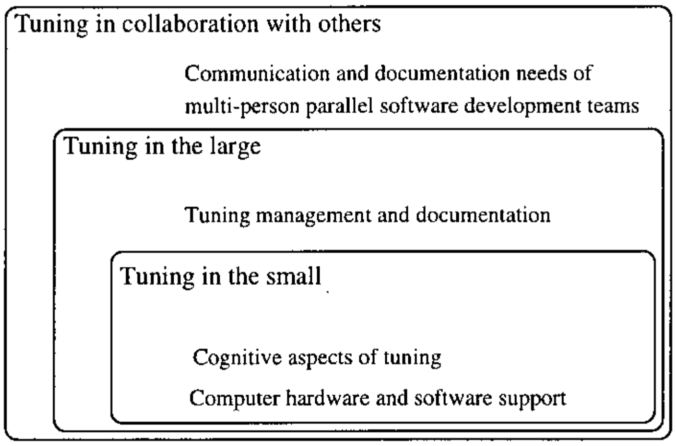

Tuning in collaboration with others
Communication and documentation needs of multi-person parallel software development teams
Tuning in the large
Tuning management and documentation
Tuning in the small
Cognitive aspects of tuning
Computer hardware and software support

Figure 1–1: Structure of investigation

Subsequent chapters of the thesis propose two tools for addressing some issues revealed by the investigation. Some of the tuner requirements mentioned in Chapter Four were re-inforced whilst co-ordinating a project at the local supercomputing centre (EPCC) to design and build a tuning tool for the analysis of parallel programs running on a network of workstations. A number of tuners within EPCC were interested in participating in the tool design process. Their requirements were realised in the design of VISPAT (**VIS**ualisation for **P**erformance **A**nalysis and Tuning) – a tool for supporting tuning in the small – which is presented in Chapter Seven.

The participative design process of VISPAT is described in Chapter Eight. Other local parallel program tuners participated in an informal evaluation of VISPAT. The evaluation of VISPAT gave those who did not participate in the design process a chance to voice their requirements and most importantly showed areas where the tool could be improved in future iterations of the design cycle.

The findings of the investigation described in Chapters Five and Six comprised the basis on which a preliminary specification of a tool for supporting tuning in the large is proposed in Chapter Nine.

Finally, Chapter Ten summarises the results of this research and is concerned with further work.

Chapter 2

Parallel Program Performance Analysis Tools

This chapter provides an overview of contemporary performance analysis tools. The next four sections consider four distinct issues in the design of tuning tools:

Performance data collection. There are a number of techniques for gathering informative data whilst intruding as little as possible into program execution (probe effect). The first section presents some of the data collection techniques adopted by current tools.

Performance data analysis and filtering. The amount of performance data obtained can be very large, particularly in the case of massively parallel systems. This data has to be filtered and structured in order to convey intelligible information on the program behaviour. Section two describes some of the most prevalent analysis and filtering schemes.

Performance data visualisation. Advanced visualisation techniques are employed to assist in the interpretation of the performance data collected. It can often be very difficult to relate the low-level account of the program behaviour to the source code. This problem is aggravated by the tendency for parallel programming environments to provide progressively higher-level programming facilities. Whilst this is of great assistance in the design and coding stages of program implementation, it often makes tuning more difficult, with users having to relate low-level events to increasingly abstract program representations. Section three describes several visualisation displays employed by current tools.

Automatic performance diagnosis. The latest tool research focuses on building tools which can assist in performance diagnosis by borrowing technology from other fields such as software engineering, databases and expert systems. Such tools attempt to undertake the task of finding the reason for a performance bottleneck and guide users in their effort to improve program performance. Section four illustrates some of the approaches.

Work on performance analysis tools has still a long way to go to close the gap between tool functionality and user requirements. Traditionally, tuning tools have employed advanced graphical techniques for user interface design and trace data visualisation. In spite of this, however, there is evidence that they are unable to assist effectively in the task of performance optimisation [133]. Due to the lack of tool integration and consensus amongst tool developers, a lot of implementation efforts overlap whereas a lot of issues remain largely neglected. The fifth section presents some of the first examples of tool integration. The last section discusses some of the issues that tool design should take into account.

2.1 Performance data collection

Any attempt to observe the execution of a program changes its behaviour as the monitoring tool shares the system resources with the program. Existing monitoring tools vary in the kind of performance data they collect and the amount of overhead they introduce. Monitors can be classified according to the way they are implemented and the way they are activated [81].

2.1.1 Implementation level

Depending on the level at which a monitor is implemented, it is classified as a hardware, software, or hybrid monitor.

Hardware monitors

Hardware monitors are implemented by additional pieces of hardware that are embedded within the parallel system. Measures such as MIPS, Mflops, program counter samples, memory and network traffic can be gathered by a hardware monitor. For instance, a performance chip collects such information for the IBM Research Parallel Processor Prototype (RP3) [90]. An alternative approach is described in [1] where the ATUM tracing system modified the microcode on a VAX to record all instruction and data references in a reserved portion of memory. Hardware monitors have a low overhead in program execution but higher-level displays such as algorithm animation or other application specific visualisations require the specification of higher-level events. This is possible with software or hybrid monitors.

Software/hybrid monitors

Software or hybrid monitors insert monitoring instructions at strategic points in the operating system, in the run-time system, or in the application program (*program instrumentation*) which can generate program related information. These instructions write performance information into a memory area of the system under study (software monitoring), or to a hardware interface that is accessed by a hardware monitor (hybrid monitoring) [92]. In case of a software monitoring tool, performance data can be written to local buffers on every node which are flushed at regular intervals and share the same interconnection network with the application program.

2.1.2 Triggering mechanism

Depending on what triggers their activation, monitors can be either time-driven or event-driven.

Time-driven monitors

Time-driven monitors are activated at fixed time intervals by clock interrupts. The program counter is sampled periodically on every node of a parallel system in order to time parts of the code. Sampling is performed by the operating system and can be used in conjunction with information provided at compile time in order to give timing estimates at the procedural level [57]. The sampling frequency may not be high enough to provide accurate estimates but the method has a low implementation cost and exposes the most expensive, i.e. time-consuming, parts of the program. More sophisticated profilers require that the program is instrumented. Small pieces of code are inserted in the program source in order to query the system clock and count the number of times a piece of source code is called. Alternatively, as in MPP Apprentice [171], the calls are inserted by the compiler in the intermediate code. Profiling tools introduce a low overhead and the volume of data they collect scales well with the number of processors in the system.

Event-driven monitors

When the monitor is activated by the occurrence of specific events, it is called an event-driven (or event-based) monitor. This is the most detailed and the most perturbing technique for collecting performance data. The occurrence of interesting events is recorded during the execution of the parallel program. The program behaviour can be represented by the sequence of these events. The type of event information obtained depends on whether hardware or software monitoring is used. Every event generation instruction introduces a delay in the execution of the parallel program. Accumulated delays may change the order of the program events and result in prolonged execution times.

2.1.3 Instrumentation techniques

The level at which instrumentation is placed (operating system, run-time environment or application program) determines the information that can be generated. Instrumentation of the operating system can gather data about events such as process creation, scheduling, context switching, etc. These events can highlight the interactions between the system software and the hardware. Instrumentation of the run-time environment can provide information about the state of various run-time queues, the acquisition and release of locks, arrival at and departure from barriers, and procedure calls and returns. This type of data can be used to associate performance problems with specific sections of the application code and to present aspects of the run-time system itself. Instrumentation of the application program can link particular parts of the program code to the operating and the run-time system events. At the same time, higher level information about abstract events can be gathered. The insertion of instrumentation code is most commonly automated [102] and users do not have to modify their programs.

A common approach is to instrument the communication libraries employed to provide the communication and synchronisation operations in the parallel program. The Portable Instrumented Communication Library (PICL) runs on a number of message passing systems and produces trace files consisting of send and receive events as well as communication statistics [54]. The Message Passing Interface Standard (MPI) [113] specifies a standard profiling interface which allows tool developers to easily attach their instrumentation instructions without the need to have access to the MPI source code [74,89].

Xab [16] is a monitoring and visualisation tool for programs using the Parallel Virtual Machine library (PVM) [159]. Trace data is generated by instrumented versions of the PVM routines. The instrumented routines, apart from performing the application's message operations, use PVM to send event tracing messages to a monitoring process. The event tracing messages contain information such as the event time stamp, the event type and event specific data. PVM also facilitates the communication and synchronisation among the distributed components of the monitoring tool [134,168].

Other systems support instrumentation at several levels. For example, in DELTA-T [130], a tool for programs running on multi-transputer systems, performance data is gathered in two ways: by instrumenting interesting parts of the program; and by observing the hardware scheduler. A monitoring process is introduced on every node of the Intel Paragon system [15,142] for instrumentation purposes. The monitoring process calls a library that generates trace data which is sent to event servers. The event servers post-process the trace data and write it to a trace file.

PERFSIM [165] uses an alternative data collection method which combines modelling and measuring. PERFSIM executes the sequential part of a CM-Fortran program and estimates the run-time of all the communication and computation operations involving vectors. The estimation is based on an analytical model of the machine architecture and on compiler information.

Selective instrumentation

The huge volume of data generated during execution necessitates that instrumentation is inserted selectively:

Instrumentation can be inserted in the program during its development stages, and can be removed once the program is implemented. Instrumentation can be added before compilation, during compilation, after compilation or during run-time.

It can reside permanently in the code, but can be switched on and off.

It can be inserted dynamically or optimally into interesting parts of the program. In VISTOP [13], the user may choose to instrument only a subset of the program objects, thus, improving the scalability of the visualisation displays which depict only the events relevant to the objects characterised as interesting by the user[1].

Another technique reduces the points where instrumentation code has to be added by analysing the relations among the blocks of instructions in the program [7,102]. A different approach is followed by Paradyn [118], a tool targeting a number of programming models, languages and architectures. Paradyn inserts minimal instrumentation into the code to identify high-level problems such as too much synchronisation blocking. Having identified a particular problem, more detailed instrumentation is inserted to find the

[1]Further reduction of the amount of performance data can be achieved by switching the instrumentation on and off during execution.

cause of the problem. Interesting program behaviour may be missed by employing dynamic instrumentation. Paradyn is more appropriate for applications that take hours or even days to complete and is able to capture information on repeated events that have a large enough accumulated impact on the program execution.

2.2 Performance data analysis and filtering

This section deals with the analysis and filtering mechanisms adopted by various performance analysis tools in order to reduce the volume of performance data the user has to consider and transform it to meaningful information. Part of the analysis of the data aims at changing it into a suitable form that can be stored as a relational database or a hierarchical graph. Due to time constraints, extensive analysis can be performed only by post-mortem systems (see third section). The following subsections give examples of tools employing three types of analysis and filtering techniques:

Data analysis. The tool facilitates operations performed on the trace data.

Selection. The tool enables the selection of program or system entities for which performance data can be displayed.

Source code reference. The tool relates low level performance data to the source code. This may be done automatically or the user may specify a collection of interesting events which constitute a phase.

It is during the analysis stage that restoring of the order of the events is attempted. The lack of a global clock in a parallel machine makes it quite hard to achieve a total event ordering. In the absence of synchronised clocks running at the same rate and having sufficient resolution, some other techniques of restoring the order of the events are employed such as the "happened before" relation introduced by Lamport [101]. That is, if an event *e1* has an effect on an event *e2*, then the event *e1* has to be visualised before *e2*.

2.2.1 Data analysis

In Pablo [137,138], trace data processing is performed in an analysis environment consisting of data transformation modules which can be interconnected to each other in a number of ways (Figure 2–1). Modules can perform semantic-independent operations on the input trace data such as averaging or histogramming. The output of a module is the processed data and any unused input parts. Each module consists of a user-defined data transformation function and a system provided interface which accesses the trace data. The user does not need to have any knowledge about the format of the trace file, or any data field information about the event records in the trace file. Via a Graph Editor the user can connect various modules to each other and direct their final output to Pablo's data presentation displays. The abstraction away from the semantics of the trace file is achieved through the Self-Defining Data Format. The trace file consists of a set of record definitions and a stream record of tag/data record pairs. The record definitions define the format of the types of events and the record tag identifies the record type before any processing can be applied to the data record [5].

2.2.2 Selection

SIEVE (Spreadsheet based Interactive Event Visualisation Environment) acts essentially like a sieve that provides a means for selecting events generated by program execution [147]. A program graph display is a representation of the application program source code. The nodes of the graph correspond to function calls in the program. The user can browse the call graph and by selecting a set of nodes selects the corresponding temporal regions within the trace file. Trace data

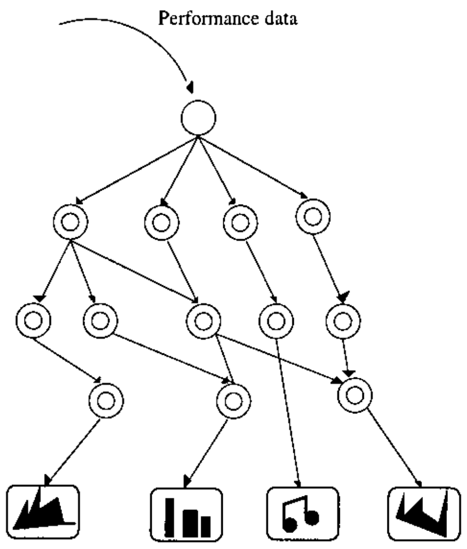


Figure 2–1: Pablo analysis environment

obtained during program execution is highly structured tuple-oriented data. Performance data can be retrieved from the selected regions of the trace file in a way similar to querying a relational/temporal database. Further filtering of the data is achieved by built-in data manipulation functions, operating on the spreadsheet cells. In TATOO [25], the user is presented with a list of performance metrics that have been preprocessed by the tool. Having selected one or more metrics, the user can further choose the appropriate visualisation displays to depict those metrics over time.

In Prism [2], performance data about a processor node is displayed only if this processor belongs to a selected subset of all the processors in the system. Processor nodes can form processor sets depending on their state (running or stopped) and on any other user-defined conditions. In PATOP [63], measurements focus on specific periods of the program execution and on specific nodes where interesting events happen. Further filtering is possible by attributing behaviour to performance metrics such as thresholds of the metric values. The visualisation displays are driven by user-defined expressions which set the thresholds for various metrics.

In Projections [154], a performance tool for programs written in the CHARM language, the execution of the program is divided into equal-length periods of time called stages. The length of the time period is defined by the user to allow for coarser or more refined event filtering. The performance views are organised as a set of overviews and a set of more detailed in-depth views. A performance metric is a function of the stage s and the processor p and ranges over a stage set and a processor set. There are two types of overviews: the first visualises the sum of a performance metric over the whole processor set and the second depicts the sum of a metric on one processor over all the stages of the program. In-depth views can be of one of the following kinds:

- A metric as it varies during a particular stage over a selected processor set.
- A metric as it varies on a particular processor for a selected stage set.
- A metric as it aggregates for a particular processor set over all the stages of the program.
- A metric as it aggregates for a selected stage set over all the processor set.

2.2.3 Source code reference

Existing parallel programming environments provide high-level programming facilities which help the tuner abstract away from the specifics of the parallel machines. This makes tuning even more difficult since the programmer has to relate low-level information to program behaviour. Tools address this issue by presenting the large amount of low-level performance data in terms of the fewer source code structures (source code reference).

In ParaMap [79], a framework is introduced to map low-level performance data to language constructs. Within this framework, called Noun-Verb (NV) model, a parallel program consists of nouns, the structural elements of the program, and of verbs which are the actions performed on nouns or by nouns. A set of nouns and verbs at a particular software or hardware level is called a level of abstraction. Nouns and verbs at one level of abstraction can be mapped to nouns and verbs at another level of abstraction. This mapping is possible because high-level language constructs are implemented by low level hardware and software. The NV model can be used along with any programming model. ParaMap uses the NV model to study CM Fortran programs which run on the CM-5. Three levels of abstraction are identified in the CM-5. The highest level contains the constructs (verbs and nouns) of the CM Fortran language. The second level is the Run-Time System level, RTS. RTS level nouns are all the arrays allocated during execution. These are the arrays of the language level plus all the intermediate arrays created by the compiler. Verbs at the RTS level include all the operations performed on the arrays, such as Copy, Shift, Rotate, etc. At the lowest level of abstraction, nouns are the processor nodes and verbs include Compute, Wait, Broadcast and Point to Point Communications. The user creates a sentence consisting of a noun and a verb and requests from ParaMap the performance cost of the specific sentence. Costs are provided in three forms: as the number of times a sentence is used; as the total time a sentence took to execute; and as a time-histogram showing the cost of a sentence varying over time. The cost of a sentence is amortised over the two lower levels.

The Performance Debugger of EPPP (Environment for Portable Parallel Programming) [41] attempts to provide the user with performance information directly related to the source program. EPPP supports a high level programming model based on a data-parallel programming language called High Performance C (HPC) [40]. HPC is a C based language, which supports data distribution directives similar to the ones found in HPF [140]. The user is able to select array objects within a selected region of the source program. The visualisation displays of the Performance Debugger will show performance data related to the selected source code objects. For example, EPPP's communication display depicts statistics about the inter-processor communication (message numbers and message volume) involved in a specific data distribution.

2.3 Performance visualisation

Performance analysis tools fall in four categories according to the visualisation schemes they provide:

Program specific. These tools show in an application specific way how computation progresses by animating the data structures of the program and the operations performed on them. The animated views can assist in debugging both for correctness and performance [157].

System oriented. These tools focus on the impact the application program has on the parallel system. For instance, SHMAP [39] visualises the memory and cache access patterns in hierarchical shared memory systems. The Front Panel Visualisation tool (FPV) [12] measures and visualises system level performance parameters such as CPU and interconnection network utilisation on the Paragon machine.

System and application independent. The majority of tuning tools fall into this class and provide a repertoire of displays that can generally depict performance data for a variety of systems and applications. The visualisations range from views of the effect the application program has on the system use, to views that represent the communication or synchronisation behaviour of the program.

Meta-tools. Tools in this group facilitate the development of custom visualisation tools. They are more general than the program specific tools mentioned above, since the visualisations created by a user are not necessarily program specific. PARADISE (PARallel Animated DebuggIng and Simulation Environment) [96] and IMPROV [95] introduce a framework in which the application and/or the system behaviour are modelled as a set of visual objects having certain functionalities and interacting with each other. The defined visual objects and their interactions can simulate the events generated by the system under study. POLKA [158] adopts an object-oriented methodology to map program events to visual objects. Object attributes, such as colour and location and object actions such as motion, colour, and resizing, change according to rules that map events to objects, attributes and actions. POLKA was used in the development of two performance visualisation tools, namely the Gthreads [177] and the PVaniM [58,167] systems.

Tools can also be divided by whether visualisation takes place simultaneously with the program execution (on-line) or after it (post-mortem). In the latter case, trace data is gathered in files which can be processed later by a visualisation tool. On-line visualisation can be intrusive when the parallel program shares the same interconnection network with the monitoring and visualisation facility. The volume of information conveyed by the visualisation tool at any time may be too large to be easily interpreted by the programmer without the assistance of "playback" functionalities such as the ones found in off-line tools. However, on-line visualisation, often used in conjunction with dynamic instrumentation, allows the user to observe only the interesting parts of the program execution and reduces the storage requirements of the monitoring tool. VISTOP [13,21] provides two on-line modes of operation; in the first mode, visualisation is driven by breakpoints set in a debugger style of interaction; the second mode is less perturbing, enabling visualisation without breakpoints.

2.3.1 Animation of communications

Animation has been employed by various tools to assist in program execution replay. Communication operations are the most commonly animated events [158]. VISTOP [13,21] (VISualisation TOol for Parallel systems, is part of the TOPSYS (TOols for Parallel SYStems) integrated tool environment for programming distributed memory multiprocessors. TOPSYS supports an object based message-passing programming model implemented with the parallel programming library MMK (Multiprocessor Multitasking Kernel) [14]. Three kinds of objects exist within the MMK model: tasks, mailboxes and semaphores. A parallel program consists of multiple instances of objects which interact with each other. Tasks communicate and get synchronised via mailboxes and semaphores respectively. All interesting MMK objects are displayed by little icons showing the class and the name of an object. If a task communicates or waits for a mailbox or a semaphore the corresponding task icon moves itself into a queue of tasks waiting at the communication object.

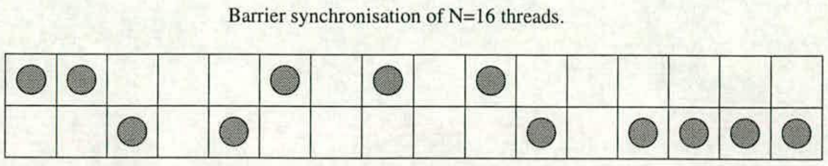


Figure 2–2: Barrier visualisation in Gthreads

Gthreads [177] is a monitoring and visualisation tool for C programs using the pthreads library [77] on the KSR machines or other thread based machines. Gthreads provides a view of the barrier synchronisations that take place during execution (Figure 2–2). Each barrier is represented by a 2 x N array of boxes where N is the number of threads in the program. The first row of boxes in the array represent the check-in phase of the barrier operation for every thread. If a thread joins in the barrier, the corresponding box is filled by a small circle. The second row of boxes depict the check-out phase of the barrier. When a thread checks-out of a barrier, a circle appears in the corresponding box.

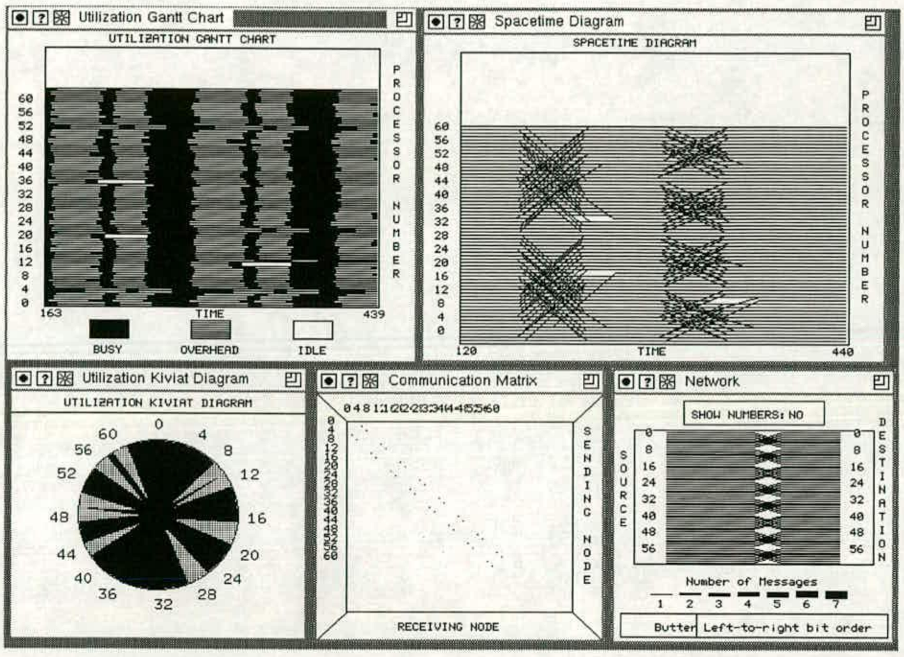


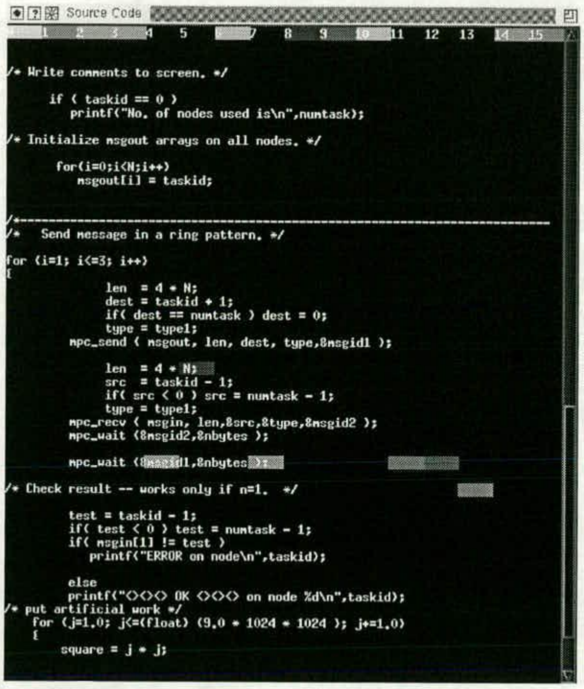
Figure 2–3: A collection of Paragraph displays

The Feynman or Space-Time display, first used by ParaGraph [55,65], can be found in many performance analysis tools [173,56,60] (Figure 2–3). The execution thread for each processor is represented by a horizontal line, drawn from left to right, which changes colour to indicate whether a processor is active, idle, or waiting for a send or receive to complete. Message operations are depicted by lines which connect the two processors communicating. The points where the lines meet the processor lines mark the receive or send times. The display reveals processors idle, waiting for a blocking communication to complete.

The Source display identifies the last communication event processed by VT[2] for

[2]VT is a proprietary tool which runs on the IBM SP-2 parallel machine. It was not possible to find a reference for this tool.

each process. Each line of the source code is segmented in N segments where N is the number of processes in an SPMD program. When VT processes a communication event, it highlights the appropriate segment. Colour is used to differentiate among the segments of different processes (Figure 2–4).



```
/* Write comments to screen. */

     if ( taskid == 0 )
        printf("No. of nodes used is\n",numtask);

/* Initialize msgout arrays on all nodes. */

      for(i=0;i<N;i++)
         msgout[i] = taskid;


/*-----------------------------------------------------------------------
/*   Send message in a ring pattern. */

for (i=1; i<=3; i++)
{
              len  = 4 * N;
              dest = taskid + 1;
              if( dest == numtask ) dest = 0;
              type = type1;
         mpc_send ( msgout, len, dest, type,&msgid1 );

              len  = 4 * N;
              src  = taskid - 1;
              if( src < 0 ) src = numtask - 1;
              type = type1;
         mpc_recv ( msgin, len,&src,&type,&msgid2 );
         mpc_wait (&msgid2,&nbytes );

         mpc_wait (&msgid1,&nbytes );

/* Check result -- works only if n=1.  */

         test = taskid - 1;
         if( test < 0 ) test = numtask - 1;
         if( msgin[1] != test )
            printf("ERROR on node\n",taskid);

         else
         printf("<><><> OK <><><> on node %d\n",taskid);
/* put artificial work */
    for (j=1.0; j<=(float) (9.0 * 1024 * 1024 ); j+=1.0)
    {
        square = j * j;
```

Figure 2–4: The Source code display of VT

2.3.2 Program displays

Program graphs, Gantt charts and data access displays fall into this class of displays.

Program graphs

In PV [91], a static call graph has its nodes coloured according to: the number of processors active in each node; the number of times a processor has arrived at each node; or the amount of time spent in each node aggregated across all processors. In

Prism [2], the call-graph nodes are accompanied by a percentage indication which shows the time each node spent utilising a specific resource of the system. In VISTOP [13,21], the program graph is expressed in terms of the objects provided by the MMK [14] programming model. The graph is animated in order to depict dynamically the creation of the communication and synchronisation objects in the program.

Gantt charts

DELTA-T [130] uses a Gantt chart display to denote the changes in the communicating processes states. In VISPAT [74,73], a gantt chart animates the occurrence of MPI and user-defined phases in the parallel program.

Critical paths

Critical path analysis [174] is concerned with identifying the program regions which most contribute to the program execution time. The critical path analysis constructs a directed acyclic graph which depicts the synchronisation and communication dependencies among the processes in the program. Dependencies are message operations, semaphores, barriers or locks. Each edge in the graph is assigned a weight proportional to the duration of the operation represented by the edge. The time spent on a edge may be CPU bound or may be the time required to send a message. The path with the longest weight sum is the critical path.

Data access displays

The Performance Debugger of the EPPP environment [41] employs an animated two-dimensional display which shows the distribution of arrays on a set of processors and their access with respect to computations performed by one processor over time. Each cell of the two-dimensional display is devoted to an element of the array. Different sections of the display correspond to parts of the array distributed on different processors. The colour of an array cell can distinguish whether an access is local or remote and whether it is a read or write. The programmer can analyse the access patterns so that better array distributions can be achieved in order to reduce inter-processor communication.

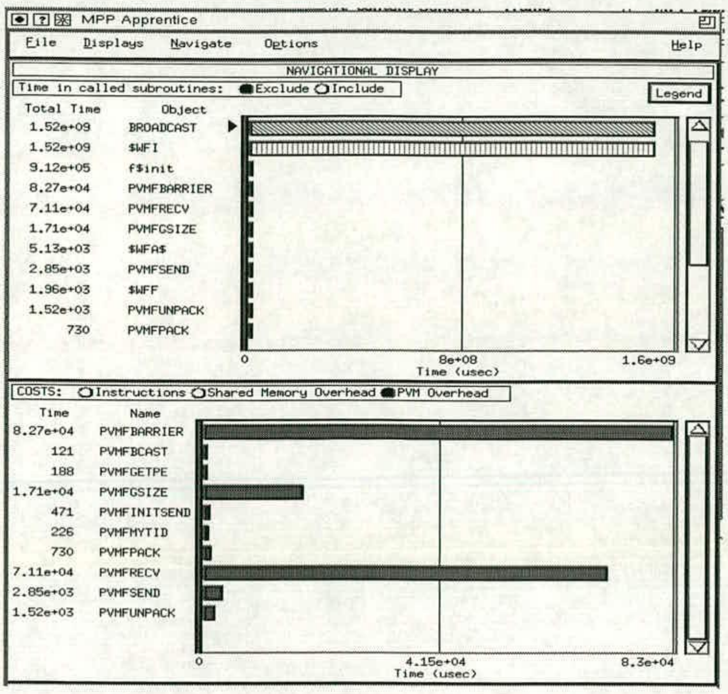
2.3.3 Aggregated metrics

Static bargraphs and plots can represent the values of performance metrics as they accumulate over a period of time. Animated bargraphs can show how measurements evolve over time.

Histograms and bargraphs

A hierarchy of histograms in Prism [2] show where the program spends its time. The top-level histogram shows the time taken by a program utilising each resource of the system. Clicking on a histogram reveals the usage of the corresponding resource by each subroutine of the program. The last level of the histogram hierarchy is the source-line histogram. For a particular resource in a particular subroutine of the program, a source-line can be accompanied by a percentage bar showing the amount of time the particular source-line contributed to the use of the specified resource. MPP Apprentice [171] visualises profiling information in an hierarchical way. At the top level, the Navigation display lists the subroutines of the program sorted from the most to the least critical. Next to each subroutine name, a bar indicates the time this routine took to complete (Figure 2–5).

The MasPar profiler [27] provides profiling histograms at the routine and statement level through the graphical interface of the debugger. The user may stop the program execution and see the cumulated profile information up to the point execution has proceeded. In ATExpert [97], a histogram shows the contribution of the serial and parallel regions of the program to the program speed-up.



Total Time	Object
1.52e+09	BROADCAST
1.52e+09	$WFI
9.12e+05	f$init
8.27e+04	PVMFBARRIER
7.11e+04	PVMFRECV
1.71e+04	PVMFGSIZE
5.13e+03	WFA
2.85e+03	PVMFSEND
1.96e+03	$WFF
1.52e+03	PVMFUNPACK
730	PVMFPACK

Time	Name
8.27e+04	PVMFBARRIER
121	PVMFBCAST
188	PVMFGETPE
1.71e+04	PVMFGSIZE
471	PVMFINITSEND
226	PVMFMYTID
730	PVMFPACK
7.11e+04	PVMFRECV
2.85e+03	PVMFSEND
1.52e+03	PVMFUNPACK

Figure 2–5: The Navigation display in MPP Apprentice

Spreadsheet - XY contour plot

In SIEVE [147] the programmer can create customised performance views by the combined use of the spreadsheet facility and the XY plots. After having applied any data manipulation functions on the spreadsheet cells, the programmer may create XY plots by selecting specific rows and/or columns of the spreadsheet.

2.3.4 Animated system views

A two-dimensional arrangement of processor cells and interconnection network is commonly found in displays which depict system utilisation. In DELTA-T [130], all the transputer nodes are arranged in a two-dimensional way. The display is animated with each node changing colour according to its utilisation. Communication among links and processes is shown by lines connecting the nodes. Colour is used to denote variations in the link utilisation. The System Performance Visualiser (SPV) is designed to portray the overall system usage [15,142]. An overview display shows utilisation of all nodes, whereas more detailed ones show the utilisation of the message-passing processor, the Bus and the Network Interface for the Intel Paragon.

Data presentation model of Maritxu

Based on vision theory, Maritxu [175,176] proposes an independent visualisation schema which is a spatial two-dimensional arrangement of processor nodes. Instead of confining the visualisation to one metric shown by the colour of the node cell, Maritxu depicts a number of icons on each cell, each dealing with one performance metric. The attributes of the icons such as colour, shape, orientation can convey the variations of the performance metrics. Figure 2–6 shows some of the performance metrics that can be assigned on a processor cell, for a transputer platform. Animation can show how the performance metrics change over time.

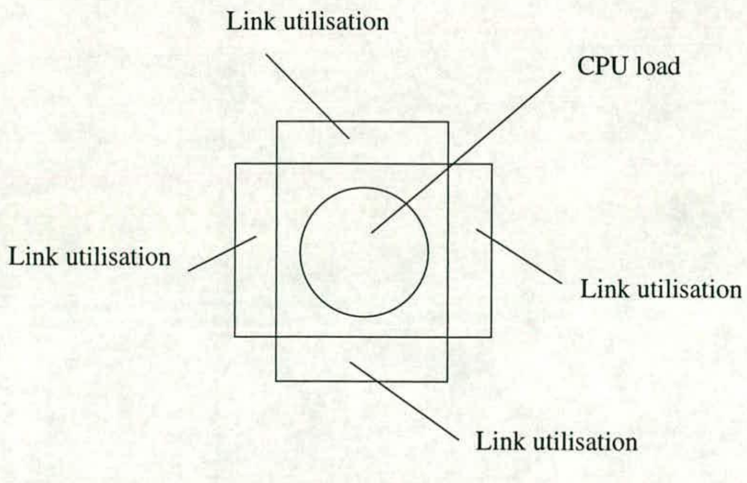


Figure 2–6: Data presentation model of Maritxu

Machine views or processor views

VISTA [145] introduces a version of ParaGraph's [55] Processor views, called Machine views, for SPMD/MIMD and SIMD programs. The Processor views display is a two-dimensional array of NxN processors. Each processor is assigned a cell in the NxN matrix and the colour of the cell denotes whether the processor is busy, idle or communicating at a particular moment. The animation of the display provides a series of snapshots of the system over time. VISTA uses Machine-Views for showing the utilisation of a large array of Processor Elements (4096 PEs) of a MasPar MP-1. The space occupied by each cell is naturally a lot smaller, and differences in shading denote different PE utilisations. The variation of system utilisation over time is shown either by attaching Machine views together in a "filmstrip" manner or by adding a third dimension to the two-dimensional display.

Memory access patterns

SHMAP, the (Shared-Memory Access Pattern) tool, visualises the memory access patterns of parallel algorithms in a Fortran setting [39]. Memory is represented two-dimensionally on two separate displays visualising the read and write operations respectively. On a memory access the corresponding memory element is illuminated. The illumination fades gradually to identify recently accessed elements. A similar display is used for visualising cache accesses in a hierarchical memory environment.

2.3.5 Alternative representations

Scientific visualisation, virtual reality and auralisation are some of the alternatives proposed to meet the increasing requirements for scalable and meaningful data representations:

Scientific data visualisation. With the advance of parallel computing the requirements for performance visualisation become more demanding. Work in [62] investigates the feasibility of using scientific data visualisation software to generate new performance visualisations.

Virtual reality. Head-mounted displays are introduced in Pablo [137,138]. A small head-mounted display conveys stereoscopic three dimensional performance graphics. When an interesting event requires additional instrumentation and more thorough visualisation, graphics are conveyed to the high resolution workstation displays.

Auralisation is proposed in [51,80] as an alternative to visualisation. The scalability of the combined use of sound and graphics is examined in [52]. Each processor is mapped to a different note which is played whenever this processor sends or receives a message. The send-notes can be played on different stereo channels or with different instruments from the receive-notes. Massively parallel systems require a more scalable scheme because the generated sounds can be unpleasant. Scalability is achieved, as processors are separated into groups, and each group is assigned a note. Inter-group communication is directed to one channel while intra-group communication is directed to another one. The additional information conveyed by sound can reduce the complexity of the visual displays.

2.4 Automatic performance diagnosis

A number of tools support performance diagnosis in ways described below. ATExpert [97] is a performance analysis tool for programs which use the Autotasking programming model on the Cray Y-MP C90. ATExpert goes further than just presenting performance data. It attempts to interpret the behaviour of the program on behalf of the user and suggest the appropriate actions which can improve the program performance. ATExpert uses a rule-based expert system to make observations. For a given region of code, it gathers data such as the actual speed-up, the overhead time, the serial time and the number of processors. A subset of rules are chosen according to what dominates execution. Having selected the set of rules, patterns in the performance data are used to determine the possible causes of performance problems. Observations and suggestions are presented through graphics and text.

Paradyn's [118] Performance Consultant is based on a "why, where, when" (W3) search model to assist the programmer in locating the causes of bad performance. The tool tries to answer these questions on behalf of the user. First, the system makes a number of hypotheses as to what could be the cause of the problem. Gathered trace data is examined to test and validate these hypotheses. Once a hypothesis is validated, the tool attempts to identify the location in the program where the problem appears. Finally, the "when" question is answered by observing the distinct phases of a program execution.

IPS-2 [78] can work in comparison mode and have its displays show simultaneously performance data from various runs. In TraceView [114], a tuning session consists of a set of open trace files, a set of views and a set of displays. A view is defined as a sub-region of the trace file comprising of interesting events. Unwanted events are filtered out of the event stream. A session manager enables the user to save a tuning session for later use. In this way the user is able to compare trace files from multiple runs of the program.

On-line steering can improve the performance and the functionality of parallel application programs. On-line steering is defined as the on-line configuration of programs in order to affect execution behaviour and performance. In Falcon [60], the user can improve, for example, the load balance by interactively changing program parameters such as domain boundaries.

2.5 Tool integration

There have been a number of cases where initial work towards tool integration has been done. Tool integration can be of three kinds:

- tools can share information (data sharing),
- tools are managed in a common framework (control integration) or
- tools have a common user interface (presentation integration).

The toolkit approach to tool design whereby tuners may extend the system by providing their own trace data processing and visualisation mechanisms is one technique for tool integration, but some tools are designed in a way that a great deal of end-user programming is required to extend their functionality or integrate them with other tools [65]. The adoption of an "open tool architecture" is the way to ensure tools from different developers are interchangeable. Alternatively, the Self-Defining Data Format of Pablo [5,137] abstracts away from the semantics of the trace events, thus facilitating tool integration and extensibility.

The design technology of existing tools should be improved to allow them to be successfully integrated with parallel programming environments [9,91]. The approach of the TAU (Tuning Analysis and Utilities) project is that of integrating the tuning tool to the pC++ portable parallel programming environment [22,121]. pC++ is an extension of C++ and facilitates the creation of distributed data objects. TAU interacts with the pC++ compiler in order to access information about objects in the application program and with the pC++ run-time system to get profiling and tracing information. TAU's trace data can be converted by built-in utilities into one of the following data formats: SDDF[5]; ALOG [67]; and TDL/POET [120], allowing TAU to share tracing information with the Pablo [138], UpShot/ALOG [67] and SIMPLE [119] tools respectively.

Other tools attempt to enable users to work in their favourite environment of debuggers and performance visualisers by employing control integration [103]. For example, VIZIR [64] consists of a number of Tool Interfaces (TI) that allow interaction with each of the integrated tools. VIZIR caters for PVM applications that run on distributed systems and the tools integrated include: Hewlett Packard's DDE, IBM's XDE, Matlab, ParaGraph and Gnuplot.

Performance visualisation could benefit from the integration with scientific data visualisation software. So far performance visualisation has been limited to two-dimensional views. For example, Hackstadt and Malony demonstrate how a data visualisation tool could provide a rapid prototyping environment for creating and interacting with multi-dimensional performance views [62]. Raw trace data is structured so that it can be processed by the visualisation system. Automating and formalising the interaction between the performance data generation environment and the visualisation software could be the next step towards integration.

Poirot seeks to pursue integration with performance analysis tools in order to apply its automated performance diagnosis methods to a number of targets [66]. This approach relies on research in software development environments and databases which provide access to tools, programs and data independent of tool command syntax and data format.

2.6 A Design Agenda

Many performance analysis tools have been described in the literature with examples of their use. These typically illustrate indicative situations and codes with regular, easy to understand behaviour which does not correspond to large, real world application programs. In practice, the problems that programmers have to face may be more complex and the amount of data they have to consider very large.

Tool developers have a somewhat restricted understanding of the requirement for ease of use. For example, the developers of DELTA-T consider that the requirement for ease of use is met by simply allowing users to monitor the execution of their programs without having to insert any extra statements in the source code [130]. Tool designers are motivated by technological challenges and design systems according to their own personal requirements and their intuitions about what will be good for the users. As we have seen, graphical user interfaces and sophisticated visualisation means have been widely employed to facilitate user interaction and interpretation of performance data. Mere use of visualisation and graphical interaction should not be regarded as a panacea and there are a number of issues which should be part of the developer's agenda:

- *Knowing the users and their tasks.*

  Designers should not make assumptions about users and their tasks. Users beyond the immediate tool development context should play some part in design since local users may be more knowledgeable and may have been exposed to the design too much to be representative of the user community and their needs. Design should cater for the experienced as well as for the novice user. Experienced users may need to conduct more detailed analysis and this can be supported by allowing them to customise the tool and provide application specific visualisations[3]. Novice users may require more assistance in finding and eliminating a performance bottleneck.

- *Providing meaningful visualisations of program behaviour.*

  Visualisation is a powerful tool for data presentation but it should be used with care. For example, Miller gives some guidelines for meaningful and us-

[3]With the advent of programming environments for structured parallel programming, application specific visualisations could be applied to program templates.

able performance visualisation [117]. Usability issues should be considered, for example, program visualisations should scale well with the amount of information that a program run produces. Tools may provide zooming in and filtering facilities [136], but these methods should be used carefully as they are likely to disorient the user from the task in hand if, for instance, too many zooming operations in combination with filtering operations (isolating particular metrics for examination) are performed. Some of the responsibility of identifying performance bottlenecks could be undertaken by the tool, thus reducing the search space that the user has to investigate.

A number of software engineering and performance issues arise as well. Tool response time and space requirements should scale well with the amount of performance data that monitoring generates. Apart from reducing the amount of data and the level of detail that the user has to consider, abstraction and filtering mechanisms should be designed to cater for reducing tool response time as well.

The overwhelming variety of programming, architecture and machine models has resulted in a large number of diverse tools which often fail to meet user requirements and thus, become unusable. Almost ($\simeq$ 46%) of the parallel programmers who participated in the investigation of the tuning task described in Chapter Four did not use any tool at all. The vast majority of tool development has been ad-hoc without complying with any standards or with accepted user-centred design methodologies. This has raised the development cost of the tools and has decreased their life-span since new machines and new programming languages keep emerging. With the hardware platform and programming environment scene changing so fast, portability and extensibility become major issues. Due to the lack of tool integration and consensus amongst tool developers, a lot of implementation efforts overlap whereas a lot of issues remain largely neglected. The following are prerequisites for tool integration to occur [144]:

- the traditional approach of building tools should change,

- there has to be greater consensus about what kinds of tools are useful,
- standardisation in data formats and interfaces is a necessary step,
- an infrastructure is needed so that tool development need not start from scratch for each new machine and
- vendors should realise that good and usable tools are as important as high performance parallel hardware.

The agenda of this work has been to seek an understanding of how tuning is performed and to use this understanding in informing the design of tuning tools. An initial investigation by means of interviews and questionnaires provided the basis of a framework for the analysis of tuning. This analysis pointed to the difficulties that tuners encounter with tools and in performing tuning. The results of the investigation are described in Chapter Four. For example, an issue which has so far been neglected by tool developers is the fact that most of the tools have been data driven i.e., the emphasis of tool design has been on the analysis and visualisation of performance data stemming from one instrumented program run. However, a performance problem is rarely solved by a single tuning cycle. Support is missing for facilitating the process of experimentation and repetition. Chapters Five and Six identify and analyse the requirements for supporting tuning in the large.

Chapter 3

A framework for studying tuning

Parallel computing enjoys significant interest in the scientific community because of the increasing demand for computational speed and the decreasing cost performance ratio of parallel computers. The scientific community is now able to model and solve more accurately larger compute intensive problems. In the quest to produce functionally correct application programs that take advantage of the power of the underlying hardware, usable and efficient software tools and environments are required to assist with parallel program development.

Whenever possible, the development of parallel programming techniques and tools has sought to take advantage of advances in sequential programming. The existence of large "dusty deck" sequential programs motivated the advance of compiler technology [20] in order to identify potential parallel regions in the sequential program and transform them into parallel code (implicit parallelism). However, research in automatic parallelising compilers is far from complete, so much of the work of parallelism has still to be explicitly performed by the programmer. Apart from a few languages designed with constructs (e.g. [6]), the rest are based on existing sequential languages. Languages such as C or Fortran are extended or host libraries to allow for explicit parallelism. Current research in parallel program debugging has allowed the leverage of techniques used in debugging of sequential programs.

In contrast, performance optimisation of parallel programs is dominated by different and more complex problems than its sequential counterpart. In sequential programming the choice of algorithm is the main performance determining decision as the execution time of the program is the sum of the instructions executed whose number can be expressed in terms of the problem size n. In addition, sequential compilers are sophisticated enough to perform code optimisations in order for the code to perform well on the target machine. Thus, it is relatively straightforward to analyse the performance of a sequential program. On the other hand, in parallel programming the choice of algorithm is only one of many considerations. The algorithm should be chosen to map well onto the underlying parallel machine and further performance tuning has to be performed as parallel compilers are not often sophisticated enough to perform extensive code optimisation. Additionally, the execution time of a parallel program is no longer the sum of the instructions, but the critical path [104].

The large diversity of parallel architectures makes it difficult to try to abstract parallel machines for analysis. The PRAM model [88] has been used extensively in the analysis of parallel algorithms but is of little use to the parallel developer as it does not correspond to any real machine. Whereas there are more realistic analytical models [115] being developed, successful analysis of parallel applications is still highly machine specific [99]. As a result, analytical methods are not well suited to cater for all the performance issues implicated in the immense variety of architecture and programming models. Analytical techniques are also very hard to learn and use so as a result, parallel programmers cannot generally start developing their application by forming a theoretical analysis of their algorithm nor can they start from a known optimal algorithm. The most common approach to developing a parallel code is to look for opportunities for parallelism in existing sequential codes. After the program is developed, its performance must be measured and analysed to pinpoint to sources of poor performance and to determine how to modify it to remove them. Performance evaluation, performance analysis, performance debugging and performance tuning are all terms which describe the process of measuring and analysing program performance with the purpose of improving it. Tuning is a difficult task and tuning tools are crucial aids in the development and performance optimisation stage of parallel programs. The work presented in this thesis argues that an in-depth investigation of parallel program development and specifically of the tuning task itself is needed, before any assumptions can be made about the design of tuning tools.

3.1 Studies of programming

In the absence of studies of parallel programmers, work on empirical studies of sequential programmers must serve as a starting point for the specification of usable and useful parallel programming environments. As early as in the mid-eighties, however, the relevance of the up-to-then largely laboratory-based studies of programmers to real world software development problems was questioned. Curtis, for example, stressed that a great body of research concentrated on expert-novice differences in programming using students as subjects and small size programs (e.g. [23,61,69,86,143,156]) [35]. Apart from some notable exceptions e.g. [151], this research failed to observe processes that occur in real world large projects with experienced programmers.

Curtis's observations are even more valid when the relevance of these studies to parallel programming is considered. Computer science students and professional programmers are not representative of parallel programmers dealing with large scientific codes. Parallel programmers may have been using sequential machines for years in academic or industrial organisations so they cannot be compared to first and second year computer science students. In addition, they may be infrequent programmers without the characteristics of full-time programmers.

Studies have considered only fragments of sequential programs and focus mainly on issues of understandability. The inherent complexities of parallel programming introduce a number of other issues: having to learn a lot about e.g., how the hardware operates; having to keep up with fast changing programming environments; and having to produce faster solutions, to name but a few.

The investigation described in this thesis is among the first examples of research to focus on practices of "real" parallel programmers [35,132,133]. Pancake et al. [133] asked 448 participants of Supercomputing '93 whether they had used any computer based tool during the stages of parallel program development, namely: fixing a serial program; writing a parallel program; improving the basic model; debugging; tuning performance and setting up program runs. Among other results, it was reported that almost 35% of the parallel developers asked used an internally developed tool for performance tuning, while 29% had used no tool at all. The remaining respondents were using an externally developed tool. Pancake admits that the results of the survey set optimistic upper bounds regarding the use of current tools, since the people attending the conference were probably more computer science oriented than those who would attend a purely scientific meeting. Past results of a smaller survey among more conservative Fortran users showed that almost 90% relied exclusively on PRINT statements instead of using a debugging or a performance tuning tool [132]. These results demonstrate that the large number of tuning tools described in the literature (Chapter Two) are not used by real parallel programmers.

The investigation described in the following chapters draws upon the experiences of a much smaller number of parallel computing users, but focuses entirely and in more detail, on performance tuning and spans across a wider range of issues. Whereas Pancake's survey investigated the use of parallel programming tools, this work studied and analysed tuners' work practices as they are formed within the context of current tool support and collaborative program development. One starting point for this research has been one of the few studies of real programmers i.e., Eisenstadt's study of the phenomenology of sequential code debugging i.e., how debugging is performed by real world programmers [44]. Eisenstadt elicited debugging anecdotes, descriptions of bugs and of bug-fixing strategies from developers of academic and commercial software. His analysis focused on the reasons bugs are difficult to find; how the bugs were found; and what were the causes of the bugs. He identified four[1] categories of reasons that bugs are hard to find (in order of frequency of occurrence):

Cause/effect chasm. Often the manifestation of the bug is far removed in space and/or time from its cause. It may not be easy to find the root of the problem when the evidence of the bug cannot be related to the program source code. Among bugs classified in this category are timing and synchronisation problems which can be intermittent, inconsistent or infrequent. Too many degrees of freedom exacerbate the difficulty of the task e.g., when a piece of software works perfectly in one environment, yet fails to work in another. If many environment parameters have changed then there are too many degrees of freedom to conduct controlled experiments especially under time or resource constraints.

Inapplicable tools. The bug disappears when the tool is switched on (probe effect). The program may have to run for too long in order to replicate the mistake. Tools may not be usable for various reasons, for example, memory constraints.

Faulty assumption/model. Possession of a conceptual model that explains the operation of a system software or hardware component reduces significantly

[1]Actually, Eisenstadt identified five categories but for the purposes of this study that fifth category has been merged with another one.

the search space for the cause of a bug; equally it can exacerbate the difficulties if it is faulty. Examples are faulty assumptions about how stacks grow or what is the correct behaviour of a function. Eisenstadt asked programmers to report on extremely difficult to find bugs. According to the answers to his third question – what was the root cause of the bug – most of the problems were related to memory allocation, or to conceptual errors.

Spaghetti code Eisenstadt found that there was a 100% correlation between complaints that a code was too messy to debug and that it was written by another person.

Eisenstadt recognised four main ways in which bugs are found. These are (again in order of frequency of occurrence):

Gather data. This category encompasses experimentation techniques such as use of printfs and inserting breakpoints in the program execution.

Controlled experiments. Controlled experiments are conducted which attempt to investigate what the cause of a bug may be. Controlled experiments begin with a hypothesis which is based on a conceptual model of a system component (software or hardware). If this model is good it is expected that the programmer will find quickly the cause of the bug and eliminate it. If the model is wrong then more experimentation is needed to find the bug and the model gets updated or extended.

Speculation. This class includes cases where programmers speculate about what has caused a bug. A number of techniques were reported for example, inspection, meditation, simulation, articulation and others.

Expert help. Alternatively, help is sought when a problem exceeds the expertise of the programmers.

Eisenstadt's categories were used by this work as the basis of a framework for the analysis of tuning which was subsequently expanded in the light of the investigation to reflect the differences between debugging and tuning. Whereas Eisenstadt's research was based on responses received through electronic mail and news bulletin boards, the research described in this thesis employed a large number of methodologies such as interviews, questionnaires, in situ observations and others[2]. Investigating the difficulties in tuning and how tuners discover and eliminate the performance bottlenecks in their programs is the theme of the next chapter of this thesis.

Whereas Eisenstadt focused on the cognitive issues involved in debugging, a consideration for this work has been to investigate parallel program development and tuning as a repetitive and collaborative activity. Once again, only a few studies of sequential programming have investigated the issues involved in team programming. For example, Flor examined cooperation in a small programming team consisting of two people [48]. It was shown how goals and plans were shared and how people collaborated through sharing artifacts which were external representations of the problem in hand. That investigation focused on a very small group

[2]Curtis has suggested that empirical studies of programming will need new techniques for gathering data such as videotaping programming in real world contexts [35][3]. Soloway focused on the need for alternative research methodologies in order to study *programming in the large*, and questioned the effectiveness of controlled experimental studies for studying large projects [155]. For example, the diary of an experienced programmer was studied where the programmer recorded argumentation about the program design and some results of test runs [125]. At the same time, system designers started facing the need to incorporate the work setting's social and organisational knowledge into the systems design [59]. Part of the motivation for this emphasis on the socio-technical can be attributed to an increased interest in usability issues. These include supporting co-operation in working environments and relating design to informal - as well as formal - work practices.

and could not obviously capture the ways that artifacts are shared within larger groups, among groups or even among organisations. Chapter Four extends Eisenstadt's framework to take into account the issues involved in performing tuning in a collaborative environment and the extended framework is used in the investigation of tuning in the large presented in Chapters Five and Six.

Chapter 4

Tuning in the small

The first section of this chapter investigates the issues which contribute to the difficulty and complexity of parallel program performance tuning. Fifty two tuners participated in this study. Twenty six tuners responded to questionnaires posted to relevant news bulletin boards and made available through the World Wide Web. In addition, and to enable issues raised by questionnaire respondents to be explored in depth, twenty six tuners working for a number of UK and European organisations and institutions were interviewed. In total, eighteen respondents worked for university establishments and thirty four for companies. A profile of the respondents is shown in Figure 4–1.

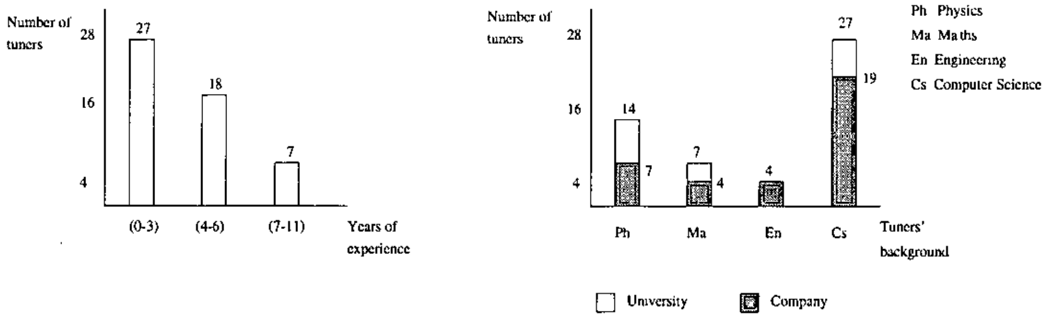


Figure 4–1: Profile of respondents

Their answers, summarised in the first section, drew attention to the need to extend Eisenstadt's framework to cater for the new issues introduced by parallel program performance. From an analysis of the experiences reported, an adapted framework is proposed and its validity examined.

Tuners talked also about the shortcomings of the tools they used, and gave suggestions on how these tools can be improved or what functionalities they would like their ideal tool to provide. The sixth section is concerned with issues such as access to tuning training and documentation. The final section discusses the implications that the results of this study have for tool design.

4.1 Difficulties of tuning

Fifty two tuners were asked to nominate the most difficult aspects of tuning. Their answers were analysed and classified in a number of categories which are presented in Table 4–1 in order of frequency of appearance[1]. Some answers are reproduced here by way of illustration; a more complete record can be found in Appendix A (e.g. [Ae41], [Ae43], [Ae44], [Ae45], [Ae46], [Ae47], [Ae50], [Ae51], [Ae52], [Ae55] and [Ae54][2]).

Finding and fixing. All entries in this class emphasised that although it is relatively easy to spot the symptoms of poor performance, it is very difficult to find the cause of the performance problem and fix it. Some answers

[1]Eighteen, fourteen, nine, two and one respondents mentioned one, two, three, four and six answers respectively. The total number of answers are eighty six. Some responses were included in two categories, hence the numbers 10.5 and 19.5.

[2]Ae denotes that the excerpt can be found in appendix A.

Category	Occurrences
Finding and fixing	19.5
Keep track of changes	17
Know specifics of machines	13
Interrelated parameters	13
Inadequate tools	10.5
Somebody else's code	5
Change	4
Time constraints	4

Table 4–1: Sources of difficulties in tuning

indicated that part of the difficulty is attributed to misleading information given by tuning tools.

Keeping track of the changes. The tuners whose responses are included in this category were all concerned with how to organise controlled tuning experiments and compare the different program runs.

Specifics of (often different) machines. Having to know the specifics of the underlying machine and system software is not something that the average tuner finds easy to do. Not surprisingly therefore, porting the program to another machine with a different architecture is regarded extremely difficult as well.

Interrelated parameters. One of the problems in tuning is that the parameters which have an impact on the program performance are often many and interrelated. Thus, the optimum value for one parameter may not belong to the set of optimum values for all the parameters that affect performance. The lack of conceptual models of parameter inter-relationships results into tuners employing trial and error in order to specify a model of interactions between the parameters. These interactions are then verified through a number of edit-compile-run cycles which limit the implementation of program changes. As it is not easy to try all the possible combinations, tuners try to predict what changes will work together. Responses of the form "lack of understanding of the basic issues" were included in this category, as they imply a difficulty in understanding the interactions among the various performance determining parameters.

Inadequate tools. Answers in this category included cases where it was difficult to interpret correctly the information provided by the performance analysis tool or tuning tools were simply unavailable.

Other people's code. Programs written by other people are extremely difficult to understand and optimise. Answers indicated that codes with very complex data structures are extremely difficult to parallelise efficiently.

Change. Parallel computing is a particularly dynamic area which changes constantly. Compilers that have bugs or keep changing, hardware which is not configured fully, working with prototype parallel machines or newly implemented languages exacerbate the difficulty of the task.

Time constraints. Two informants were concerned with tuning taking place under conditions of time pressure. Tuning can prolong parallel programming because programmers have to develop code which is not only functionally correct but performs well at the same time. Progress is limited by the edit-compile-run cycle which can take a considerable amount of time. Time constraints can be imposed by project deadlines and by the available time tuners have on parallel machines.

At this point, Eisenstadt's classes can be revisited to examine their relevance to parallel program performance tuning. The responses show that parallel programming introduces issues which are not present in sequential program development so new classes emerge and existing ones need to be augmented.

- "Cause/effect chasm"

  In tuning, the "cause/effect chasm" is instantiated in the gap between *finding* where performance is lost and knowing the reason for the performance degradation in order to *fix* it (*finding and fixing*). It is very difficult for tuners to try to find in large volumes of performance data and tuning information where performance is degraded in the program and establish the reasons for the performance bottlenecks. Porting the program to a new parallel machine is very common and this introduces many more degrees of freedom in the search for bottlenecks. Due to the complexity of programming models, there are many *interrelated parameters* whose model of interaction has to be investigated before any assumptions can be made about the cause of a performance problem.

- "Faulty assumption/model"

  Eisenstadt in his "faulty assumption/model" class included cases where a bug was exceptionally difficult to find because of a wrong assumption the programmer had made about a model of the system software or the machine operation. In tuning, a lot of knowledge about the *specifics of the underlying machine* is required in most of the cases and not only in particular difficult ones. The challenge of tuning is not unlike that of sequential programming at assembly level, where the programmer has to know how a large number of low level entities operate. In addition, whilst there is only one model of sequential programming, there are many models of parallel programming. Models are more complex and often involve a large number of *interrelated parameters*. Porting the program to different machines increases the likelihood that details of models may get confused or forgotten.

- "Inadequate tools"

  The impact that "inadequate tools" have on the development process is larger in parallel than in sequential programming, especially when the immaturity of system software is considered as well (see next category). Tuning tools are often unavailable. Enabling the tuning tool alters the behaviour of the executing program. Tools may mislead tuners by the way in which performance information is presented.

- "Change"

  This is a new category as tuning is often performed when hardware and system software are still unstable and immature. It is almost impossible to overcome hardware inefficiencies. Programming environments are not mature enough and tuning of programs, written in a new language without advanced compiler and run-time support, is common. Tuning is made harder by the fact that optimising compilers are not sophisticated enough and often affect the correctness of the program.

Eisenstadt focused on the cognitive aspects of debugging and dealt only with the difficulty of bug finding without examining debugging in the context of requirements and work practices found in large project development teams. Eisenstadt's findings refer to exceptional bugs which were very difficult to find. His respondents did not talk about cases where debugging was routinely performed during program development. Thus, the picture of debugging that emerges from Eisenstadt's study is that of a task aiming at eliminating a bug in one instance of time. Hence, the categories described above are pertinent to tuning in the small and can be used to investigate how the unit tuning cycle is performed and how it can be affected by parallel software and hardware.

On the other hand, a number of issues stem from the fact that tuning is repetitive and can occur in the context of collaboration with others. The two categories described below are relevant to tuning in the large as they can be used to examine tuning in a wider context than that of tuning in the small.

- "Repetitive nature"

  Tuners must consider many reasons for performance loss and many locations in the program where bottlenecks may occur. In cases where tuners attempt to establish a model of the interaction of the many *interrelated parameters*, experimentation is prolonged and results in tuners having to manage the tuning process. Whereas debugging is successful when bugs are found, performance tuning does not have a discrete result. Performance can be improved to different degrees. Tuning can be repeated until specific performance goals are met. A tuning project may take considerable time, during which it may be stopped and resumed numerous times. Tuning may be performed under conditions of time pressure due to deadlines or to the finite time available on a parallel machine. *Time constraints* have an impact on the way tuning is performed and are an additional source of difficulty. Tuning may involve many changes to the program and tuners have to check for correctness after every substantial change. *Keeping track of changes* is difficult especially in the context of large development projects. Moreover, when programs are ported performance issues are reopened.

- "Somebody else's code"

  Eisenstadt's category "spaghetti code" has been renamed here *somebody else's code* because tuning code for others is a very common pattern in parallel programming. There are many implications for the way tuning is performed when the code is written by someone else or it is written in the context of multi-people development projects. Lack of documentation makes the task harder.

Category	Occurrences
Cause/effect chasm	25.3
Repetitive nature	23.8
Faulty assumption/model	17.3
Inadequate tools	10.5
Other people's code	5
Change	4

Table 4–2: Revised framework

Under the new extended framework the category *finding and fixing* corresponds to the category "cause/effect chasm"; the class *specifics of machines* corresponds to the "faulty assumption/model" category of the analysis framework. The *keeping track* class is relevant to the "repetitive nature" category. The answers in the category *interrelated parameters* were shared among the above three classes. The *time constraints* answers were included in the "repetitive nature" class. It is interesting to see in Table 4–2 that the "repetitive nature" category whereas it is not mentioned in Eisenstadt's study, is ranked as the second most difficult aspect of tuning.

The following sections present evidence of the kind of problems tuners face in four of the above categories. Issues in relation to the *repetitive nature* and *somebody else's code* categories (tuning in the large) are examined in the next two chapters.

4.2 Cause/effect chasm

As programmers get more exposed to parallel computing their experience in performance tuning is enriched in the form of heuristics or cliches. These heuristics can vary from being general e.g., "remote memory accesses are more expensive than local memory accesses, so they should be avoided" to being specific to the features of the underlying compiler and hardware e.g., "given that the size of the T3D cache is small, loop unrolling could make cache accesses more effective". When such rules are applied the programmer expects the performance of the program to improve. There are cases, however, where other factors contribute in such a way that the impact of applying a well known rule is not positive. Although such unexpected outcomes can be explained[3], the explanation may often be far away from the context of the rule itself. Thus, there are many degrees of freedom which determine the effect of a performance optimisation and the impact of a performance improving design decision cannot generally be known *a priori* without experimentation. Experimentation, however, may produce huge amounts of performance data and this makes even identifying the manifestation of poor performance (effect) a difficult task.

For example, in [Ae25] the tuner knew that next neighbour communications in the array of processing elements of the Maspar data parallel machine were almost as fast as local memory access and that performance could improve if they were preferred instead of communications through the global router. When this type of communication was used the tuner discovered that, in fact, the performance of the program worsened:

> [Ae25] ... because the processors are so small, this neighbour communication is nearly as fast as a local memory access. But, on the other hand, if you go too far by that neighbour communication, you worsen the performance and also there's a chance that too many processors try to send a message through the same route[4]... Informant 42

[3]Actually, studies of programming have shown that while the knowledge of the less experienced programmers is fragmented consisting of heuristics and cliches existing independently of one another, more experienced programmers integrate these heuristics in the context of global conceptual models.

[4]Novice programmers may not know how routing mechanisms can actually affect the communications of their programs.

Sometimes, tuners have to deduce the value of a particular performance metric from other ones that are available within the tuning tool. In [Ae108], the tuner was given an estimate of the load imbalance problem of the program. He had, however, to determine the degree of load imbalance as that estimate was made on a different machine. The information that the tuner had had about the performance problem compensated for the poor support the tuning tool provided for load balancing investigation. The tuner tried to verify the load imbalance problem indirectly by looking at a second level metric such as the time spent waiting for messages.

In the following case, the manifestation of the problem was that some CM Fortran statements took too much time to execute. Informant 55 was an experienced tuner, and knew that statements in a CM Fortran program are split into blocks, which are communicated from the front end of the machine to the array of processing elements. The communication of each block of code has a startup cost related to it so it is better if program statements are ordered in a way that as many instructions as possible are placed within a block. Informant 55 looked at assembly code to find out how best to re-order the statements in the program. A novice parallel programmer would not know what caused specific statements to be very expensive. For less experienced tuners, who may not be able to work at such a low level, the informant suggested that they should try random combinations of the order of the source code statements:

> [e16] The fact is, that I have a lot of experience with the lower level assembly language and I look at that and I can see it is splitting the code into many blocks which is inefficient. The most efficient thing is to have everything in a single block. If you write the program so that it is split into many blocks, the execution time is increasing, because there is a startup time for each block. You don't have to go right down to the assembly code; you can use a profiling tool and if you see three statements having the same number next to each other, then you add this number three times and this is the completion time for this block which has been averaged. Whereas if you see statements next to each other with a different number, that means that they belong to different blocks. So maybe, you can try different combinations of the order of the three statements. It may be the second one that stops the first and the third to be put into one statement. So maybe, you could re-order them. *Informant 55*

4.3 Faulty assumption/model

There are a large variety of high performance parallel computer architectures. One way in which they can be classified is by memory access:

- **Shared memory.** Each processor has direct access to memory by use of a shared bus or network.
- **Virtual shared memory.** Each processor has local memory but may access remote memory by use of global addresses. This access is accomplished by low-latency messages sent over an interconnection network.
- **Distributed memory.** Each processor has access to local memory via a local address space. The processors are connected by some network. Access to remote memories is only available via some message-passing system.

Another distinction can be made between Multiple Instruction Multiple Data (MIMD) and Single Instruction Multiple Data (SIMD) hardware. In the former there are typically n complex processors executing up to n distinct instruction streams independently. An SIMD machine has each (simple) processor operating synchronously and obeying instructions from a single, central controller. For a machine to be used efficiently, the processing of data has to be distributed onto the machine such that each processor can be kept busy on part of the problem. It is also desirable that as little data as possible is moved between processors since communication tends to be very time consuming compared to computation. It is important that any programming environment gives adequate control over the distribution and communication of data. Two programming paradigms have emerged and gained wide acceptance by the user community:

Message passing. In this paradigm, individual programs written in a serial programming language are executed on each processor having access to its own local memory. A number of message passing environments exist. To transfer data between nodes the programmer can use send and receive routines provided by the system library. Often more complex collective operations are also available.

Data parallel. In this paradigm, a single program controls the distribution of, and operations on, data distributed across all processors. A data parallel language will typically support array operations and allow whole arrays to be used in expressions. The compiler is responsible for producing code to distribute the array elements on the available processors. Each processor is "responsible" for the subset of the array elements which are stored in its local memory.

The architecture of the target machine, together with the available compilers determine the options the programmer has for writing the parallel program. Once the programming model is selected, the programmer then has to consider how data and control can be best distributed in the program. The programmer has to be aware of low level details which affect the performance of the program. Some aspects of some programming models can be too complex for the inexperienced programmer to use (e.g. [Ae101]). The need to know low level details about the operation of the machine can be a problem ([Ae12], [Ae20]) when different kinds of optimisations are needed each time:

> [Ae12] ...and the latest architectures are RISC which I am not familiar with, for example, with vector architectures you know what to do, you know with the inner loops and things. In terms of cache use, I am little bit lost. *Informant 47*

> [e700][5] I started using Occam and I got so much used in using it that I didn't want to start using a different language. For my thesis work I kept on using Occam all the time.
>
> *Informant 57*

The tuners in [Ae20] knew how to apply the standard optimisation techniques for the kind of architecture they were considering, but these did not bring the expected results. The code was developed twenty years ago for vector supercomputers and drastic changes were needed if more performance gains were to be achieved [Ae38].

The problem can be exacerbated when changing a machine means that the performance problem is re-defined from the beginning (e.g. [Ae41a]). Inexperienced programmers usually start from a parallelisation scheme which is easy to implement correctly:

> [Ae21] The strategy for parallelisation we adopted wasn't the only one. But the other options we had would make the code so complicated that it would be unmaintainable, so we had to put up with less performance but with a more maintainable code. Parallel programming is difficult anyway. So, I tried to implement the simplest approach I could think of, to section data in a regular manner. I didn't actually at that time envisage that things could be so computationally heavy. In respect to the actual core code. So many processors were sitting idle. Because experience with parallel programming is so little, most of the people have a sequential background. And so if you get a non regular decomposition that increases the design so much that it is not worth it. And that was were most of the mistakes with the TLM code were made. For every design decision, we asked what is easier to switch to this path or to the other path. We took always the easiest option. The overhead of going another way was just too much. Any problems with the code, the project leader was not there, the learning curve to understand the code was too large. *Informant 47*

Novice programmers who do not have a conceptual model of how parameters which affect performance relate to each other may make wrong performance determining design decisions whose correction may require substantial re-organisation of the

[5] *e* denotes an excerpt that is not continued in the appendix.

source code. In [Ae7], the tuners' inability to predict where the performance of the program would be degraded meant that the data decomposition scheme and the communication patterns were altered in order to increase the efficiency of the code. Performance mistakes may escape even the most experienced tuners during the design and the implementation phases of the parallel program. These mistakes, however, are simple and are not likely to require major changes of the control structure of the code as experienced tuners have a correct model of the performance determining factors:

> [e23] In my experience you know, you always make a stupid mistake, while designing your parallel code, which will result in performance degradation, so tuning simply means to find it and remove it. *Informant 42*

Hoc reported that novice programmers find it difficult to grasp the implications of the machine model for program design [69]. In parallel programming, however, even the most expert tuners may fail to appreciate the implications of machine specific models for performance. In the following case, not understanding the degree to which performance depended on the machine load resulted in tuners rejecting this as a reason for performance degradation and experimenting instead with all the other possible factors:

> [e42] On the KSR we had a really hard time because they had a multi-tasking, multi-user operating system and I knew it would change the results, but I didn't realise it would be that bad, so we were for a week or something thinking that the code was wrong; it was actually that there were people on the machine and the speed up would totally change. Yes, we made some measurements using printf and timing routines and then we started using pmon, it is some monitoring stuff from the machine, which is not bad, and we ruled out everything else and then we said that must be it, and we tried measuring without people on the machine and it was fine. *Informant 42*

4.4 Inadequate tools

To investigate in what way inadequate tools can affect program development and tuning and to validate the corresponding class of the analysis framework, fifty two informants were asked whether they had ever used any performance analysis tool with their parallel code. Eight of them had ported their code to a second machine, four to a third machine and one to a fourth machine. This increased the answers to seventy one. Table 4–3 illustrates the answers received; despite the large number of tuning tools reported in the literature (Chapter Two), in thirty three ($\simeq$ 46.4%) out of seventy one cases, there was no tool available. In eight cases ($\simeq$ 11.2%), tuners built their own performance visualisation tool. In the remaining thirty cases, informants had used an externally developed performance tool which accounted for the $\simeq$ 42.2%.

Tuning tool?	Occurrences
Yes	30
No	33
Own tool	8

Table 4–3: Tuning tool use

The performance tuning support that the machine vendor provided in [Ae21] was a number of flashing lights on the parallel machine indicating when the processors were busy. In the absence of tools, tuners may develop their own which are sometimes too closely oriented to the problem in hand. This means that the same tools may be inapplicable to other situations:

> [e35] Yes, I use a tool I developed myself. One which is tuned very closely to the problem I am solving and the approach I have taken to solve it.
>
> *Informant 16*

Lack of an adequate tuning tool means that parallel programmers may be unable to gather performance data [Ae53]:

> [e14] Taking a program written in CS-tools from the Transputer or i860 box to a cluster of workstations led to very poor performance. The performance degradation was much worse that it would have been expected due to the slower communication medium and lower bandwidth of that medium. It wasn't until I tried using PVM that I could convince people that it wasn't my program that was at fault, but Meiko's workstation implementation instead. The two worst aspects of the whole trial were the inability to see what was going on in the parallel message passing library itself and the ability to get a good measure of what my program was doing on each processor without having to write my own tools. *Informant 16*

Users want performance analysis tools to be flexible in the amount of information they show: they dislike tools which do not have any mechanisms for data filtering (i.e. mechanisms for reducing the amount and the level of detail of performance data). Also, they would like to be able to set themselves the level of filtering ([Ae74], [Ae75], [Ae76]), instead of being presented with information which is either too condensed or not summarisable at all [Ae90], [Ae89], [Ae100]:

> [Ae85] ...`Apprentice` always shows this global view, it shows statistics. You have the total time spent in one subroutine, but you cannot see how much time was spent in this subroutine over the processors. So you can't have a histogram of this subroutine over all the processors to see where the time is spent ... *Informant 51*

Tools may be successful in assisting the tuner to spot the performance problem, but this meets only a minimal requirement, as users need assistance with discovering the root of the problem and eliminating it i.e., bridging the cause/effect chasm [Ae78]. The gap that tools are called upon to bridge can be attributed to users' non-computer science background and to lack of knowledge regarding the specifics of the hardware and the system software the application program uses (faulty assumption/model). Moreover, tools are designed by computer scientists who cannot imagine what tool users know or do not know. The few tools that provide advice in the form of "observations" may sometimes convey this information in inefficient, incomprehensible or even misleading ways [Ae102], [Ae84]:

> [e77] ...`Apprentice` is not very good suggesting how to improve performance.
>
> *Informant 41*

> [e111] The problem with `apprentice` is that if you have a particular routine it will tell you how much time you waste on memory access. Now, the better the routine, the more time you waste in memory access. Well, if the routine is very inefficient in its numerical operations, memory access time loss gets hidden by the inefficiency of the routine. If you rewrite the routine to optimise it, so that if you optimise the use of the CPU, the bottleneck shifts to memory access, which will result in apprentice telling you "your routine is wasting 60% of its time in memory accesses, now it is wasting 80% of its time". The total time of the routine has gone down because, obviously, when memory becomes bottleneck, you are running at optimal speed. So, `apprentice` can be misleading, if you use it for one routine, you might get the wrong idea. *Informant 44*

In [Ae84], Informant 45 (a computer scientist) admitted that he was helped by a course on performance optimisation to really understand the information provided by the tool. The course was also valuable because it focused on the attributes of the parallel machine that really make a program susceptible to memory bottlenecks. This case demonstrates that even an averagely experienced computer scientist may have difficulties in understanding the information provided by the tuning tool.

Evidence is lacking regarding the usefulness or the usability of tuning tools [Ae82], [Ae79]:

> [e36] I find it difficult with the CM to match the timings, because if you have a large piece of code running on the CM and a large piece of code running on the front end, you cannot actually profile them together. You have to profile a piece of code on the front end with a piece of code on the back end which is empty. And reversely. And you have to get these times together and see what they mean. Another difficult thing about the profiling is accepting that you have actually done wrong. Sometimes it is not easy to see, when you have the information, to understand what is wrong. *Informant 64*

Tool documentation may cover issues such as the user interface, but does not often address how the tool can be used to tune real programs. The study of an email-based user support service (described in Chapter Six) has shown that a number of users embark on parallel program tuning without having any strategy for it. Tools are sometimes unreliable and industrial users express doubts about tools which are developed in the context of academic projects, as they may not meet the requirements of large projects. Often, tuners find out too late that a tuning tool cannot actually work with traces from their machine [Ae83], [Ae76]. Once confidence in a tool is destroyed, it will not be used again:

> [Ae88] ...I wanted to produce a profile of the code, a very simple thing, you know I am not asking for a lot of things. Just a profile on where the time is spent, I had one vector node so I used `prof`, `grpof` and the ...`profiler`. All failed, for three different reasons. It is great to have a utility, but it must be robust. Because if it fails for whatever reason, it is of no use to you. And the users will never use it again. And profilers are the most basic tools. You couldn't even consider them as tools necessarily. *Informant 50*

Performance analysis tools may have bugs themselves, which may be easy to detect if the manifestation of a bug is obvious ([Ae91]), so the impact on the programmer's work is minimal. In more severe cases, however, the bug could bring the system down ([Ae99]) or even mislead the tuner by presenting an inaccurate picture of program performance.

Instrumentation perturbs program execution and this affects how and when tools are used [Ae92], [Ae97]. Tools are used to verify that the program works as the programmer thinks it should work [Ae107].

> [e105] We used `Paragraph` mostly later. When we had something that worked in parallel, when the message passing worked, we looked to really to see if we communicated in a way we really wanted to do. *Informant 51*

A problem mostly met in early examples of tools was that performance metrics were isolated from the program source code, thus contributing to the cause/effect chasm [Ae104], [Ae103]:

> [Ae104] ...So, my tool gave hundreds of different performance metrics about the processors but we would not see where these metrics applied.
>
> *Informant 59*

The performance of some programs is not reflected in the performance metrics that a tool may provide. Application specific visualisations are sometimes necessary:

> [Ae106] ...While `Gecko` would say that if everything is busy, everything is red, so everything is ok. But this doesn't apply in the case of this simulator[6] because there are many cases that the simulator can be working really hard. Everything working red hard. But the simulation doesn't progress. It progresses in very tiny steps. It is like busy waiting times. *Informant 43*

The repetitive nature of the tuning task has been ignored and existing tools do not support tuning experimentation. Storing each version of the application program along with tuning products in different directories is perhaps the most common practice for differentiating among program versions. Programmers typically copy the code to a new directory when a substantial change in the code's modules is made which is expected to bring substantial performance gain. Fast assessments of performance using a tuning tool are hindered when performance information is stored in trace files whose name depends on the executable program's name or is always the same:

> [e500] In principle, if you had actually two different programs from two different runs, you could actually keep the source code and the tif files (compile-time information files) and the rif file (run-time information file) go to another directory, compile the other code again and keep the other rif and tif files and ...you could start `apprentice` more than once. The problem with `apprentice` is that you can not actually give `apprentice` a different rif file. The problem is that you need your old code and your old tif files otherwise `apprentice` will not know what it is doing[7].
>
> *Informant 44*

[6]The simulator was the application program.

[7]A compiler information file is generated for each source file. It is named after the name of the source file and has the suffix ".T". Thus, a second compilation in the same directory would overwrite all the information from the previous compilation.

4.4.1 Tool requirements

Fifty two tuners were asked to specify their requirements with respect to either the tool they use currently or their ideal tool. The requirements were edited, summarised and are presented in Figures 4–2 and 4–3[8] (selected requirements are also presented in Appendix A as [Ae66], [Ae67], [Ae68], [Ae69], [Ae70], [Ae71], [Ae72] and [Ae73]). Thirty nine tuners answered this question, and there were in total 246 requirements recorded. From those, 133 ($\simeq$ 54%) were concerned with specific features which tuners would want their tools to have; 75 ($\simeq$ 30%) were concerned with abstractions that would reduce the information displayed; 15 ($\simeq$ 6%) were requirements for tuning management mechanisms; 13 ($\simeq$ 5%) and 10 ($\simeq$ 4%) were requirements for friendlier and more reliable tools, respectively.

Due to the different programming models that respondents were using it is not easy to comment on the specific tool features required. On the other hand the requirement for meaningful abstractions is independent of the programming model. In summary, users do not want to be swamped with information ([Ae94]) but they want to have the means to select only the relevant performance data and be able to eliminate the rest. The requirements revealed a number of ways in which this can be achieved and reflected the way tuners go about the task. For example, the selection mechanisms (summarised in Figures 4-2 and 4-3 under the heading abstractions) could support: the creation of user defined events that are aggregates of other lower level events; the selection and display of a number of metrics that are of interest; the display of information when a performance metric is below or above a threshold; the display of information only for processes that are representatives or outliers of a class of processes; and others (Figures 4-2 and 4-3). Tools should

[8]The numbers close to a requirement denote the participants who expressed that requirement.

be easy to use and should be able to handle the requirements of large development projects by handling large trace files, and by facilitating tuning management.

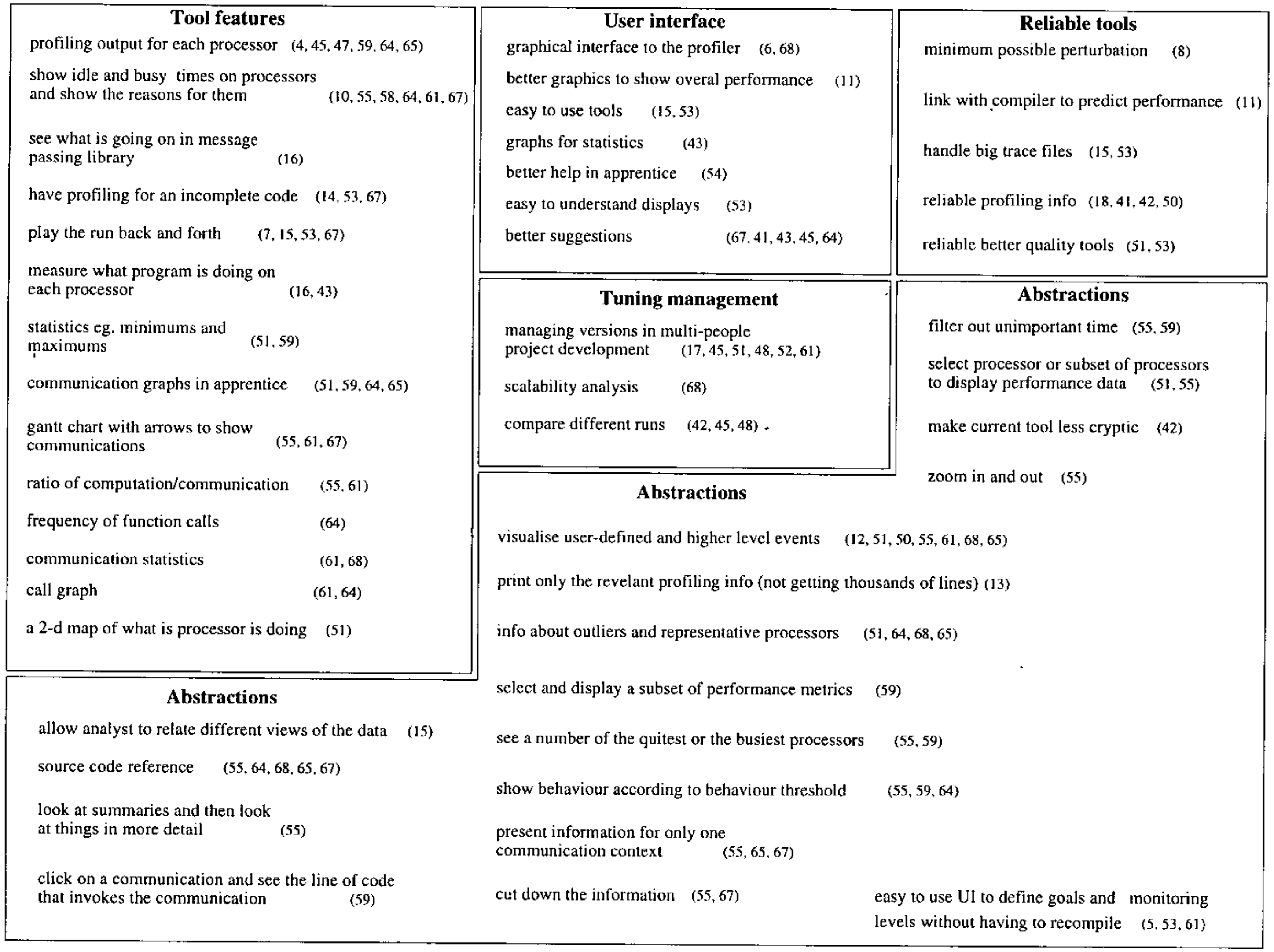
4.4.2 Case study I: tool evaluation

Informal tool evaluation sessions with a small number of tuners revealed a number of usability problems.

Informant 44 was observed while using `MPP Apprentice`. The user was almost at the end of optimising a large fluid dynamics code. He had tuned all of the most critical routines in his program and he attempted to fine-tune some of the less critical ones. His program contained a large number of routines which took little time. He decided to deal with one of the most expensive routines of this list. This was the MPI_Barrier function. He invoked the display which showed where MPI_Barrier was called from: a routine called MPItime. The tuner's immediate reaction was to try to click on the MPItime function hoping that the navigation display (Figure 2-5) of `apprentice` would find the specific function so that he could have a look at the code. Instead, the user had to go back to the navigation display of the tool and scroll the list of routines until he found the routine MPItime which he then expanded to examine [Ae112].

`Paragraph+` was evaluated by Informants 53 and 50. The informal evaluation revealed a number of bugs and inefficiencies in the design of the tool. Amongst other things, the tuners suggested how certain features could be improved and stated a number of requirements. The discussion has been modified slightly to be more intelligible, and summarised for brevity's shake. Modifications include the separation of the transcribed conversation into parts each addressing a different issue[9].

[9]The corresponding excerpts can be found in Appendix A.



Tool features

profiling output for each processor (4, 45, 47, 59, 64, 65)

show idle and busy times on processors and show the reasons for them (10, 55, 58, 64, 61, 67)

see what is going on in message passing library (16)

have profiling for an incomplete code (14, 53, 67)

play the run back and forth (7, 15, 53, 67)

measure what program is doing on each processor (16, 43)

statistics eg. minimums and maximums (51, 59)

communication graphs in apprentice (51, 59, 64, 65)

gantt chart with arrows to show communications (55, 61, 67)

ratio of computation/communication (55, 61)

frequency of function calls (64)

communication statistics (61, 68)

call graph (61, 64)

a 2-d map of what is processor is doing (51)

User interface

graphical interface to the profiler (6, 68)

better graphics to show overal performance (11)

easy to use tools (15, 53)

graphs for statistics (43)

better help in apprentice (54)

easy to understand displays (53)

better suggestions (67, 41, 43, 45, 64)

Reliable tools

minimum possible perturbation (8)

link with compiler to predict performance (11)

handle big trace files (15, 53)

reliable profiling info (18, 41, 42, 50)

reliable better quality tools (51, 53)

Tuning management

managing versions in multi-people project development (17, 45, 51, 48, 52, 61)

scalability analysis (68)

compare different runs (42, 45, 48)

Abstractions

filter out unimportant time (55, 59)

select processor or subset of processors to display performance data (51, 55)

make current tool less cryptic (42)

zoom in and out (55)

Abstractions

visualise user-defined and higher level events (12, 51, 50, 55, 61, 68, 65)

print only the revelant profiling info (not getting thousands of lines) (13)

info about outliers and representative processors (51, 64, 68, 65)

select and display a subset of performance metrics (59)

see a number of the quitest or the busiest processors (55, 59)

show behaviour according to behaviour threshold (55, 59, 64)

present information for only one communication context (55, 65, 67)

cut down the information (55, 67)

easy to use UI to define goals and monitoring levels without having to recompile (5, 53, 61)

Abstractions

allow analyst to relate different views of the data (15)

source code reference (55, 64, 68, 65, 67)

look at summaries and then look at things in more detail (55)

click on a communication and see the line of code that invokes the communication (59)

Figure 4–2: Tool requirements

Tool features

Utilisation display: how much dead time varies over time (63, 56, 60, 62)
Which processes finished last (56)

Animation of communications (63, 56, 60)
and structure of sapgroups (62)

Structure of communications (63, 56)

Gantt chart with green for computation and red for communication (63)

Zoom in visualisation displays (63)

Measure how much you have gained by doing a non-blocking call (63, 56)

Profiling (64, 60)
see time spent in a phase (56, 62)
frequency of functions calls (60)

Replay of events (56)
Colour processes in terms of having executed a phase (56)

Computation to communication ratio (56)

Call graph (56)

How long you have been blocked in a message (60, 62)

Encode each send with a filename and line number (63, 64, 56, 60, 62)
Click on a send in the source code and have the display show the first instance of this event (63)

Measure things between the arrival and reception of messages (63, 56)

Frequency of lengths of messages (63)

Show how many messages are outstanding to be received (63, 56, 62)

Message type to be distinguished (control and data) (64)

Graphs of lengths and sizes and completion times for messages (56, 62)

Contents of messages (56)

After execution static map of messages (56)

Abstractions

Relation of displays: pick a point on one display and see what else there is on other displays (63, 64)

See behaviour of PUL and CHIMP groups separately (63)

Visualise perrformance data separately for different sapgroups (63, 64, 62)
Statistics only for the workers of the task farm (63, 64)
Consider a group as one entity (60)
Communications to a specific sapgroup (63, 64)
See communications within a specific group and outside a group (56)
Phases for complex PUL and CHIMP events and user defined events (63, 64, 56, 62)
User should be able to combine events together to define states (64)
Visualise only under certain conditions (threshold) (63, 64, 62)

Filter in and out processes from the displays (63, 62)

Visualise processes and processors (63, 62)

Reduce scope of monitoring (63)

Selection mechanisms (63)

Visualise data for representatives and outliers of sapgroups (64, 56, 62)

Zoom in a group and see what each process is doing (64)

Ignore data of no interest 64)

Average data across processors (64)

See information at various levels of abstraction (60)

Compare a processor with the rest of the group (62)

Tuning management

Relate performance results to changes (56)

Compare different runs (56, 60)
Keep statistics about the runs (56)
Scalability of pul utility (62)

Figure 4–3: Tool requirements expressed by participants in the design meetings of VISPAT (Chapter Seven)

Informant 53 complained that `Paragraph+` could not cope with the generation of large trace files, and that meant that he could run `Paragraph+` only with a scaled down version of the program solving a small problem. Even if a subset of processors was selected, for which `Paragraph+` could show performance data, a full trace file would be generated [Ae114]. Selective monitoring would alleviate this problem. Informant 53 stated that sometimes he was not interested in the full code, but he would like to isolate and work on a small section of it. At the time of the evaluation, `Paragraph+` did not support selective monitoring apart from the ability to instrument user-defined events [Ae117].

Next, informant 53 focused his attention on a load balancing display which in his opinion did not depict information he would need in order to examine the load balance of the program [e115]. He went on to suggest a display that would really help with load balancing [Ae116]. On the other hand, the statistics display was considered redundant since it provided information the tuner had already known [Ae125].

> [e115] ...So this is a display with user defined events which are very strange. I have difficulty in understanding them, for example, this display will give you the first time on any processor that this user defined event was started and then gives you the last time when it was finished. Well, I understand what the display means but I have problems relating this to some information I would be after ...like did it take longer on processor one? Than it did on processor 28 or something? And you can't tell since this is a kind of strange summation of time. And maybe you interested, some times, in one instance of this event. *Informant 53*

A number of bugs were revealed when the tuner tried to "zoom in" a specific period of time [Ae123]. The way to zoom in was very complicated and required the tuner setting the values of a number of fields.

A series of excerpts ([Ae124], [Ae126], [Ae127], [Ae128], [Ae119], [Ae118], [Ae120], [Ae121], [Ae122]) demonstrate that a consequence of poor design was that tools were used mainly for demonstration purposes during presentations and for reassuring developers that programs ran according to how they expected them to run rather than for tuning itself. An issue raised was that tuning tools should be able to work with incomplete programs [Ae113].

4.5 Change

Discussions with informants revealed that tuning can be affected by inadequate system software. Tuners have to be aware of compiler changes which can make performance related design decisions obsolete after some period of time. For example, in [Be628], remnants of old code which were necessary under the old version of the compiler had to be erased from the source code files as the compiler changed in the middle of the tuning efforts. At the same time, the tuner knew that some of the performance improvements could be introduced only when the future version of the compiler would be installed. These changes were postponed until the new version of the compiler appeared.

System software may arrive in an inconsistent state when different software components get improved at different points in time as in the following case. The application program (ARPS) used CMAX a parallelising pre-processor which transforms Fortran 77 programs to CM Fortran programs:

> [Ae15] ...ARPS had been performing a nearest neighbour computation in one part of the code, which CMAX translated into an expression containing a number of EOSHIFTs (End Off Shift). Unfortunately, the latest version of CSHIFT (Circular Shift) in the CM Fortran run time library is currently more optimised than EOSHIFT. *Informant 17*

Insufficient support from compilers of new languages disrupts the code development process ([Ae86]) and may mean that tuning is prolonged in vain only to prove that the system software needs to be changed [Ae11], [Ae22] [Ae48]:

> [e10] ...the biggest effort took two weeks and wound up causing changes to the Sisal compiler. *Informant 20*

Tuning is concerned with minimising the cost of the most expensive parts of the program. Despite the processing speed of modern parallel machines, I/O is the source of the largest performance losses. When the execution time is dominated by I/O operations, which usually are very difficult to improve, any further tuning may be deferred:

> [e6] There are programs that take 45 minutes to read the data in and that changes your attitude to tuning. *Informant 64*

In the following case, time constraints and system failures meant that the tuner had to learn how to switch between different activities. Suspending the tuning effort under sudden conditions exacerbates the difficulty of the task:

> [e9] Yes, it made me learn to do some context switching. So one day I would work with the transputer and the other day it was down, so I had to do something else. If one make fails, the whole system fails. And when the transputer was working, the SUNs' network would go down and then you had a crash and you could loose everything and then debugging on the trasputers is really difficult. *Informant 43*

The problems that the tuners encountered in [Ae17] can be attributed to the lack of fully configured hardware, the lack of reliable profiling tools and the lack of a sophisticated enough compiler. The work described in [Ae17] tried to produce an optimised port of the IFS code for the Meiko CS-2 MPP system[10]. During that period, the CS-2 consisted of 8 scalar nodes and 1 vector node. The tuners had three months access to the CS-2 which was located in France. As part of a phased delivery program, the vector nodes on the CS-2 system were initially installed

[10]The Integrated Forecast System (IFS) code has been ported to a number of computer architectures in the past 2 years including Cray C90, Cray T3D, IBM SP1, and Meiko CS-2. The model is typical of state-of-the-art applications in that it is over 100,000 lines of Fortran 77, consists of 650 subroutines and consumes large amounts of memory and processing power to perform its calculations. As an example, a 10-day T213L31 resolution forecast takes about 2 hours on a 16 processor Cray C90.

without memory coherency support between the Sparc processor on-chip cache and the two mVP processors on the same vector node. Effectively, this required the Sparc processor on the vector node to run with its on-chip cache disabled if any part of an executing program was vectorised, with an expected impact on performance: during the first week, the team experienced an initial degradation of performance. The tuners concluded that until cache coherency hardware became available this issue alone was enough to deter users from using the vector node on the CS-2 for large applications. The team was able to run only a small version of the application so it was not possible to extrapolate what the performance of the full model would be on that machine. Due to the fact that the vectorising compiler was not very sophisticated, some of the loops had to be changed manually to be vectorisable. The lack of a profiling tool made the tuners profile the same application on another parallel machine in order to constrain the manual changes to the most expensive loops. Since the IFS code had to be maintained as portable as possible it was concluded that a more sophisticated compiler was needed in order to minimise the manual changes that had to be done in the large loops of the application.

4.6 Training in Tuning

While it is evident that tuning requires extensive knowledge about the different programming models and the specifics of the different parallel computers, few parallel program tuners have access to training and documentation. Thirty three tuners were asked whether they had attended any appropriate performance optimisation course or had had any access to tuning documentation i.e., manuals describing how to achieve performance optimisations specific to a programming environment (Table 4–4). Sixteen were working for an academic institution, seventeen for industry. Ten of them ($\simeq$ 33%) had some training in tuning and all but three were working for either a company that manufactured supercomputers or used them extensively. Only three ($\simeq 9\%$), on the other hand, had access to tuning documentation and all of them were working for companies as well.

question	Yes	No
Training in tuning	10	23
Access to manual	3	30

Table 4–4: Tuning training and access to documentation

Courses are given either by vendors [Ae31] – their courses are regarded as very successful [Ae33] – who sometimes work closely with their clients, or by experts at various High Performance Computing Centres [Ae32]:

> [e34] Well, all the projects we work on, include very tight links with the computer companies since usually they pay us to have our software working on their machines. Hence, I spent 11 weeks working together with a Maspar guy (I mean in the same office both in Sunny Valley and France) and he explained to me almost everything necessary for me about tuning on the MPP systems. With TMC[11], it is the same, I was sent for at least 6 weeks in Cambridge and some TMC guys came for another 5 weeks to MECALOG. Hence, it is not really training in the academic way, but it was very efficient.
>
> *Informant 13*

4.7 Conclusions

At this point a model of tuning in the small can be considered. For a selected machine and programming model the tuner has to adopt the best data and control distribution policies. Knowledge is needed about how the machine, the compiler and the run-time libraries operate in order to drive performance oriented design decisions or to reason about the performance results. The assessment of the program performance is done using a tuning tool or timing functions.

[11]Thinking Machine Corporation.

Effective tuning is of great importance to the parallel programming user community and understanding the issues which contribute to the difficulty of the task may lead to better tool design. First, there are many models of parallel programming and despite the fact that they are designed to abstract away from the underlying machines, tuners still have to know a lot of details about how the parallel machine and the system software operate to obtain best results. There are three levels of tuning expertise:

- The novice tuner who may not know the details of the programming model and machine specifics that affect program performance. Tuners may try to overcome poor performance by changing their algorithms in the absence of appropriate knowledge of how to resolve a performance problem. In fact, the findings of this study described in Chapter Six verify that novice parallel programmers may embark on tuning without having any tuning strategy at all[12].

- The moderately experienced tuner who knows how underlying machine and compiler operate. Usually, this tuner has organised tuning knowledge in the form of heuristics. When these heuristics are applied, a positive outcome is expected.

- The expert tuner.
  The cause/effect chasm issue which emerged in the findings of this work implies that there is a third level of tuning experience which is accumulated from knowledge derived from the resolution of episodic tuning problems which cannot be easily explained. The experienced tuner has accumulated enough experience to recognise how other factors can contribute to program performance.

[12]In contrast, the programmers who participated in Eisenstadt's investigation were not novices (Chapter Three).

This study revealed a large number of problems regarding computer hardware and software support:

- Tuning may be performed on incompletely configured machines, and tuners may be exposed to inefficient system software.

- Apart from system failures, tuning is prolonged in cases where the tuner relies on the compiler to optimise the code. Such performance gains do not come for free as tuners have to check thoroughly program correctness after using an optimisation flag.

- Tuning can be postponed until reliable tools and sophisticated compilers become available in order to minimise the manual changes done in the large parallel codes.

- Parallel codes are ported to a number of machines and if tuners experience a sophisticated programming environment on one system, they expect to have the same level of support on other machines.

Tuning tool development is very often undertaken by computer scientists in the context of research projects (as shown in Chapter Two). The emphasis is on collecting as many performance metrics as possible and on displaying those metrics using sophisticated visualisation mechanisms. Not surprisingly, tuning tool design does not meet the needs of tuners who complain that tools are unreliable, difficult to understand and difficult to use with real world large codes. As a consequence, whilst it was clear that tuners need tuning tools during the whole process of code development and tuning, they tend to be used at the end of the development process for presentations or as a means to verify that everything runs as it is expected to run. Tuners do not participate in tool design and that results in tools which do not address important aspects of the tuning task.

Porting a parallel code across different machines and programming paradigms is very common and this introduces many more parameters. The program may be rewritten in a different programming paradigm/language e.g., it may be converted from data parallel to message passing and run on the same machine, or it may be ported to a different machine adopting possibly a different programming model. Compilers on even the same machine may differ in sophistication and this increases the tuners' burden of finding which optimisations are needed each time. Programmers who have adopted one particular programming model for a long time, show a certain degree of inertia to change to a new one. This is attributed to the long learning curve that is needed to be able to switch to a new machine and programming style and produce a parallel program which presents the same if not better behaviour than the old one. Whereas most of the problems that tuners face are attributed to cause/effect chasm and faulty assumption/model, almost two thirds of the tuners asked have never had any training in tuning, while almost 91% have never had access to tuning documentation. Access to tuning documentation and training could help in alleviating some of the problems in tuning.

It is easy to hypothesise that the phenomenon of the transfer effect[13] may occur during the programming model change. Further work can investigate the validity of this hypothesis and in particular how the transfer effect can occur and affect the tuning process. The findings of such an investigation could inform the design of parallel programming environments as well as provide useful information about how parallel programming and performance optimisation should be taught. The existence of many programming models may necessitate a more systematic approach to teaching performance optimisation which should attempt to see how

[13]Programmers who move from one programming model to another bring with them programming procedures which whilst appropriate to the old model, are not necessarily suitable to the new [153].

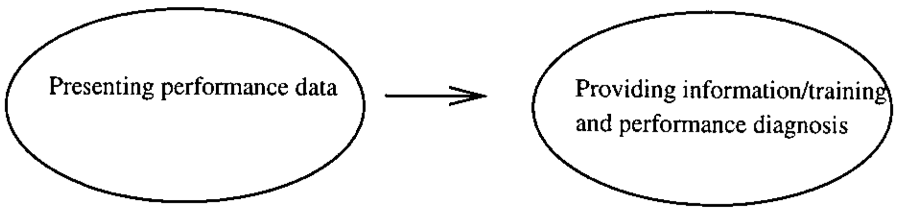

Presenting performance data
Providing information/training and performance diagnosis

Figure 4–4: Role of tuning tools changes

the transfer effect – if it exists – should be encouraged for similarities between programming models and minimised where differences are concerned.

The investigation of the difficulties that tuners face and the current computer support suggest that the role of tuning tools should change to assist with tuning where help is needed most. The emphasis should shift from the mere presentation of performance data to: transforming the data to meaningful visualisations of aspects of the program behaviour; education; and performance diagnosis. Figure 4–4 shows this transition in role in relation to the support required for addressing the problems of faulty assumption/model and cause/effect chasm.

The findings of this study can be used to inform a model for tool development and also propose the processes which could best serve this model (Figure 4–5). According to this model tool design should try to consider providing:

Portability and extensibility. Software engineering principles could assist with building the tool portable and extensible.

Reliable mechanisms to capture program behaviour. Issues that should be addressed here are the space requirements of trace files produced by large codes and the reliability of performance measurements. User participation at this stage could inform design about the requirements of large applications and the performance metrics that tuners need.

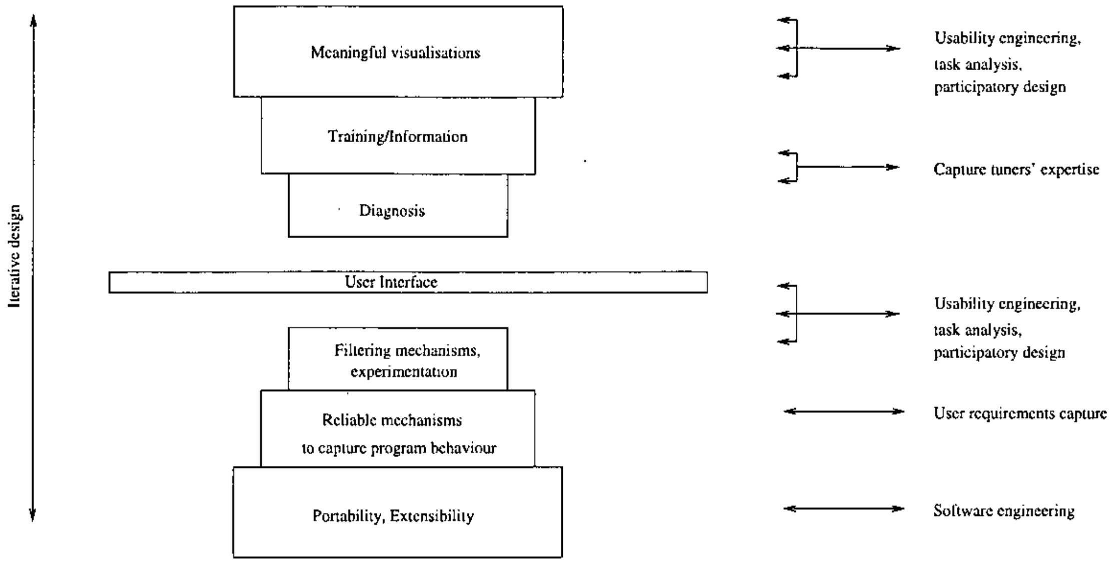

Meaningful visualisations
Training/Information
Diagnosis
User Interface
Filtering mechanisms, experimentation
Reliable mechanisms to capture program behaviour
Portability, Extensibility
Iterative design
Usability engineering, task analysis, participatory design
Capture tuners' expertise
Usability engineering, task analysis, participatory design
User requirements capture
Software engineering

Figure 4–5: Model and processes of tool development

Meaningful abstractions should be provided to abstract away from large quantities of low level detailed information to high level information which can be related to the entities in the source code that affect program performance. These abstractions should be designed based on the ways tuners perform the tuning task. Task analysis and participatory design could be employed to inform design at this stage. The repetitive nature of tuning should be considered to allow for experimentation management and program version control[14].

User interface. Interaction with the tool should be facilitated through a user interface. Usability engineering in the form of task analysis and tool evaluations could be used to design the user interface so that users can perform their tasks easily. The terminology adopted by the tool should be understandable by novice and experienced users.

[14]These issues are investigated further in Chapters Five, Six and Eight.

Tool design and development should be iterative and should be guided by user feedback and changes in the programming environment. The tool should assist tuners in a number of ways by providing:

A manageable set of visualisation means. Visualisation mechanisms should be used carefully and displays should correspond to aspects of the program behaviour that tuners are interested in. Again user participation could ensure that visualisations assist rather than impede tuning.

Information/training. To reduce the impact of the faulty assumption/model problem, tools could attempt, wherever possible, to demonstrate possible relations between performance determining factors and performance results. Adaptive interfaces could be used to provide background knowledge to novice tuners, which can be hidden when their expertise increases. Since it was obvious from this investigation that tuners often need to know a lot about the operation of system software and hardware, this knowledge can be conveyed by means of detailed documentation or by use of exemplar programs whose behaviour can demonstrate the factors which affect performance. Exemplar programs could exist in two forms: before and after the optimisation. Additionally, tuners could change some of the performance determining parameters in the exemplar applications and could investigate the results of those changes using the tuning tool[15]. A more systematic approach is needed to discover the nature of cause/effect chasm tuning problems. Due to the way this level of expertise is acquired, it cannot be anticipated by systematic training. In these cases, making a record of specific tuning experiences and making it available for others may be more effective. To this end, the exemplar programs could also correspond to exceptions i.e., situations where

[15]For example, a toy program could demonstrate the effect of varying the granularity of Regular Domain Decomposition on regular and irregular problems.

an unsuspected parameter may affect program performance. Background information should be provided for the cases which are not addressed by the exemplar applications. Hints of other external factors should also be given such as e.g., the workload of the machine or changes in the hardware configuration. The tool should also suggest tricks and techniques that tuners can use. These techniques could also be demonstrated by small example programs. The help of experienced tuners should be employed to capture tuning knowledge and expertise in tool design.

Performance diagnosis. Part of the responsibility of identifying performance bottlenecks should be undertaken by the tool which would narrow the search space that users have to consider. The tool could identify parts of the program where certain performance metrics have poor values. Meaningful observations should be made and the help of experienced tuners should be employed to ensure that observations are conveyed in meaningful ways to users.

Finally, a major problem with tuning tools is that they are often unavailable as their use remains local to the academic institution in which they were developed. More than half of the tuners asked answered that they have not used any externally developed tuning tool. Some of them had to develop their own tools which were tightly coupled with the problem they had to investigate. More communication is needed among tool developers, tool users and vendors of parallel computers so that tool development efforts are not isolated. The activities of the Parallel Tool consortium (PTools), described in detail in [71] are examples of efforts to foster such communication. The PTools consortium attempts to provide the communication framework among tool developers, users, and vendors, encourages tool development which relates as much as possible to parallel programming standards and promotes reusability, tool integration and portability.

Chapter 5

Tuning in the large

The previous chapter examined tuning by looking at the problems which tuners have to face in tuning in the small and the state of current tuning tool support as well as the support from available parallel software and hardware. This initial investigation pointed to some additional considerations which arise from the repetitive character of the tuning task and from the fact that parallel program tuning may be part of a software development process involving large programming teams. In particular, these considerations are pertinent to acting upon managing and sharing the information produced during tuning.

This chapter deals with work practices that tuners employ in order to manage tuning information as they go through a series of tuning efforts. These practices are studied in a wider context in the next chapter, which focuses on sharing this information with others.

In order to capture tuning work practices as accurately as possible, several data gathering techniques were employed: structured and unstructured interviews; questionnaires; *in situ* observations and verbal protocol analysis[1]. Local tuners

[1]In Chapter Seven the participation of tuners in the requirements and design specification of a tool for performance analysis and tuning is examined. Requirements engineering and participatory design attempt to put user practices and expertise in the

were observed during tuning and their tuning notes were collected and analysed. The approach taken by this research has a lot in common with ethnography, a methodology employed increasingly often as a requirements analysis technique. The ethnographic approach tries to study the social character of groups and the activities of their members in their natural settings. Its sociological emphasis means that it examines activities as socially organised from within their natural settings by participants of those settings [75]. Of particular importance here is the drive to uncover the tacit knowledge and implicit practices which are normally invisible to the casual observer, or are taken for granted, or are unexamined by task analytic approaches which seek to break down activities into discrete free standing components, without regard to how they are interwoven into a set of socially organised work activities. The ethnographic approach is valuable because it prescribes the need to understand requirements in the organisational context within which the system will be used and to critique many assumptions which are taken for granted [172].

The limits that an unobtrusive observational technique would impose on this study discouraged the use of ethnography as such. For example, the use of video recording [68] could be indicated if tuning practices occurred only in a confined environment e.g., an office room for a period of time. The combination of more direct tools such as interviews and questionnaires with occasional observations allowed a larger number of tuners to respond and provide this study with more information. In fact, interviews have been used in a number of studies such as: in [123] and in [122] which reported on the collaborative development of spreadsheets; in [124] which investigated the use of task-specific vs generic application software; and in

centre of system development. However, although it is clearly beneficial to include users in the software development process, this itself is not enough. Users are not always capable of describing work practices (tacit knowledge) and can be unaware of the organisational and cultural context of their work due to being over familiar with the work activities.

[141] where subjects were interviewed about their daily activities recorded in diaries. Similarly, an empirical study of software maintenance was described in [85] where the informants were interviewed before and after a maintenance task and work described in [11] studied the use of paper media for pseudo-code informal notations which helped with the cognitively complex task of programming.

In total, fifty three tuners contributed to this investigation. Twenty five of them worked in university establishments and the rest for companies that either manufacture or use parallel machines extensively. Their experience with parallel computing varied from one to eleven years (Figure 5–1). Forty two tuners filled in a questionnaire whereas eleven local and non-local tuners were interviewed. Unstructured interviews were used in the case of two non-local industrial people whose views nevertheless are included in this chapter. Seven of the eleven interviewees worked for companies and the rest for university establishments.

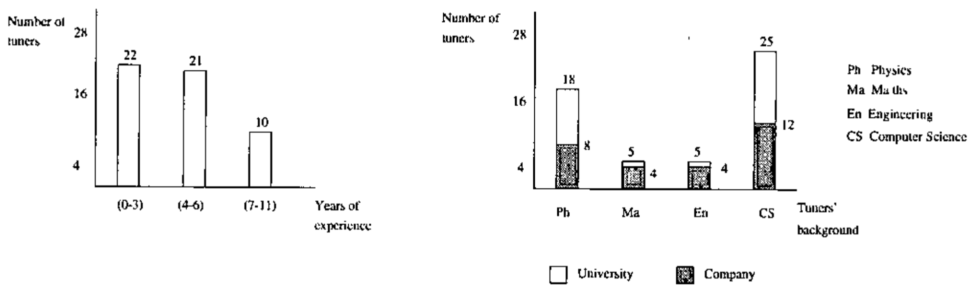


Figure 5–1: Profile of respondents

The first section of this chapter presents a model of tuning in the large. The second section examines tuning as a problem solving activity and discusses some of the ways in which tuners eliminate performance bottlenecks from their application programs. The third section focuses on how tuners document their efforts and the last section illustrates how porting parallel programs across different machines increases tuner requirements for efficient documentation.

5.1 Model of tuning in the large

The performance of a parallel program depends on a number of parameters which can be characterised as external or internal: the external parameters consist of the run-time environment of the parallel application and include the number of processors, the structure and size of the input problem and the machine. Internal parameters are the type of parallelism used, for example vector, data parallel or message passing and how much of the code is parallelised. Depending on the specific requirements of a project, the performance goal set for the program may be one of the following:

- For a specific machine, and a specific number of processors, the best parallelisation may be required as the structure or the size of the input problem data set varies.

- For a specific machine, and a specific problem size or structure, the best parallelisation may be required as the number of processors varies.

- For a specific problem size and/or structure, the best parallelisation may be wanted as the program is ported to different machines.

A particular performance goal is accomplished through eliminating all performance bottlenecks in the program. The resolution of a typical performance problem consists of a number of iterations of the unit tuning cycle, where each one is guided by the outcome(s) of its predecessor(s) (Figure 5–2). The program is run and its performance is assessed. During performance assessment the tuner attempts to relate the manifestation of poor performance (effect) to the cause of poor performance (values of performance determining parameters). A hypothesis is made as to what parameter values could improve performance according to a model (machine or software specific) that the tuner has. The parameters are set to these values and the experiment is iterated. Tuning in the large may therefore be thought of as a heuristic search within the program's performance space (defined by the set of performance-determining *parameters*) which continues until some optimum (or near optimum) outcome is obtained. Crucial to the efficient conclusion of such tasks is the availability of appropriate search management information. In the case of tuning, this information is generated as a by-product of each individual unit tuning cycle, or tuning *experiment.* Its importance points to the need to maintain adequate documentation of the task as it progresses [72].

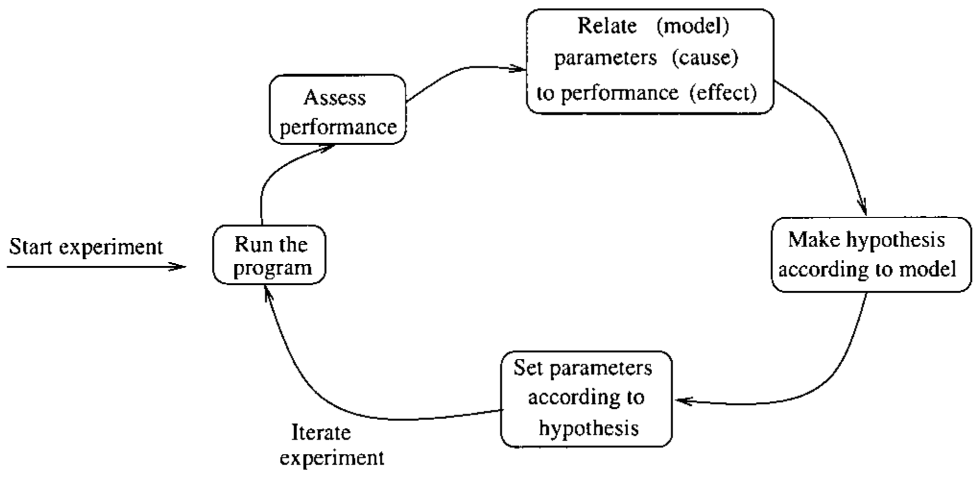

Relate (model) parameters (cause) to performance (effect)
Assess performance
Start experiment
Run the program
Make hypothesis according to model
Set parameters according to hypothesis
Iterate experiment

Figure 5–2: Resolution of a performance problem

The tuner may utilise records of tuning experiments in a number of ways. For example, the tuner may wish to keep for later reference all or part of the records of several experiments in order to document:

1. the problem under investigation and its manifestation in the trace data,

2. the parameter values, and the path through the search space,

3. performance metrics associated with each set of parameter values, and

4. the state of the program when tuning was concluded.

At any time during the course of tuning, the tuner may need to consult tuning experiment records in order to find out:

1. the parameter values that produced the best overall performance so far,

2. the parameter values that produced the best value for one specific performance metric, and

3. whether current performance is better than that achieved with a specific set of parameter values.

Examples of complete tuning problem case histories might be kept to create an archive which tuners may consult as a source of ideas and strategies for tackling new problems.

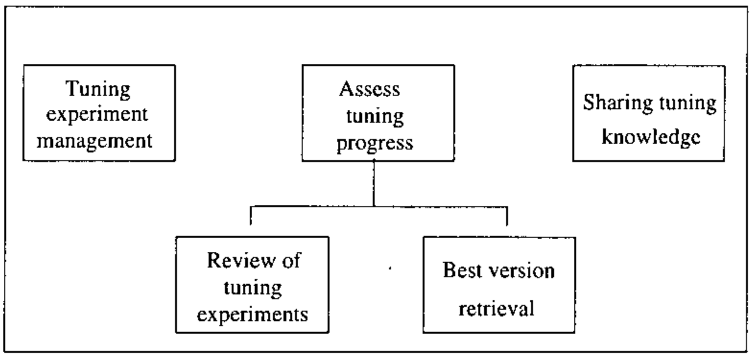


Figure 5–3: Tasks associated with tuning in the large

For the tuner, the difficulties associated with tuning in the large stem from the effort required to manage the tuning process and, in particular, its documentation — the capture, storage and retrieval of tuning experiment records (see Figure 5-3). Furthermore, in the context of an organisation where parallel program development and tuning are done in collaboration with other people, tuning results and tuning knowledge are shared with colleagues. An issue here is to what extent computer-based support may help the tuner to perform such tasks in collaboration with others.

The benefits of good documentation are widely extolled within the software engineering literature [135]. These can be summarised as follows [30]:

1. to support reasoning processes during software design,
2. to facilitate communication amongst the various members of the software development team, and
3. to further the accumulation and development of software design knowledge and experience from project to project.

Some of the many computer-based tools now available to support software documentation (e.g. [47,50,93,105,162]) might find application within parallel software development. However, as this investigation makes clear, there are significant qualitative differences in the documentation requirements of parallel software development, which reflect both the experimental, iterative nature of the tuning phase (as opposed to the more orderly and predictable course of conventional software development phases) and the large volumes of information that tuning generates.

To investigate tuning management and sharing of tuning knowledge further, tuners' practices within a number of organisations engaged in parallel software development have been investigated[2].

[2]In particular, tuning knowledge sharing and communication in large projects are examined in Chapter Six. A large part of this investigation is performed through case studies. Short case studies are presented in the main text whereas longer ones are presented summarised as boxed texts and are only commented for convenience.

5.2 Tuning problem solving

This study shows that a number of experimentation techniques are employed in the search for better performance [Be158], [Be159], [Be160], [Be161], [Be162], [Be142], [Be26][3]. Not all environments encourage experimentation, though. Large parallel code development is more conservative and extensive changes cannot be afforded. In these cases, tuning is incorporated within code development as programmers design the code to be both correct and efficient. Selective time measurement is then used to verify the correctness of the performance determining design decisions [Be13].

This study shows that tuners typically follow a worst case elimination strategy, where the major performance bottlenecks are eliminated first [Be129][4], [Be131], [Be132]. Some tuners start from a profile of the sequential code and continue with a gradual parallelisation:

> [Be163] I would profile the sequential code to see what improvements should be made in the parallel version. *Informant 61*

In the first instance, the sequential algorithm may be altered if this leads to better parallelisation [Be140]. In the remaining time, tuners may be occupied with examining the behaviour of the program in order to form a hypothesis as to what the parameters that affect the performance of the program are and so enable further progress to be achieved more systematically. Also, they may address the less expensive parts of the code. At this stage, having eliminated all major and minor bottlenecks, tuning may become an ad-hoc search as tuners may consider what

[3]Be denotes that the excerpt can be found in appendix B.

[4]Also in [Be133], [Be135], [Be136], [Be137], [Be138], [Be142] and [Be144].

other alternative code design decisions may yield some additional improvement[5]. As experience increases, tuning tools are used less frequently and only at important stages of the tuning project:

> [Be134] ...but then I did it more systematically and I made more extensive changes, as I gained more confidence. The frequency decreased if you like, initially, I made a lot of small changes and then, when I understood what the problem was actually, I made more extensive changes. Recently, I stopped using `apprentice` and I used time calls, possibly, now that I finished the load balancing and the tuning I may go back to `apprentice` to see at a high level how much the original version without the load balancing varies from the current one. Because the access to the T3D is limited, it motivated us against the idea of changing a lot of small things and see what was happening as quickly as possible. Perhaps, that was one of the reasons that the frequency of using `apprentice` decreased, because I realised that it is not possible, or appropriate. *Informant 45*

The next section discusses some of the problem solving tactics tuners employ.

5.2.1 Experimentation techniques

A number of runs may be used to establish the model of interactions among the various parameters that determine the performance of the program [Be26], [Be139], [Be141], [Be145], [Be146], [Be147] and [Be148]:

> [Be139] ...In many ways, you will need to have complete information about the profile or the task balance or whatever in your problem and therefore you would make certain assumptions that you will then try to investigate and you can either go about investigating in more detail the characteristics of your problem or you can try a strategy whereby you attempt to address one part of the problem, so you can address the load balance in isolation from the communication cost and try to observe the effect of the other, if you can actually measure it correctly. So, in many ways, you are not

[5]In [Be628], as there was no routine that stood out as being more expensive than the others, the tuner thought that new approaches could be tested, regarding the distribution of the arrays. The changes introduced at this stage did not improve performance and were abandoned as sometimes run-time errors were produced.

> actually looking for the most optimal solution. It would be nice if you could get the optimal solution to your problem in terms of load balance and communication cost, etc. But in general that wouldn't happen, and you look at ...you identify a problem and you address that problem and along the way you uncover other problems, so ...by addressing a problem like load imbalance, you would discover that the different load balance techniques would have a communication cost associated with them. And they may have other costs associated with them as well. *Informant 66*

Comparison of runs on different numbers of processors is a technique used to identify whether communications are done efficiently [Be152], or whether the problem is load balanced [Be151]. The scalability of a performance determining change may be tried on all sets of processors only if the result of the change depends on the number of processors. In the remaining cases, a change is tested on a random set of processors using a random data set [Be150].

After verifying that the control structure of the program is efficient, that there is no excessive synchronisation for example, an alternative communication harness may be employed such as lower level communication routines which will decrease the communication cost [Be149].

Alternative solutions may be compared in small experimental pieces of code before incorporating the best one in the application program [Be153], [Be154]:

> [Be153] ...And the way I solve these problems is that I develop two small programs, which have these two kinds of communications, and I see which is the faster one. *Informant 42*

An application code may be designed in such a way that optimisation can address separately and simultaneously different parts of it [Be155], which is useful in the context of multi-person program development efforts. In this context, the need to test the parallel code for correctness after each person's performance optimisation change prolongs tuning [Be37].

The input data size can be decreased in order to allow the tuner to optimise first the sequential program by mentally executing the program[6]:

> [e156] I did a lot of experiments with a small network so that I could check out the process of routing the messages. From that, I could see where the modelling approach was wrong and I could change the algorithm. After that, I could experiment with process mapping and granularity. But someone should start by optimising the sequential program. *Informant 43*

Conditions of time pressure may have an impact on the way tuning is performed e.g., tuning may have higher priority than coding. In [Be628], initial runs attempted to get an estimate of the program performance before all the intended changes were put into the code[7]. A specific part of the code, which was not going to be included in the timings[8], had produced a run-time error. The tuner decided that this part of the code could be excluded from the program being tuned and be debugged later by another colleague. Timings of the program processing a larger problem size were taken without the program processing the correct input file for that larger problem size.

Tuning is often done comparing the results of the program on two or more different machines. In [Be628], the tuner wanted to compare the performance of his program on machines *a* and *b*. Timing results were not available for the current version of the code on machine *b* so the tuner, who did not have access to machine *b*, had

[6]Eisenstadt has found that a lot of programmers run their programs mentally speculating about the possible causes of a program bug [44].

[7]The tuner's notes were written during a period of time that tuning was performed in somewhat pressurised conditions. The tuner was in a hurry to produce some good performance results on time for a publication deadline.

[8]This part of the code performed I/O. Due to the fact that the I/O parts of parallel codes are very expensive and very difficult to optimise, usually, they are not included in timing results presented in publications.

to compare the results on machine *a* with the results on a third machine *c* whose speed was comparable to *b*'s. As the time that the tuner had available on that machine was finishing, the tuner tried to organise a small series of experiments on a few sets of processors while timings on the remaining sets were extrapolated from older results.

In the presence of deadlines a quick and possibly "dirty" implementation may be pursued. Tuning may be performed conservatively when regression to a non-working program cannot be tolerated:

> [e157] We had to make the thing running for the demo. It changed the way we worked because when you are working with deadlines, the first thing you want to do is get the thing to run in a way you are able to live with. And then, when you do tuning you do it in a very careful way so as not to blow something really important. Maybe, if there were no deadlines, you would use more time to tune during development. *Informant 42*

Extensive experimentation may be expensive in terms of machine time and money. The time that a single parallel program can run on a parallel machine is finite when the machine is timesharing. For example, in [Be628] the run of the parallel program on one node was prohibited because the program had to occupy that node for sixteen hours.

5.2.2 Duration of tuning

The time taken to complete a particular tuning task (see Table 5–5) is a pointer to its complexity and to the importance of documentation. The fact that respondents reported that tuning typically lasted for several days or weeks suggests many iterations of the unit tuning cycle and the handling of large volumes of associated data. In such circumstances, the limitations of human memory would necessitate that the tuner at least document the sequence of changes made to the program to ensure that changes are not inadvertently repeated. One tuner commented:

> [e629] It varies by an infinite amount, depending on how well the original concept was thought out. Generalisations are impossible. *Informant 18*

[e630] A day to find and fix the first problem, several days or weeks to continue tuning till I am happy. *Informant 17*

question	less than a day	several days	several weeks	months
duration of tuning	3	21	21	3

Table 5–1: Duration of tuning

5.2.3 Size of the parameter set

The size of the performance-determining parameter set is another indicator of the complexity of the task. Adopting a faulty assumption/model was reported to be one of the major sources of difficulties in tuning (Chapter Four). When tuners have a wrong or incomplete model of the operation of the system software or hardware, extensive experimentation is needed to tune program performance and establish a correct model of relations amongst performance determining parameters. If the number of parameters is large, it will be difficult for the tuner to establish the relationship between parameter-value pairs and performance without some form of documentation.

Moderate to large numbers of parameters predominate slightly in this sample (see Table 5–2). The complexity of the task may be increased by the fact that parameters are often interdependent. Consequently, the optimum value for each parameter alone is not always the optimum for the parameter set. One tuner suggested performance prediction as a solution to this problem:

[e631] Because it might be the case to try one change by itself or two changes together to see if they work well together. So if you have three you have somehow ...you can't check everything so you want to predict which are going to work well together. *Informant 55*

During tuning one respondent reported on having to think about the following parameters:

> [e632] I had to deal with loop ordering, optimisation flags, debug switches left on, if statements in do loops, use of hand-coded vs BLAS libraries, work replication, use of cache. *Informant 40*

question	1-4	5-10	11-15	16-20
size of parameter set	20	25	2	2

Table 5–2: Size of parameter set

The parameter space may contain parameters such as the version of the compiler or the version of the program itself as in the following case where the tuner was assigned to optimise a program which was still under development:

> [e633] And also the particular version of the program we used. Another tricky thing about the project was trying to keep track of the changes the client made to the software. Also something that recently has become apparent ...it is also the compiler version number because the initial optimisations that the compiler did were not very extensive but now the compiler changed. *Informant 45*

5.2.4 Case study I: ad-hoc tuning management

This case study examines the impact that the lack of a tuning management tool can have on the effectiveness of the tuning process. Informant 44 agreed to participate in a think aloud session whilst tuning his program. The tuner wanted to find the optimal version of a routine which calculated an inner product. A fast inner product routine would improve the program performance because more than 50% of the execution time was spent in this routine. The tuner compiled and linked to the rest of the program an old version of the routine which he had developed before [Be600]. While the program was being compiled the tuner opened the file with that version of the inner product routine to make sure that the file contained the version he intended to time (Case study I, part a).

The tuner had devised a very simple way of differentiating among the different implementations of the inner product routine. He had to rely on remembering a correspondence between file name and version in order to retrieve the version he wanted. The tuner had put all ten files containing a different implementation of the routine in question in a directory. The names of these files differed only by the last part which was of the form: 1f.f, 1x.f, etc. The tuner looked for the file which contained the version he wanted and linked this file with the rest of his program. He compiled the file which contained the specific version without being absolutely sure he had got the right version: *I think this is the routine I was working with before.* The tuner had to remove from the file, which contained the call to the inner product routine, some calls to PVM (Parallel Virtual Machine, [9]) which remained from a previous experiment. This resulted into several compilation errors which the tuner was trying to correct. He wanted to run the program on one processor in order to make sure that the version still worked because it had been a long time since he last used this version of the inner product. Invoking the tuning tool would really produce more information than the tuner needed [Be601].

Case study I, part a

Summary: The tuning process was affected by the lack of an effective management tool. Informant 44 adopted an ad-hoc mechanism for version control which was not applied to all the modules of the program. For example, compilation mistakes were introduced because code changes corresponding to different tuning experiments were made in the same source code files. Such a way of managing the program versions proved to be inefficient in the long term since the tuner had to rely on his memory for differentiating between the versions. Rapid changes in the programming environment affected the tuning process since the tuner had to repeat old experiments in order to make sure that he compared versions under the same conditions. For this reason he did not use the timings of the old version, which were written in an on-line file, but instead ran that version once again (Case study I, part b). It was obvious from this study that the tuner postponed investigating a new alternative because it was difficult for him to obtain the necessary information from documentation available on-line. The search for the optimal solution stopped only temporarily, and he would come back to the specific routine at a later point in time.

The tuner had tried before the specific version of the inner product routine. For this routine, he had timing results in a on-line file. The tuner was not willing to use these results. Instead, he repeated the experiment once more because as he said he would like to make sure that the specific version still worked. Due to frequent changes in the programming environment, the tuner thought he had to test again the specific version before he experimented with an alternative one [Be602]. The decision to use a certain processor set for the execution of the program was based on the available resources and not on the best interest of the tuning process. The tuner had to run the program on a smaller set of processors because he would have to wait a lot of time in order to get access to a larger processor set. The optimal version of the inner product routine was found, but it could not guarantee that the program could always give right results. As the tuner said himself, the performance gain was too little to justify using a version which had a slight probability of introducing errors in the calculation. The tuner was not sure whether this conclusion was correct. He would have to read the man pages more carefully but he decided he did not want to spend more time on this problem. Provided that time allowed it, further performance improving options could always be considered in the future [Be603].

Case study I, part b

5.3 Documentation methods

All but one of the respondents reported that they employed some form of documentation (see Table 5–3). The totals exceed the number of respondents because some reported using more than one documentation method[9]. "Other" methods of documentation include: plotting graphs (1); using a versioning system (2) ([Be612], [Be613]); scripts (2)[10]; spreadsheets (1); a whiteboard (1); program output files (1); screen dump of the output of the profiling tool (1); and writing a report (7). Six of the seven people who answered that they wrote a report have used some other means of documentation. In fact, two of them used an on-line file, one used scripts, two used paper notes and one documented the source code.

[9]Three, twenty-nine, and eighteen respondents reported using three, two and one documentation methods, respectively. One respondent did not reply.

[10]In [Be610] and [Be611].

This day to day documentation was used later to produce a report, a scientific paper or part of a thesis. It is interesting to see that eight of the sixteen answers which belong to the "other" category imply the use of some computer tool. This added to the number of people who use an on-line file (eighteen) increases the number of people who use some form of organised documentation to twenty six.

Email responses emphasised the variety of documenting methods and tools:

> [e604] Keeping track of the changes; I don't find it difficult any more because I keep notes of what I do, because I was caught by that once. *Informant 64*

> [e607] Each concept involved in the optimisation process was documented. Followed by a `diff` type comparison to indicate actual source code changes for that concept. *Informant 1*

> [e608] If you do your quick and dirty testing it is the back of an envelope. If it is a serious routine, for example, I have been working on an inner product, then I document run times. If I want to do more analysis, I would try to use my timings which are written in a file. But if I have to know whether something is faster or not then I would just write it in my mind. Because there are not so many possibilities. Or I would keep the source file to remind me of the things that did and didn't work. *Informant 44*

> [e605] On pieces of paper. They were not organised. I often had to repeat the experiments to carry on from where I stopped.
>
> *Informant 15*

> [e606] The documentation we did was ad-hoc: a file containing notes on the various configurations we tried, and on the tuning results. These notes were never written up in any formal way, just kept for informal reference by the programmers.
>
> *Informant 17*

> [e609] I tend to keep sets of timings for each version in a spreadsheet allowing me to easily compare improvements or degradations between one version and the next. *Informant 13*

Informant 44 kept a log file of all the changes he made to his program, the dates of these changes and whether they were successful or not. He developed these documentation practices gradually as his expertise in performance optimisation

question	source code	on-line file	paper notes	other
documentation methods	28	18	23	16

Table 5–3: Documentation methods

increased. Asked whether he used to keep such a file back home[11], he answered: "not really, usually it was such a struggle to get things to run ...". It seems that the more the tuner's programming and tuning skills increase, the more the tuner is able to shift his attention to the management aspects of tuning. At the same time, the search space gets bigger since the tuner is aware of more parameters, which can possibly affect the performance of the program[12]. Apart from the log file, informant 44 kept a file containing timings for every version of his program. This file contained the execution times of his program run on different numbers of processors and using different input data. When a performance optimisation change altered substantially the execution time of the program, the runs were repeated on all sets of processors:

> [e621] These timings are for the optimised MPI version, i.e, the latest version. Timings for the previous version are in another file. After a code change, I check to see if it changes the timings; if it does, then a new set of timings will be needed. I keep the timings of the previous version and all the versions in a directory in order to have a development log.
>
> *Informant 44*

In the next case, informant 46 has created a file containing tuning notes by cutting and pasting some parts of the output of his program. A typical entry of the

[11]The participant was visiting Edinburgh Parallel Computer Centre.

[12]If sufficient tool support existed so that tuning experiments could be managed with relative ease, then tuners could perhaps focus on improving their understanding of performance determining parameters, i.e., they could create and improve models of what determines the performance of their programs.

programmer's on-line file is shown in Figure 5–4. The first line of this entry indicates the directory where the executable resides. The path also indicates the version of the program and the name of the Unix machine where the program was run. The entry is further identified by stating the value of a parameter in this case that "profiling" was "on" during that run. The entry includes the output of the program and the total execution time. The program output is used to record program correctness and memory usage.

```
acp/new/new.sentence3.galjas with profiling on
  found sentence 36800!!!!!!
  deleting dictionary ... done
  found a total of 36822 derivations
  32364.3 real    24492.4 user    49.2 sys
  table[8] = 497768, 3982144
  total allocated = 11234392
  overflow = 1088
  zoo50: 18.208 of 36400 mips, 896 Kb free
```

Figure 5–4: Tuning notes of Informant 46 ([e626])

In [Be627], informants 50 and 53 kept tuning results on a white board. The most important of these day to day notes were written down in a diary in order to record the porting and optimisation process. The diary included information such as the dates on which various activities took place; performance results; parameter values and program changes. This data formed the basis of an internal technical report. The report contained among other things a chart which summarised the performance of the code for every day's effort. Table 5-4 lists some entries of the tuning diary[13]. A second diary (not shown here) was also generated which logged a change in one of the parameters that controlled the execution of the program.

[13]The diary continues in Appendix B.

T21	NSTOP =20	lephys=.true.	lsslag= .false.
day	secs	mflops	notes
1	272	8.8	vector node, using only sparc processor (-novpu)
6	348	6.9	more work on cuadjtq, vectorised: qpassm, fswr
8	279	8.6	experiment to recompile all routines vectorised so far using only sparc processor (-novpu)
9	256	9.4	start with changing ifs, more work on vdfexcs
10	256	9.4	experiment noaltcode option, no improvement
11	222	10.8	recompiled with nproma=64 (was previously 32)
12	214	11.2	recompiled with nproma=128
13	18.1	132	recompiled meiko set of vectorised routines on
13	10.9	220	recompiled all routines on c90 (nproma=64) only source modification made was to change sqrthf to sqrt in lwttm and lwtt

Table 5–4: [Be627] Diary of porting IFS to CS-2

Case study II: tuning notes of informant 45

Informant 45 kept his tuning notes in a notebook on a day to day basis. In terms of the overall project, he had to produce a report which would deal with performance tuning issues and results with respect to the choice of data distribution. This case is an example of trying to determine the effect of the structure of the problem on program performance. The large number of factors that affected program performance necessitated a more systematic way of documenting the tuning progress:

> [e620] So, I kept results of these timings as well as the level of optimisation and the compiler version. Another option is whether `apprentice` is used because it introduces some perturbation. My documentation should be more systematic. I guess what I should do is use some short of form to make the experiments rather than using the book. At the moment, I also keep the makefiles and executables in different directories. *Informant 45*

His task was to design the best data distribution policy for the input data the parallel program had to process. The distribution of the data depended on the relative computational requirements of the different kinds of material encoded in the data. Each cell in the input matrix could be either metal, air or carbon fibre. Carbon fibre can be found in many fitnesses i.e., having different conductivity, and its computational requirement depended on this parameter. Informant 45 conducted several runs of the program processing only carbon fibre of different fitnesses and compared the execution time to corresponding runs which processed only air. In an excerpt from his log book, he wrote down how much more expensive it is to work on carbon fibre than simply on air: The first column (Nrings) in Table 5-5 represents the fitness of carbon fibre. The second column lists the actual time spent working on data consisting only of carbon fibre and the third lists the corresponding times of working on data consisting only of air.

Case study II

Nrings	Time	Greater than air
2	1.21	10.6
3	1.45	12.5
4	1.54	13.3

Table 5–5: Informant 45's notes on a series of tuning sessions [e625]

5.3.1 Case study III: tuning notes of informant 17

The entries in the on-line tuning notes file[14] of informant 17 were introduced on a day to day basis. Part of the notes are presented in Figure 5–5. An analysis of this file revealed that entries could actually be categorised in a number of classes:

Action done. An action such as that the program was compiled or was run was reported.

To do. The entry of this category mentioned things that the tuner would have to try to do in the immediate future.

[14]This file is included in appendix B as excerpt [Be628].

Report on a compile or run-time error (bug). In some cases the tuner would explain what caused the error, for example, the fact that a specific directive was not necessary. Usually, such an entry would be accompanied by the error messages the tuner was getting.

Code inclusion. At some parts of the file, the tuner would include some piece of code which created a problem or which was different in another co-developer's source file.

Report on a trick. An entry could describe a trick that improved performance e.g., a specific combination of compilation flags or setting the size of the arrays to a power of two. In a specific entry, the tuner said that what he discovered should be put in training reports.

Code Change. An entry could report on a code change, for example, on the addition of some special directives. In another case, the tuner would give justification for the adoption of a change.

Speed of the code. A report on the speed of the code, for example, "the code ran twice as fast".

Include results. Actual timing results were included, usually accompanied by the kind of processor configuration, data sizes and parameter values for the specific timings.

Comments on timing results. Timing results would be sometimes commented. For instance, the tuner wondered why the timing results in a specific case were so bad.

Mail message. A mail message sent from a person who was working on the same project was included and commented.

```
April 13
 -- After Jonas's changes are in, next thing is to add the axis
    permutation stuff and timing stuff (latter with ifdef's).
 -- NODEPENDENCE's in chksym3d.f possibly out of place; possibly
    unnecessary.
 -- Blew off the I/O changes in dump3d.f.
 -- BAH!  Gave up fooling with the DIFF's.  Found and grabbed Jonas's
    sources.  Highlights:
>          kew = kew + 0.5*wrho(i,j, 2  )*w(i,j, 2   ,tlevel)
>          kew = kew + 0.5*wrho(i,j,nz-1)*w(i,j,nz-1,tlevel)
April 15
 -- Recompiled with larger problem size (67 x 67 x 35) but don't have
    quite the right input file for it.  Results look promising, however.
 -- Preliminary numbers (pre-cshift) per iteration:
                      4PN           32PN          64PN
    35x35x35:         18.2           4.5           2.8
    67x67x35:         62.6          10.6
 -- Bug in CMAX_CSHIFT processing found and fixed.
                      4PN           32PN
    67x67x35:         59.6          10.6
 -- After bounds trimmed (so no masking in AVG and DIF ops):
                       4PN           32PN          64PN
    32x32x32:         11.0           2.0           1.33
    35x35x35:         17.2           3.7
 --  Using exact powers of two sure helps.
```

Figure 5–5: Excerpt from tuning notes

Apart from giving a record of the tuning process, documentation helped the tuner to coordinate his day to day activities. A closer study of the informant's notes revealed aspects of the task which are relevant to both tuning in the small and tuning in the large. The observations refer to the tuning process and how it is affected by the programming environment and the conditions under which tuning occurs.

5.3.2 Reasons for documenting tuning

It is interesting that the need to communicate results to colleagues is rated quite highly by respondents (see Table 5–6[15]). This, together with the last three reasons listed — aiding portability, report generation and consulting a document later[16] — emphasise the collaborative nature of parallel software development, and the importance — as in conventional software development — of sharing knowledge amongst project members. Some email respondents pointed to the difficulties ensuing when projects get interrupted before completion. Respondents also stressed the value of accumulating tuning case histories *across* projects: "to use as a learning aid and for future performance sessions":

> [e622] I can see what changes made the biggest differences hopefully to help me the next time I tune a parallel program. *Informant 21*

[15]Eight, thirty and twelve respondents gave three, two and one reasons for documenting tuning respectively.

[16]E.g. see [Be612], [Be613], [Be614], [Be615], [Be616], [Be617], [Be618] and [Be619].

reasons for documenting tuning	occurrences
as a reminder of progress	27
to keep track of changes	37
to aid communication with colleagues	16
to aid portability	3
report generation	5
consult a document later	8

Table 5–6: Reasons for documenting tuning

5.4 Case study IV: porting a program across many different machines

It was shown in the previous chapter how program porting across different machines exacerbates the difficulty of tuning in terms of increasing the volume of expertise that is required from the tuner. This section examines the documentation and versioning requirements that program porting induces.

Program versions proliferate in cases where the program is ported to other machines. Porting programs has almost always been motivated by the extra performance which can be gained by just using a more powerful machine. In some cases, though, program development has to be transferred to another machine because of the programming tools available on that machine. Sometimes, version creation may be automatically facilitated through conditional compilation. This is very common in the case of sequential programming. In parallel programming too, conditional compilation may be feasible when the programming model is the same as, for example, in [e623]. This investigation shows, however, that in cases where the programming model is changed as well, the changes that the program undergoes are generally too extensive to be handled by conditional compilation.

[e623] Generally, we keep only one version with conditional compilation to allow it to run on the machines we have here: Suns and SGIs (good debugging tools), Ncube-2, Intel iPSC/860, Intel Delta, Intel Paragon; this helps greatly in the ability to write code since the code can be debugged on a workstation or the Ncube or iPSC which have debuggers; the Delta has none and the Paragon one is only minimally working so far. *Informant 18*

1. A PVM version of the program was developed for a network of Hewlett Packard 9000/700 series workstations.
2. The PVM version was ported to a T3D emulator running remotely on a Cray C90 in Lausanne. Using the emulator avoided wasting T3D time while removing possible bugs resulting from switching from 64 bit to 32 bit arithmetic.
3. The program was ported and optimised on the T3D in Lausanne. This included inserting some shared arrays in place of the PVM code.
4. The program was ported to the T3D at EPCC in order to be further optimised:
    - The PVM version was converted to MPI and the solver part of the program optimised.
        - The tuner experimented with the use of a number of alternatives for the inner product routine. The tuner had already eliminated all sources of large bottlenecks and the inner product routine was almost optimal. The last alternative did not result in a major performance gain and endangered the program correctness. There was one more alternative that the tuner had to try, but this was postponed for a later time.
        - The MPI program used the BLAS routines and Fortran 90 array syntax wherever possible.
        - The tuner tried to optimise routines which were not critical for the program's performance.
        - The tuner shifted his attention to the problem of memory use. The program did not make efficient use of memory. The tuner was using a process to read in the input data and distribute it to the remaining processes. This master process ran out of memory space for larger problems.
        - The tuner changed the algorithm of the code on the master process by streamlining I/O and distributing/gathering of data. Another alternative that the tuner considered was to have two master processes in the program.
        - The tuner attempted to optimise further the inner product routine.
        - Finally, the tuner intended to try to experiment with the data decomposition scheme his program adopted.

Case study IV

Informant 44, a physicist from the Technical University of Delft had a computational fluid dynamics code which he brought over to Edinburgh Parallel Computing Centre in order to take advantage of the large supercomputing facility – a Cray T3D – available there. This allowed him to solve large fluid dynamics problems which he was not able to solve before, and also to optimise the performance of his code with the help of expertise available at the centre. The code development and tuning process is described below:

Informant 44's code was kept in three main variants, which marked three important stages in the code development. Version management in this case became more cumbersome because at a lower level he kept versions of code components which were created and changed during day to day tuning. In case extensive changes were required, he could create a different directory. When the code was sufficiently tested to prove that it enhanced performance, its changes were incorporated in the current version of the program. The different code versions were kept in case they were useful for other machines. Not all changes required the creation of a new directory:

> [e624] Well, if I know that I am going to change more than one routine I go to another directory, essentially, if I am optimising only one routine, I will optimise this routine separately and put it in my program. If I decide to change the program structure, for instance, because I want to change the memory usage, I will go to a different directory because I know that it will be an operation that will change most routines, so if I stay in the same directory, I will either have two versions of almost every routine in that directory, or I will lose my old versions. *Informant 44*

Program development and tuning would have to be continued possibly on another parallel machine and after the tuner's return to Delft.

The reconstruction of the development process in case study IV suggests that tuning may not be homogeneous as it may consist of interleaved stages which vary in the following ways:

- They may examine different aspects of the program which have an impact to performance. For example, one effort could focus on improving communications whereas the next effort would look at memory use.

- They may be different in the degree to which performance may be affected by them and not necessarily tackled in order of priority. For instance, the tuner would interrupt the improvement of a major source of performance loss in the program to investigate the feasibility of introducing minor performance improvements into other parts of the code (opportunistic tuning).

- They may differ in the tuner's confidence that they will produce a positive outcome. Some improvements are introduced opportunistically, without experience or belief suggesting that they may indeed optimise performance.

- They may differ in the extent of changes that the source code will have to undergo. Minor changes may be interleaved with more extensive ones.

Finally, some aspects of the program are revisited when time allows it or when the tuner has in the meanwhile acquired some additional knowledge.

5.5 Tool requirements

The typical duration of program optimisation, the large size of the parameter space, the fast changing programming environments and machines necessitate the systematic management of tuning. With adequate computer support, tuners could focus their efforts on improving the performance of their programs instead of building tools to assist with tuning management. Existing version control systems may be utilised, but they do not provide support for results capturing and tuning progress assessment:

> [e528] I suppose a starting point would be something like an `RCS`. It would freeze a particular version. It is something that we find difficult in dealing with [company]B because they don't have any support for version controlling. So it is to maintain different versions several times. Yes. Something like `RCS` but which could understand what composed an experiment.
>
> *Informant 48*

Formal documentation may not be able to assist with capturing little tricks that people do to improve performance resulting into tuning knowledge and rationale not being disseminated:

> [Be522]...But other problems we find are the continuous documentation of problems, hints tricks, things that people have found in use and have it up here (he showed his head) and they never wrote them down so that everyone else can use. Yes, the biggest problem is with the little tricks and hints, ways around the problems. *Informant 48*

The need for a tuning management tool had already been raised in the study and its requirements discussed in some detail [Be523], [Be524], [Be526], [Be527], [Be529].

> [e525] For instance you could use `SCCS` and `apprentice` could understand SCCS and when you want to go back in the source files to pick a previous version that would be a very nice extension to `apprentice`.
>
> *Informant 44*

5.6 Conclusions

The investigation of how tuners solve performance-related problems revealed that they rely on information provided by a number of program runs. This information is used to form hypotheses (tuning search) and evaluate their validity (assessing tuning progress):

Tuning search. Tuning search is heuristic and opportunistic. It may take a long time and may be suspended and resumed later. Major bottlenecks are addressed first, but tuners come back to a specific problem if in the meantime some more knowledge is acquired or time exists to allocate to it. Changes are attempted even though there are no indications that they may bring a performance improvement. Performance bottlenecks are found through a search which involves a number of controlled experiments:

- Initial runs can help tuners form a model of the interaction of the various performance determining parameters. A special case in this class are the runs which verify whether a particular problem e.g., load balance, exists.
- Runs of small pieces of code which test the effectiveness of alternative solutions. Before one of the solutions is adopted, it is compared with the others in a small piece of code in order to allow easy and cheap (in terms of tuner's and machine's time) experimentation.

Assessment of tuning progress. Tools may be invoked more than once to compare codes but most usually they are used initially when areas of poor performance have to be identified. Timing functions are used later as a flexible means to determine tuning progress. Three classes of runs could be distinguished:

- Runs which check the program correctness and the performance gain after a change in the code. When the scalability of a change is examined, runs are repeated on as many available sets of processors as possible. As the experience in tuning increases, these runs test progressively more changes at any one time.
- Production runs. After a version "freezes" the program is run on many processor sets in order to show the behaviour of the program as the input data changes or as the problem size changes.

- Runs where two versions of the code are compared under the same hardware or system software.

Within this framework of experimentation, program versions proliferate. Apart from the production versions, the code may exist in versions which are only developed for performance measurements. Input data sets may also be different for the purposes of tuning. Porting the program to a more powerful machine increases the number of versions that tuners have to consider. Tuning progress may not be monotonic; in case of regression the previous version is needed and other versions may be kept to be used on other machines. A survey of programmers showed that this is done in an ad-hoc, idiosyncratic way depending on the developers' own practices. Some case studies showed that ad-hoc ways of version control can only help in the short term or for small projects.

Documentation may be used to aid tuning management, but the responsibility of devising a documentation scheme lies solely on the tuner – with the exception of large software development organisations which may enforce their own documentation standards. Some tuners go to considerable efforts trying to organise the development and tuning process. To this end, a lot of time may be spent trying to utilise a number of tools such as graph drawing packages, scripts, or even spreadsheets that could support the experimentation process. Despite the valuable time spent on devising a documentation and/or version control scheme, such schemes are not completely automated and integrated into the development process. Hence, they can be inefficient at times. For instance, it is not easy for tuners to remember to adhere to the methods they themselves have employed. For example, in one case the tuner had included (see Table 5-5) some timing results into his logbook but there was not any indication about the other parameters which determined those program runs. Due to the rapid changes in the programming environment and the lack of adequate computer support, this documentation is sometimes regarded as outdated which results in the repetition of some tuning experiments.

The findings of this study suggest some of the requirements for a tuning management tool:

- Tuning experiment management

  The tool should cater for facilitating and recording the experimentation process and its by-products. It should provide for the storage of program versions which should be accompanied by a record of the performance determining parameters along with the performance result expressed in terms of performance metric-value pairs. Successful and unsuccessful efforts with their causes should be documented to record the program development path and for use as a learning aid for future efforts. The duration of the task may be so long that documentation, apart from giving a record of the tuning process, is invaluable as a reminder of the day to day activities and of the state of the program when tuning was stopped thus helping the tuner resume tuning at a later point in time. Automation of the experimentation process should be supported. For example, a number of program runs varying in some parameters (e.g. number of processors) could be performed automatically and experiment results could be recorded and correlated.

- Access to experiment information

  Tuning experiment management provides a record of the development and tuning process. The tool should allow the tuner to navigate through the program development paths in order to have access to the rationale of performance determining design decisions, to assess tuning progress, to keep track of the program changes and to convey design decisions and expertise to colleagues.

Another basic requirement for a tuning management tool is that it does not have a negative impact on the tuner's productivity. In the long term, the positive effects of the adoption of such a tool[17] may be easily understood, especially in the context of large collaborative projects. On the other hand, it is likely that the introduction of tuning management tools will degrade the developers' performance in the short term, unless special care is taken to incorporate as much as possible the management functionalities within the tuners' programming environment.

This chapter validated a model of tuning in the large which can be used to inform the design of a tuning management tool. Further investigation in the form of *in situ* observations is needed to further refine and validate the model. More studies of tuners' diaries may reveal many of the problems associated with trying to manage the tuning task as for example in [94].

Finally, parallel codes cross national borders as scientists in the quest of large computational speed travel to other countries in order to access new more powerful machines and take advantage of the expertise available locally. Since employing the help of an expert within the working environment is a very common way of solving a performance problem, the way tuning information and expertise is shared within and across organisations is studied in the following chapter.

[17]For example, work in [125] showed that documenting the development process may lead to error-free programming.

Chapter 6

Tuning across machines, people and organisations.

Chapter Four has examined tuning in the small i.e., the cognitive aspects of tuning and the available computer hardware and software support. Chapter Five examined issues which arise from .the repetitive nature of tuning and the need to manage the information that tuning produces along with the many program versions. This chapter deals more closely with tuning practices in the context of large application programs whose development involves the efforts of a number of people. In particular, the communication and information requirements are investigated along with the ways in which performance optimisation expertise is disseminated within and across organisations. The predominant method of investigation used was that of case studies which in particular looked at projects rather than individual tuning processes.

The first section of this chapter stresses that there is not a single approach to tuning. Whereas performance may be important for one organisation, it may not be such important an issue for another. The following two sections present two case studies of software development practices in two large institutions. These institutions have made large investments in purchasing parallel machines and key areas of their research activities rely on using them efficiently. In the first institution, code development and tuning was undertaken by many different and dispersed groups of people. The second case study looks more closely at code development undertaken by several groups within the same organisation. The case study presented in the fourth section looks at code development within parallel programming teams and examines how tuners optimise code written by others and how tuning experience is disseminated informally among group members. The next section presents two case studies of the ways tuning knowledge is requested and disseminated in an organisational context. The fifth section describes the case of an on-line query answering service found in Edinburgh Parallel Computing Centre. The final section summarises the findings of this study and discusses the implications for tuning tool design.

6.1 Different attitudes to tuning

Different attitudes to tuning can be found in different software development organisations. In some cases, people are not aware of even the simplest ways to improve the performance of their applications e.g., they do not know that certain compiler flags may optimise their codes [Ce30]. The investigation reported in Chapter Four has already shown that only a very small percentage of tuners have access to tuning training and documentation. On the other hand, the need to know a lot of details about the operation of system software and hardware is one of the most common sources of difficulties in tuning (faulty assumption/model).

Time constraints may necessitate that the emphasis is on getting the functionality right. In other cases, depending on the developers' experience, performance related decisions may occur during the initial design of the application [Ce28]. In [Ce8], parallelisation was done in a conservative way as the scientists who had developed the sequential code did not want the code to change a lot. The main objective was to design for performance rather than relying on trial and error, since this would endanger the quality of the code.

On the other hand, in some other organisations, there may be personal financial incentives to improve program performance. However, performance optimisation cannot be pursued to the degree of reducing the maintainability of the code:

> [Ce30] ...If we can get the operational model code run 10% faster, we get our salary increased. There is a value associated with that, but equally, we can't go and modify 90% percent of the code to gain this performance because we will get the code unmaintainable. *Informant 53*

Extensive optimisation may be unnecessary for certain kind of applications e.g., real time systems [Ce29], or may be postponed until tuners know the target machine their organisation is going to buy [Ce19][1]. In [Ce27], performance optimisation was deferred because of lack of funds. When funds were found, tuning the application program became less important because the funds were used to buy time on a more powerful machine.

To illustrate better the different approaches to parallel software development and tuning, the following two sections deal with work practices in two large organisations.

[1]A large weather forecast code had to be ported from a Cray C90 to a number of distributed memory machines. The program underwent some partial performance optimisation in order to have an estimate of how the code would perform across different machines prior to the organisation purchasing one. Porting was done in a number of stages.

6.2 Case study I: the LIBRA project

The original serial code from which the LIBRA code was derived was developed by a company called A. The code had 5,000 lines of Fortran implementing the basic numerical algorithm. The code was bought and parallelised by company B using the OCCAM language. The parallelised code used to run on a 128 node T800 Parsytec super-cluster. B soon realised that they could not achieve the performance targets they set themselves for the software.

B decided to solve this problem by buying new hardware. A decision to move over to a specific, more powerful machine (which never appeared) left B with no machine to run their code. Finally, B got a new Parsytec machine which did not have an OCCAM compiler. That meant that the OCCAM code had to be converted again back to Fortran using the Parix message passing library to handle communications. A long time was spent trying to convert the code back to Fortran.

During all this time, the code was augmented by company A with additional features. The code was used for production runs and all this time performance optimisation was not a issue due to the lack of funds to implement any optimisation plans. The funding was found and was used to buy time on a more powerful machine (Cray T3D).

Despite the fact that optimisation became less important, as the code could run on a bigger machine, B assigned to a third company C the task of improving the load balance of the code. Simultaneously with the efforts in C, B assigned to a fourth company D the optimisation of the numerical algorithm used in the code. In addition to that, B were also modifying the communications of the code (Case Study I).

The aims of C's assignment were: to analyse the performance of the code; to determine how performance could be improved on the target architectures; to modify the code so that it was portable across a number of machines; and to devise a better load balancing strategy by means of finding a better decomposition of the input data. The project did not also exclude the possibility of incorporating any numerical improvements developed by B or any other party. C assigned two programmers in this project.

It was very difficult for C's tuning experts to track B's development of the LIBRA code. Initially, they received a parallel version of the code containing Parix communication calls. The documentation which they got was incorporated in the source code. The user guide was incomplete and out of date. C's developers had to convert each Parix call to a corresponding MPI call. It was intended that the code should run on the T3D. Choosing to adopt MPI allowed the C team to save some of the scarce time allocated on the T3D by doing some of the development on a network of SUN workstations. Insufficient documentation impeded the process of understanding the code.

With each new code, the tuning experiments had to be repeated from the beginning. At that time, the code was using a regular domain decomposition, which proved to be very inefficient because not all the areas of the data set required the same amount of processing. The data consisted of three materials air, metal, and carbon fibre whose computational requirements, according to B, were rated 1:2:128 respectively. The rate of processing required for each type of material in the data set was not the same on the T3D[a]. When the 1:2:128 rate was used by C for the decomposition strategy on the T3D, the performance gain decreased. Thus, the computational complexity ratios had to be recalculated for each new version of the code B sent to C. Even though the contract between C and B did not cover any further optimisations, C's developers were able to identify and eliminate other sources of inefficiencies in the code [Ce24]. In the long term, the code will undergo a lot of changes and performance optimisation is bound to the development process [Ce506].

Case study I

[a]The reason was that the processing of the most expensive material i.e., the carbon fibre required a lot of floating point operations, which are very fast on the T3D, whereas the processing of the air data points involved a lot of memory copying, which is relatively expensive on the T3D.

Summary: In this case, code development spanned across different and diverse organisations. Four companies were involved in the development of the LIBRA application. Performance optimisation was performed by one company while the code was simultaneously being changed by two others. The communication and configuration management requirements were increased because all people involved needed to keep up with the latest changes in the code and the large number of versions that constant development and optimisation generated.

6.3 Case study II: development in X

The X organisation is concerned with medium weather forecasting. It is funded by the European Union and is responsible to all member countries. An operational ten day forecast is done on a nightly basis and is made available to the member countries. The research department is responsible for all the software for producing and analysing the forecast. Most of the people in this department are meteorologists who try to refine the model and the analysis program. The rest of the team are computer scientists who helped them to port and evaluate the performance of the program on new platforms (the "migration project") in order to choose one (Case study II, part a).

Four computer scientists assigned with the migration project contributed to this study. Their experience in parallel programming varied from two to seven years. Informant 52, the migration project coordinator, had participated in the first parallelisation of the serial code in 1985. Informant 50 was a computer scientist with five years experience in parallel computing and had a contract for a period of two years to help with the performance optimisation of the forecast code. Informant 50 had not done parallel programming as such, rather he had been optimising other people's code mostly on shared memory machines. Informant 53's role in the team was to evaluate existing parallel programming tools and investigate their potential use in the project. Informant 51 – with a background in meteorology as well – was responsible for producing the message passing versions of the code [Ce8].

Case study II, part a

The migration project necessitated a lot of communication among the group members and between the group and the meteorologists, who formed the rest of X's research department, in order to keep up with further developments of the forecast model and analysis code. In addition, a great deal of cross-discussion occurred within the group. According to the group coordinator this was a disadvantage in some cases (Case study II, part b).

> One group member with a lot of knowledge in the parallelisation of the forecast code started to work on an new area of the model. He was continuously being interrupted by members of the group and by some of the meteorologists to answer questions and solve problems, which diverted him from his assignments. Excessive informal communication created a number of problems for him.
>
> **Case study II, part b**

Additionally, code development took place in collaboration with people working for other organisations. A German company Z collaborated with X's developers to help them convert their initial vector code to message passing. Informant 51 had to cooperate with Z during the porting of the code to PARMACS. Communication was done through electronic mail and once informant 51 visited Germany to work closely with his collaborators there:

> [e507] The introduction of the PARMACS library was not done automatically. The first version, we have done it with the collaborators from Z; so they effectively told us how to do it. We talked to them about the application and showed them how we wanted the parallelisation to be done. And they did the real work; they showed us good techniques to follow because they have a lot of experience. We had to do it safely. As I said earlier they did the first version with the limited parallelism and then we did the next step, so we have learned from there, and we incorporated all the experience and the knowledge. *Informant 51*

The group had also to work closely with the vendors of parallel machines. Communication with the vendors was conducted mainly by electronic and postal mail. Every six months, X delivered to the vendors a new version of the code, which was under continuous development (Case study II, part c).

X delivered a short description on how to proceed from the source code files to the running version. X had tried to make the code flexible so that it could run on different architectures using switches. Vendors had only to specify whether the code was to run on a vector or a cache-based machine. Also, information had to be given as input to a preprocessor which changed the data distribution in the code. In one case, the forecast code was sent to Y (one of the vendors) to test it on this vendor's supercomputer. The results of the runs were delivered in a report, which in the introduction tried to establish an understanding of the tested code. This was carried out by describing the set of parameters that affected the run: the dimensions and the sizes of the input data; a number of parameters, which defined how much physical time the forecast code simulated; and a number of other parameters [Ce530]. This created the basis for comparisons with results coming from other vendors. The introduction continued with a report on the optimisation changes made to the basic code delivered by X.

Case study II, part c

All vendors ran and optimised a specified reduced version of the model on their parallel machines and sent the execution times and Mflops rates to X. X compared these execution times with the corresponding execution times on the local machine. The results were sent by postal or electronic email and were not stored on-line. People relied on the vendor to clearly specify which version of the code corresponded to the timing results. Back at X, the management of this information was not systematic.

The programming language used was Fortran. While the computer scientists tried to introduce more modern languages, the meteorologists, who had always been using Fortran, did not want to have to learn a new language. Another reason for which Fortran was established, was the fact that code development was undertaken by a lot of visiting scientists, who visited the centre for a short period of time and had to learn bits of the code and write others. It was not unusual for the visiting scientists to try to make modifications in a matter of weeks so they did not want to learn a new language. The design of the code was such that the high level parallel parts of the code were independent from the numerical algorithm routines. This meant that the scientists, who were usually interested in that part of the code, had only to familiarise themselves with the leaves of the calling tree below the level of parallelism. Any code that did message passing, for example, was invisible to them [Ce39].

> [e18] ...Another nice thing about it is the way we have coded it, we can split tuning into two tasks. Optimising the communication and optimising the computation. And they are really totally independent and two people can do that and that is also what we have done. Some of us are trying to get serial performance and some of us are trying to make more clever communication. And you are only doing calculations on a single node, so the optimisation is really to try to optimise the code on the single node. So the people, who don't want to know about the communications and the parallel code, can optimise the serial part separately.
>
> *Informant 51*

A white board was used extensively to support the documentation and communication needs of people sharing the same office and collaborating on the same project.

> [e505] It is a scratch pad for us instead of email, where you can both look at ...old fashioned scratch pads. When you get something that is worth of writing up then it is sent through the mail or it is reported. *Informant 50*

Case study II, part d

The constant development of the model necessitated that each new version would be tested for efficiency. It had to be ensured that the new version could vectorise and did not become significantly slower than the older one. X did not have any other way to check how much care someone had taken to make sure that their code could vectorise. Versions also proliferated with porting the program to many different machines. X's version control system did not live up to the expectations of the programmers:

> [e502] The way we are working here today is from this point of view very haphazard. We have a lot of versions, and this is a significant problem. Although, we have a version control system, the guys in the parallel world don't use it because it is installed on a different system from the one they work on . . . so they are reluctant to use it and we are at the situation where there are lots of tar files lying there and it is slightly difficult to follow what you have. *Informant 52*

The development strategy and the programming language adopted were chosen to meet the needs of the multidisciplinary character of the research department (Case study II, part d).

Summary: This second case study illustrated that parallel code development can be undertaken by a large number of people with different backgrounds and roles in the organisation. Moreover, the composition of the programming team may change quite frequently, thus increasing the requirements for adequate documentation and efficient communication. At the same time, important parts of the development process may occur outside the organisation. Economic interests drive the involvement of vendors in the optimisation process since it is to their benefit when parallel application codes perform well on their hardware [Ce510]. In addition, part of the parallelisation and optimisation may also be sub-contracted to other organisations.

6.4 Tuning within the group

This study shows that less experienced programmers turn to more experienced tuners or consultants for help [Ce511], [Ce513], [Ce517], [Ce800]. Reassurance about the effectiveness of a solution may be sought before the programmer attempts to validate a hypothesis, especially if there is a lot of work involved in introducing a prospective performance optimisation change [Ce514]. Help is sought when the solution to the performance problem is not known:

> [e512] And you think how could I improve that. So basically, you speak to Malcolm and he says look there is a wonderful routine that TMC has written, why don't you use this? *Informant 68*

Parallel programming experience and tuning knowledge are shared within the context of the organisation the individual developer works in [Ce515], [e18]:

> [e516] I share my notes with other people, but mostly in an informal way: whenever I talk to somebody about parallel programs, I may end up telling them about my experiences. I have also at times made reference to these experiences in actual seminars that I've given. *Informant 16*

The experience of the people who had first to deal with advanced performance optimisation issues is shared informally and will remain within the organisation as long as the experts work for it [Ce520]. When people were asked how they would introduce another person in the project, they almost all answered that this would require an expert sitting together with the new team member in front of the workstation. On the other hand, even an expert, after being away from the project for a while, would need special pointers which reflect the current state of the project [Ce521]. Documentation could serve the purpose of communicating performance design decisions and results so that members could build on experiences of their colleagues. This study shows, however, that the time overhead involved in maintaining documentation in order to share techniques and experiences with others may put off the realisation of such a scheme [Ce504]. Instead, low sophistication media may be used [Ce503].

In a few cases, performance tuning knowledge is concentrated within a course in order to disseminate tuning techniques within an organisation:

> [e519] There is ultimately no substitute for native cunning. I have, for example, encapsulated the general principles of optimisation in a parallel environment that everyone should know into an 1 hour seminar which I give in an introductory CM-5 course. I usually follow this with taking a real life example (hopefully one of the class members' pet problem) and offering approaches that may be tried to optimise it. Unfortunately, time constraints mean that people can't actually try out the approaches there and then. *Informant 1*

In [Be628], the tuner tried to incorporate the changes that a colleague had made into the program. Initially, he tried to isolate the different parts of the code using the Unix `diff` command, but he did not succeed in distinguishing the differences. Instead, he used all the code his colleague had written. Moreover, he wasted time comparing timings of two different versions of the program on two different machines before he actually realised that he was comparing different versions. Not all versions of the code were ported to all machines available. This particular porting was the responsibility of another programmer who did not inform the tuner that he had not finished the porting. This illustrates how insufficient communication and the lack of collaborative programming tools may result in inefficient use of time.

Facilitating the growth of knowledge within an organisation and its application to future projects was identified as a critical issue in some early discussions [36] and other research has pointed to the importance of employing an informal apprenticeship model within an organisation [17]. In recognition of this, recent research has proposed systems for capturing programming knowledge and organisational memory [33,161]. For example, Terveen et al. in [160] have identified the existence of an important type of software design knowledge that they call *community specific folklore* and have focused on the problems associated with trying to manage it in large software development organisations. More specifically, they have noticed that much of the design knowledge required to be a successful developer is community specific in that it concerns the application domain, the existing software base, and local programming conventions. Knowledge is informally maintained and disseminated by experienced developers. This process is ineffective: not everyone gets the knowledge they require; inefficient communication of knowledge takes more and more time (e.g. see [Ce800]). The process is also fragile: loss of key personnel can mean loss of design knowledge. Terveen et al. propose a system for capturing and disseminating design knowledge throughout the software development organisation. The major consideration, though, in the development and adoption of such a knowledge base is the cost-benefit trade off.

6.4.1 Somebody else's code

When the tasks of porting the code to a parallel machine and optimising the parallel code are considered to be difficult, they are assigned to an "expert" who is, in most cases, either a computer scientist or a scientist long exposed to parallel computing. Sometimes the tuning expert may have an interdisciplinary background. These experts are recruited to be members of the scientific team developing the code or work for companies which act as consultants or undertake themselves the tasks of parallelising and tuning. For instance, informant 47's group cooperated with a company in Lyon in order to reduce the risk involved in porting the code to a parallel machine. The code was first parallelised locally and then was sent to Lyon. The experts in this company produced a report on the best ways to improve the code performance.

In some cases, the extent of the optimisations performed may be affected:

> [e1] Optimising code for others is problem repeating work, which discourages any large scale optimisations. *Informant 49*

During the early stages of the introduction of parallel computing in an organisation, not everybody will be exposed to parallel programming and this may affect the code development and documentation processes. For example, in [Ce501], the documentation had to be less technical in order to reach as many people as possible. In the early stages, the sequential code may also pre-exist its parallel counterpart. In cases where the development of the sequential code is an on-going process, the parallel code has to anticipate this development. This may require additional communication amongst the people responsible for the sequential and parallel development respectively. For example, informant 44 worked on the parallelisation of a sequential program which was still under development. The program was written by a large number of people and each was responsible for a part of it. Informant 44 tried to keep up with the changes in the sequential program but this was a onerous task:

> [e508] I go mad. I tried to keep up with the updates of everyone. I got three updates every day. I tried to keep as many common routines – the matrix construction, or the time integration, or the boundary value determination routines – as possible but daily changes in the serial program structure and parameter lists made this impossible. Everything each of them are doing affects me. If I change something, it affects only a very small part in everyone. Now, I quited from trying to keep up with them. Because it was too difficult. At that point in time (i.e. at the beginning of the parallelisation process), I was changing all the program. And I was not allowed to do my changes to their program because, still, parallel programming in Delft was considered unproven technology. *Informant 44*

Working with application programs written by someone else is difficult in cases where there is insufficient documentation. While most tuners emphasise the need to understand the structure of the code, other more experienced tuners (provided that the programming model allows it) may work at a higher level, dealing only with specific parts of program which are candidates for parallelisation [Ce2], [Ce3]. In every case, it is crucial to be able to communicate with the original developers of the code. Communication with developers or access to documentation can provide experts with the rationale of design decisions so that they can establish more easily the link between program performance and its cause (cause/effect chasm) or identify the model of relations between performance determining parameters upon which code development was based. For example, due to lack of documentation, it was really important for informant 51, who was assigned the parallelisation of a particular code, to be able to communicate face-to-face with the scientists who wrote it. Communication with the sequential programmers provided him with an understanding of the data structures and the data dependencies within the program. This understanding was far more crucial for the parallelisation process than any course in parallel computing could be. Informant 51 was fortunate because his assignment coincided with the presence of two visiting scientists, who had investigated a number of parallelisation techniques for the program. The work of these scientists was not documented in any formal way and, instead, informant 51 had to work closely with them for a period of time [Ce509].

Finally, as was identified in Chapter Four, optimising codes for others can be very difficult when these codes are not well written and special support is needed to cater for the increased communication and documentation requirements emerging in these cases. Old, messy codes are difficult to parallelise [Ce4]. In the following case, attempting to understand or structure the serial program prolonged the task of parallelisation:

> [e5] Structuring the code and modularising it, which if it were a decent piece of code in the first instance, we wouldn't have to do this. I had to tidy up the serial code before actually I could parallelise it. I think the sequential core of the program was very easy to parallelise. *Informant 47*

6.5 Case study III: EPCC User Support Service

In cases where organisations do not own parallel computing facilities, parallel programmers access remotely machines available elsewhere. Similarly, access to tuning advice may be facilitated through user support services available at these sites.

EPCC has long established an on-line, electronic mail based user support service. Users can send an e-mail to a specific address, stating what the problem is and expect an answer within the next one or two days. The queries and their answers are kept on-line. Seventy-five performance related queries were studied with EPCC's permission. Four queries were two-fold and five were three-fold so they were treated as different queries. This raised the number of queries to eighty nine. Queries were analysed and classified in seven categories and Table 6–1 lists the frequencies of the occurrences of the different classes:

Specific advice. This category included queries which questioned the necessity and impact of old techniques in the light of a recent compiler change; asked what performance optimising flag options could be used with the current compiler; asked for reassurance that a technique could bring a performance gain[2]. Other queries in the class were concerned with how to realise a solution to a performance problem. For example, one tuner wanted to know what synchronisations a very low-level communication harness required or where to look for information on the most relevant optimisation techniques. Other parallel programmers wanted to know how to measure the performance of their programs or stated that they did not understand how a particular compiler directive works. In a number of queries, programmers asked how I/O in their codes could be improved.

The impact of compiler changes on tuning can be large. Code optimisation techniques have to be reviewed in the light of new compiler versions. Even when documentation is readily available and up to date – which may not be the case (e.g. [Ce64]) – it is difficult to know to which extent old techniques are still necessary or even harmless. The situation is further complicated if part of the system software is not optimised for the changes introduced by the new version of the compiler [Ce63]. The problems of this class verify that *Change* identified in Chapter Four is indeed the reason for many problems in tuning.

Why is that happening. Queries of this class sought an explanation for program behaviour which tuners could not understand on their own. One of the most interesting questions was concerned with a parallel code whose timings were drastically different from one day to another [Ce65]. In this case, a compiler upgrade and some hardware changes worsened the performance. It

[2]Often the technique programmers were intending to use was not correct.

was suggested to the user to experiment with the old version of the compiler in order to verify that the cause of the problem was indeed the compiler upgrade. Tuners are quite vulnerable to software or hardware changes since it is difficult for them to know what impact they can have on the performance of their programs.

Also, programmers may not be aware of the way instrumentation may affect their codes. Extra instrumentation code may affect cache coherency as in [Ce58], where the tuner complained that the program produced different results when the code was compiled with the `apprentice` flag enabled. Questions in this class demonstrate the impact of the *cause/effect chasm* identified as the most common source of difficulties in tuning (Chapter Four). Cases where changes in the programming environment led to degradation of program performance can also be identified as instances of *cause/effect chasm* problems as tuners cannot know when these changes occur or what their impact is on the performance of their programs.

Look at my code. In this class a code was submitted along with the query. Two kinds of queries were identified: those in which the tuner asked generally for suggestions on further optimisations and those in which the tuner asked the expert to focus particularly on specific parts of the code. Again queries of this class reveal an absence of knowledge about a model which explains the relations amongst performance determining parameters and performance.

> This is my code. It only achieves 6.6 Mflops could you suggest ways to improve it? Q5

> I am trying to write some good global sums. Could you look at my code and see if there is a better way of doing it? Q41

Figures. A large number of tuners asked for typical values of various metrics which could have an impact to their code, such as e.g the bandwidth and latency rates of various communication calls:

> What are the typical maximum Mflops rates that one can achieve on the T3D? Q24a

> What is the clock speed of the front end? Q23

Run-time error. This is an interesting category as users asked for help about a particular bug or run-time error that they were encountering. They were not aware, though, that the poor performance of their codes was the reason for these problems. Experts answered all these questions with advice on how a performance improving technique would remove the problem. For example, in one case a user was asking how to do check-pointing because the execution time of his program was longer than the maximum allowable run time. It was estimated that the program could run for a day with a large part of the execution time being spent in doing I/O. The expert made a number of performance improving suggestions which alleviated the need for check-pointing. In [Ce61], a run-time error revealed that the performance of the output phase should be looked at. Even though the manifestation of the problems in this category is not directly related to performance, queries of this class can be regarded as *cause/effect chasm* or *faulty assumption/model* problems.

General advice. The queries in this class were looking for general information on efficient parallel programming. Some programmers asked for information about efficient data and work distribution techniques. Tuners who did not have any hardware or software specific performance model asked for information about such models. Performance expectations may be too high as, for example, in [Ce60] where the user thought that the peak performance of the machine can actually be achieved and sustained.

> We would appreciate any information on efficient data and work distribution techniques for large arrays. Q7

> Could you give us some example codes that managed to perform up to 130 Mflops per second on the T3D? Q32

Tools. Complaints about not being able to understand the suggestions of the performance analysis tool or some critiques about the correctness of the on-line documentation of the tool were included here.

Often tuners are not familiar with the terminology adopted by the performance analysis tool. Novice users without a computer science background, find it very difficult to understand the information provided by the tool. In [Ce56] and [Ce57], the tuners could not make use of the performance observations the tool provided because they could not understand them.

[Ce57] Q: I have a question concerned with the Apprentice tool. In the COSTS window, the bar chart always shows that Integer Adds is the dominant operation in the code. However, I expect that the code should perform floating point operations most of the time. Is there a conversion from floating point operation to integer operation inside the T3D? This will affect my judgement about how fast the code runs. In the OBSERVATIONS, two measures are printed:
1. floating point operations per second.
2. integer operations per second.
Usually, measure 2 is twice as large as measure 1. Only when I know which type of operation is dominant in the code, can I determine how fast the code actually runs. I noticed that after the new cf77 compiler was installed, the ratio of measure 2 and measure 1 has decreased for the same calculation.

Category	Occurrences
Specific advice	41
Why is that happening	13
Look at my code	10
Figures	9
Run-time error	8
General advice	4
Tools	3

Table 6–1: What tuners ask

Summary: The questions that tuners ask verify the validity of the framework for studying tuning (Chapter Four). Indeed, inadequate tools, having to know the specifics of the underlying machines and programming environments that keep changing are major sources of difficulties for tuners. Approximately, 39%[3] of tuners did not know what affected the performance of their program. This implies that programmers may embark on parallel programming without first having an understanding of the basic issues that affect performance. This problem can be addressed by adequate training. On the other hand, almost half of the queries examined ($\simeq$ 46%) were asking for advice about the use of a specific performance determining parameter or about improving a particular part of code. This suggests that programmers may know what could improve the performance of their program, but they do not have enough information to implement it. For example, users can be occasional tuners, while the programming environments change far too quickly for them to keep up with the changes. Thus, changes in the programming environment can even lead to faulty assumption/model or cause/effect problems: the models that tuners have are made obsolete and novice tuners cannot explain easily changes in the program performance.

A lot of the information that tuners lacked was included in documentation available on-line. Studies have shown, however, that users are reluctant to consult manuals, which may be re-inforced here by the fact that tuning information may be difficult to extract from within lengthy documents whose contents span a number of issues. Tuning information will have to be better organised and conveyed before tuners start using it extensively.

Typically, queries were answered within a period of one or two days. More complex queries took more time to be answered as experts dealt with them by looking more closely at the users' code or having a meeting with the users. Selected queries

[3]The occurrences of the categories "Why is that happening", "Look at my code", "Run-time error" and "General advice" were added together.

(Figure 6–1) were summarised in a way to reflect what are the issues with which users need expert help.

6.6 Conclusions

This chapter examined tuning in the context of large application programs whose development involved the efforts of many people working for more than one organisations. Three levels of communications were identified:

- Within the programming team

  Large application programs can be developed by a number of people who usually work on parts of the code which have some distinct functionality and depend on other parts in terms of their input or output data. Communication needs in large projects are increased. Performance optimisation poses more requirements for documenting, conveying and justifying design decisions for the different parts of the program which separately and combined affect the program performance.

- Between groups of the same organisation

  The difficulty of parallelising efficiently sequential codes has resulted in "tuning experts" undertaking this task on behalf of scientists, who are typically sequential programmers. Programming teams are extended with these experts who may have a computer science background. In some cases, the experts may form a separate group responsible for the parallelisation and optimisation of the application program.

- Between organisations

  Alternatively, experts working for consultancy companies undertake optimisation tasks. Parts of the program development itself can also be under-

ID	Context	Knows	Does not know
Q1	Information sought on how compiler works after compiler change	Array sizes affect performance	In the light of the new compiler how much and in what way array sizes affect performance. Some of the system libraries are optimised for the old compiler
Q2	Advice on general optimisation techniques for improving IO	Which part of the code needs improvement	Anything about how IO is performed and what are the factors that can affect performance
Q3	Documentation on alternative communication library	Existence of libraries with optimised maths functions	How to access information on the optimised library
Q5	Code is sent to be optimised		Anything about options within the performance model that have a different impact on performance, how the compiler distributes arrays and executes directives. Techniques that can be used to optimise performance
Q8	Problem with opening too many files	Knows the problem has to do with the large number of files	The reason that the files cannot be opened simultaneously, how IO is done, technique to sequentialise the opening of the files
Q10a	Apprentice distorts timings		Reason apprentice distorts timings, how instrumentation is done. What is the impact on performance
Q12	Tool showed where time was spent	Where time is spent	Basic issues such as what affects performance and what techniques could be applied in that particular case
Q14	Picking wrong documentation on compiler	Optimisation flags exist which can be used to improve performance	That two compilers exist one for the front end and another for the mpp system. How to invoke information on the one for the mpp system
Q19	Timings of the same code were longer after a specific date	Suspects a hardware or compiler problem	When the compiler changes, how much and why a compiler change can affect the code, how much a hardware upgrade or a hardware error can affect the program
Q20	Mathematical functions perform poorly compared to their performance on other machines	Knows that they perform badly	The design of the processor chip is the reason for the poor performance of the maths functions
Q23	How much code performance depends on the speed of the front end	Speed of the FE machine affects performance	Relation between the front end and the actual CM, what affects the speed of the code and how the front end interacts with the CM
Q36	Poor performing code compared to the same code running on other machines	Knows where the problem is	Too high expectations from machine based on peak performance advertised by vendors. What aspects of the machine design affect performance

Figure 6–1: What tuners need to know

taken by other companies. In all cases, it is essential to communicate with the people who wrote the sequential code. Vendors are also getting involved in the development process as they attempt to increase their competitive advantages.

Program design and tuning knowledge are shared informally within groups and organisations.

The extensive porting of parallel programs across different machines and programming paradigms results in increased communications between all parties involved in the process. At the same time, the number of program versions proliferates and this increases both the documentation and communication requirements for efficient management of the development process. Clearly, the design of technology to support tuning should take into account the accomplishment of the tuning task as part of the software development process and within a context of collaboration with colleagues and clients. In building such a technology it would be critical to examine the ways in which tuners work and participate in activities of their colleagues and the tacit procedures through which work is systematically accomplished. For example, different tools may be needed to support program design communication and documentation requirements within the team and among organisations. Conventional HCI and requirements engineering techniques tend to be insufficient in recognising and catering for collaborative tasks [107]. Rather, methodologies which are adopted by the social sciences (e.g. ethnography) would be more suitable for revealing hidden work practices and guiding the system design.

Part of the communication within and across organisations occurs when tuning knowledge is disseminated. Parallel programmers seek advice from more experienced colleagues; tuners discuss with each other their experiences; advice is even sought and exchanged through news bulletin boards. The study of environments, where tuning expert advice is offered to users of parallel computers, revealed the problems that tuners encounter and how help is provided to them.

In particular, two reasons were identified for these problems: lack of training and lack of information with respect to fast changing programming environments. The availability of public domain parallel programming environments such as PVM or MPI, which can run on clusters of workstations has brought parallelism within the reach of many people who often embark on parallel programming, without the necessary understanding of the basic issues due to lack of training (Chapter Four). Moreover, tuners need fast access to information about changes in programming environments.

The inefficiencies of sharing tuning knowledge informally and the drawbacks of the consultancy mechanisms imply that there is clearly a need for the creation of a project/organisation tuning repository for sharing tuning knowledge within and outside an organisation. Eisenstadt in [44] proposed a similar repository of debugging knowledge. Observation of performance optimisation courses may shed light on how performance knowledge can be encapsulated and presented within a constrained medium such as a minimum on-line document.

Chapter 7

VISPAT: a VISualisation and Performance Analysis Tool

A series of design meetings with local tuners defined the requirements and design specification which were used as the basis for the initial version of VISPAT, which targeted programs that run on clusters of workstations using CHIMP (Common High level Interface to Message Passing) [45] and PUL (Parallel Utilities Libraries) [32]. In addition, results from the observational study of tuning (Chapter Four) contributed to the specification of the requirements for VISPAT. A subsequent cycle of requirements capture, design and evaluation resulted in changes to VISPAT in order to cater for SPMD programs using the MPI (Message Passing Interface) standard which had already started being used by local programmers [116]. Several requirements which had been identified in the first series of meetings, were finally addressed by the third version of VISPAT. This chapter presents VISPAT whereas Chapter Eight describes its participative design process.

The outputs of the requirements capture process were eventually categorised into one of four fundamental design issues: trace data format; filtering mechanisms; display options; and the management of tuning over time. The first three are concerned with *tuning in the small* whereas the last issue is relevant to *tuning in the large* and is discussed in Chapter Nine.

The first section of this chapter presents a brief overview of VISPAT. The second section discusses the concept of hierarchical phases which can be used to reduce the volume of data the user has to consider and to relate program behaviour to the source code. Reducing the volume of performance data and relating the program behaviour to the source code can assist tuners to establish links between the cause and the effect of poor performance in the program (Chapter Four). The following three sections discuss the ways information about program execution is gathered, processed and visualised by the graphical front end of the tool. The sixth section presents an example of using VISPAT to demonstrate some of its human-computer interaction features. The final section is concerned with future work.

7.1 Overview of VISPAT

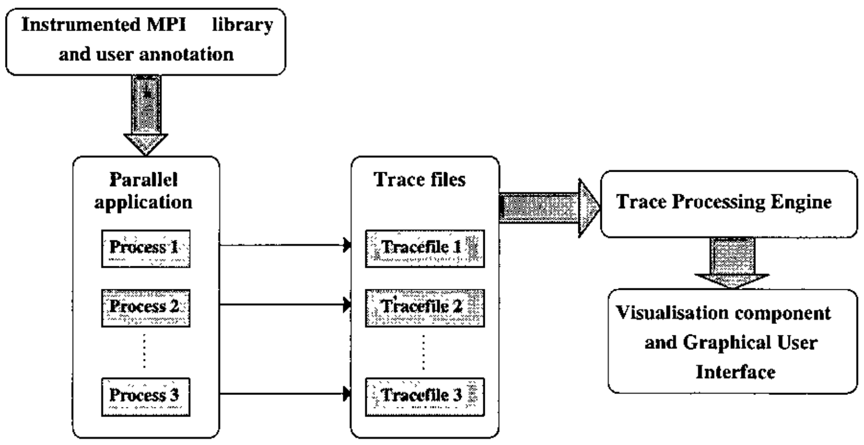


Figure 7–1: Overview of VISPAT

VISPAT conforms to the structure identified in most port-mortem tuning tools. The structure of VISPAT is shown in Figure 7–1. It consists of:

an instrumentation component which instruments the program so that when it is run, information describing the program's performance behaviour can be recorded in trace files,

a trace processing component which analyses the information contained in the trace files and

a visualisation component[1] which depicts the analysed and filtered information.

7.2 The concept of phases

A phase is an abstraction representing part of the total program execution time and can correspond to a library function or a programmer-defined logical part of the source code. Phases can be defined hierarchically as aggregations of other phases, thereby providing the tuner with multiple layers of abstractions. Their function is to provide the bridge between the cause and effect of poor performance. The most abstract layer may be concerned with logical stages evident in the execution of parallel programs such as initialisation, computation and closing. Each of these top level phases may consist of a series of sub-phases, which can have sub-phases of their own and so on.

As the investigation described in Chapter Four has shown, one of the problems in current performance visualisation tools is that a large volume of data is displayed without allowing tuners to determine the level of detail of data presentation or relate it to the source code. Viewing the application program as a hierarchy of phases allows tuners to focus only on the interesting parts of the program, thus reducing the amount of performance data they have to consider and also relate this data back to the source code. Source-code reference is provided by a number of tuning environments [2,79,118,171], which map low level events to higher level

[1]The instrumentation and trace processing components and part of the visualisation component of VISPAT were implemented by EPCC Summer Scholarship Students.

ones [8]. In the initial design of VISPAT for CHIMP and PUL programs, event abstraction was more extensive because the phase hierarchy was imposed only partially by user-defined events which included lower-level library functions. More complex phase structures were created because a parallel program could use a combination of parallel programming libraries built on top of CHIMP, the basic message passing harness. Figure 7-2 depicts the structure of a parallel program which uses CHIMP for point to point communications, PUL_EM for collective communications and PUL_RD for regular domain decomposition. It can be easily seen in this figure that the function rd_operate, for example, consists of a number of simpler PUL_RD functions and of a user-supplied code. It was a user requirement that a tuner can isolate views of such a parallel program which could consist of one class of phases e.g., CHIMP, or of a combination of classes e.g., CHIMP and PUL_EM phases, at various levels of the phase hierarchy.

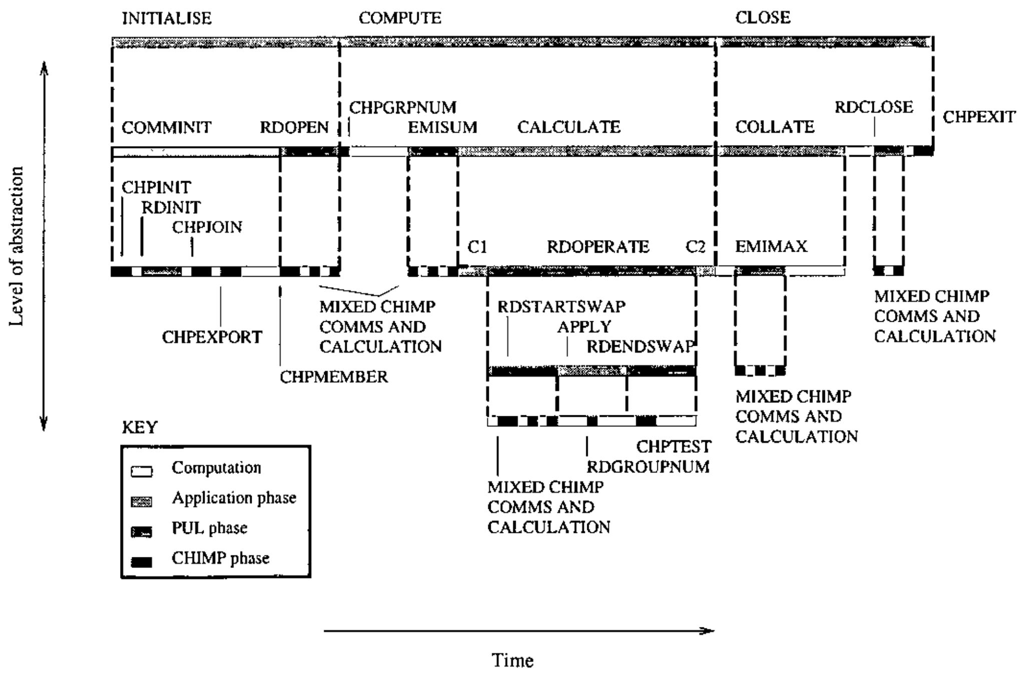


Figure 7–2: Phase hierarchy in a CHIMP/PUL program

7.3 Trace data format

The trace data format requirements were largely determined by the programming environment and the requirement for source code reference[2]. Currently, MPI is the standard message passing interface for parallel application and library programming [116]. It caters for point to point communication between pairs of processes and collective operations among groups of processes. Its more advanced features provide, amongst other things, for the manipulation of process groups and their topological structure. A local implementation of the MPI standard has been developed at Edinburgh Parallel Computing Centre based on CHIMP [26, 45].

Information about the program execution is generated by instrumenting the MPI library. MPI comes with a name-shifting interface to enable profiling of the MPI routines. The advantage of the name-shifting mechanism is that there is no need to change the source code in order to use the instrumented version of MPI. MPI programs have only to be linked with the instrumented version of the MPI library [152][3]. The resulting executable generates trace files that can be processed by VISPAT's **T**race **P**rocessing **E**ngine (TPE). Each process in the program creates one binary trace file containing all the information relevant to that process[4]. The processed information can then be visualised by the visualisation component of the tool (Figure 7-1).

[2]User requirements are shown in Chapter Four.

[3]The instrumented version of the MPI library was implemented by EPCC [152].

[4]The binary format drastically reduces the space requirements of trace files corresponding to long program runs.

Each instrumented MPI function corresponds to a *phase* in the execution of the process calling this function. A phase is identified by the name of the function, together with a number which distinguishes each unique occurrence of it within the trace file. It is also characterised by its class which can be either MPI or Application (if the phase is programmer-defined). The instrumented MPI functions record information such as the begin and end time stamps of the phase along with other information. In addition, the instrumented MPI library records a phase that begins and ends when MPI_Init and MPI_Finalize are called respectively. Thus, the life time of a process is recorded as a single phase that in turn contains various phases corresponding to MPI function calls and other interesting parts of the program. The format of the trace file is described more thoroughly in [166].

User requirements in relation to the instrumentation of MPI were concerned among others with three issues: recording information about the communication phases; recording the end of MPI non-blocking communications; and introducing a unique communicator name scheme[5].

Apart from phase time stamps the instrumented functions record information such as: the number and the type of elements sent or received; the tag used for the communication; the communicator; the rank of the sending or receiving process; the request handle for the non-blocking communications[6]; and the rank of the root process in a collective call, e.g. MPI_Bcast[7].

[5]A communicator defines a group of processes and a context of communication between them. A message sent in one context cannot be received in another context [116]. For the same group of processes there may be defined more than one communication context i.e., communicators.

[6]Data which identifies the non-blocking communication [116].

[7]MPI_Bcast performs a broadcast communication.

The actual completion time of a non-blocking communication initiated by a non-blocking MPI send or receive is not known. A non-blocking communication ends when an MPI_Wait or MPI_Test call succeeds. To match the beginnings and endings of non-blocking communications, the MPI instrumentation library stores the request handles returned by these function calls. Instrumented versions of the MPI_Wait and MPI_Test calls record the end of these non-blocking communications by recording their corresponding handles. Further processing of the trace file matches the request handles returned by the non-blocking function calls with the request handles returned by the MPI_Wait and MPI_Test calls.

Although communication in MPI occurs in well defined communication contexts i.e., the communicators, these are identified by handles which are local to each process. This is inconvenient for the users who want to consider a point to point communication, for example, as happens within a communicator with a global name. The instrumentation library provides a unique naming scheme for communicators which helps tuners understand their communications better [152].

In order to reduce the amount of trace data produced and to enable tuners control the parts of the program for which instrumentation is applied, instrumentation can be selective. Special library calls can be inserted in the program source code that can switch on and off instrumentation around parts of the program which the user is interested in investigating.

There is a one-to-one relationship between the structure of phases in a program and the structure of the trace files. This is carried through into VISPAT's visualisation facilities which are described further on in this chapter.

7.4 Data filtering

A means to analyse and filter the trace data was identified as a requirement by both local users who took part in VISPAT's design meetings and by many participants of the investigation described in Chapter Four. This requirement is realised through the Trace Processing Engine (TPE)[8]. Each process in the application program generates its own trace file. TPE reads in and processes all the trace files. Processing involves analysing traces and storing them in flexible and efficient data structures. Along with the data structures, the Trace Processing Engine provides a set of analysis functions which can perform a large number of filtering operations on the data structures.

For each process and for each trace file, TPE generates a tree of all the phases that occurred in that process. Each node consists of phase data such as: the starting and ending times; phase name; phase class (MPI or user-defined); and five pointers to other nodes namely the parent, left and right sibling and leftmost and rightmost child. Additionally, for each node there are two fields which determine whether the phase is visible or selected. Phases which have the *selected* field set can participate in a number of operations. A phase node and a complete phase tree are shown in Figure 7–3 and Figure 7–4 respectively.

In addition to the phase tree, a list of all the communication events of a process is constructed which is later merged in chronological order with the communication event lists of all the other processes. The communication event list is a doubly linked list. As shown in Figure 7–5, each node of the linked list represents either a BEG-COMM or an END-COMM event and contains the time that the communication phase started or finished respectively. The node contains also a pointer to

[8]TPE was implemented by EPCC.

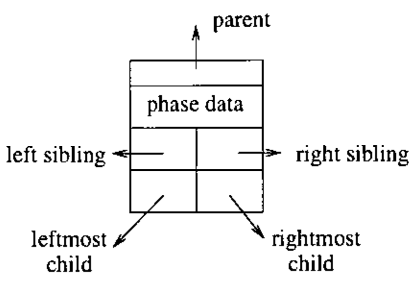


Figure 7–3: A phase node

more information about the communication such as: the type of communication (blocking or non-blocking); the source and destination processes; and a unique integer number which relates the send and receive messages among different processes.

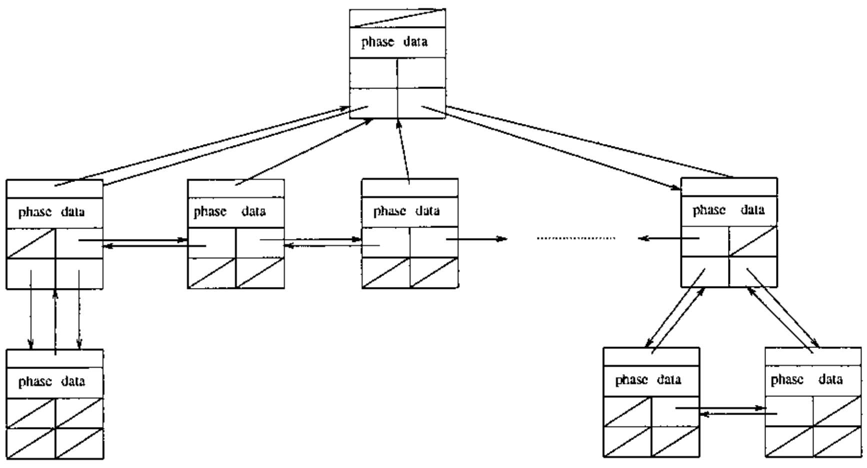


Figure 7–4: A phase tree

A number of functions provide an interface to the data stored in the communication event lists by:

- checking whether an event is a BEG-COMM or an END-COMM event,
- checking whether an event is a point-to-point or a collective communication,

- getting the event type (blocking or non-blocking),
- getting the name of the event and
- determining the communicator within which communication has happened.

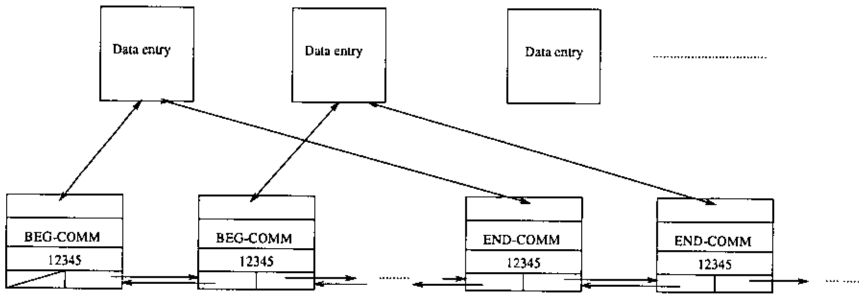


Figure 7–5: Communication event list

Other data structures that TPE creates include a *join* event list. The instrumented MPI library records all the communicator creation events. Whenever a process creates a communicator with other processes, it is said to *join* that communicator. The ranks[9] processes assume when they join a communicator, are local to this communicator. TPE is able through a simple mechanism, to translate the local ranks of processes participating in communications to their global process identity in the MPI_COMM_WORLD communicator[10]. This join event list is merged with the join lists of all the other processes and is used by an animated display which shows the processes joining the communicators over time. In addition, TPE creates a list of all the communicators in the program with information about the member processes and their ranks in the communicator. Apart from the mechanism which

[9]Rank is the "local" identity of a process member of the set of processes that communicate within a communicator context.

[10]That is the set of all the processes in the program.

translates local process ranks in a communicator to their global process identity, TPE contains a number of functions which can:

- locate a communicator with a particular identity,
- check if a communicator is an intra-communicator or an inter-communicator[11],
- check if a process is a member of a particular communicator,
- find out the rank of a process in a communicator (using this process's global identity) and
- find out the identities of the two intra-communicators which form an inter-communicator.

In summary, the TPE analysis functions act as a query mechanism on the created data structures and can provide information that can be used by the front end of the tool either as data to be presented to the user or as the means to perform further filterings and abstractions on performance data. Providing meaningful abstractions has been recognised as a key requirement for tuning tools in order to help users investigate more effectively causes of poor performance and thus, alleviate the problems associated with the cause/effect chasm (Chapter Four). The role of TPE is evident in the following section.

[11]Communication that takes place within a group of processes, occurs within the context of an intra-communicator whereas communication between processes that belong to separate process groups occurs within the context of an inter-communicator.

7.5 Visualisation

The design of the visualisation component of the tool was to a large extent driven by the requirement for source code reference. This requirement was mainly realised by depicting the phase hierarchy of each process in the application program. In this hierarchical presentation of events and phases, a mechanism to help the tuner identify events of interest is important and addresses a second requirement for reducing the amount of data and making the data meaningful. *Data navigation* provides the tuner with a means of determining what data will be subsequently visualised by the performance displays. The tuner should be able to determine interactively: a region of the trace file (pan over the data); the time grain (zoom in or out of the chosen region); which events will be visible (filter out unwanted events); and, finally, control the level of abstraction (fold or unfold phases).

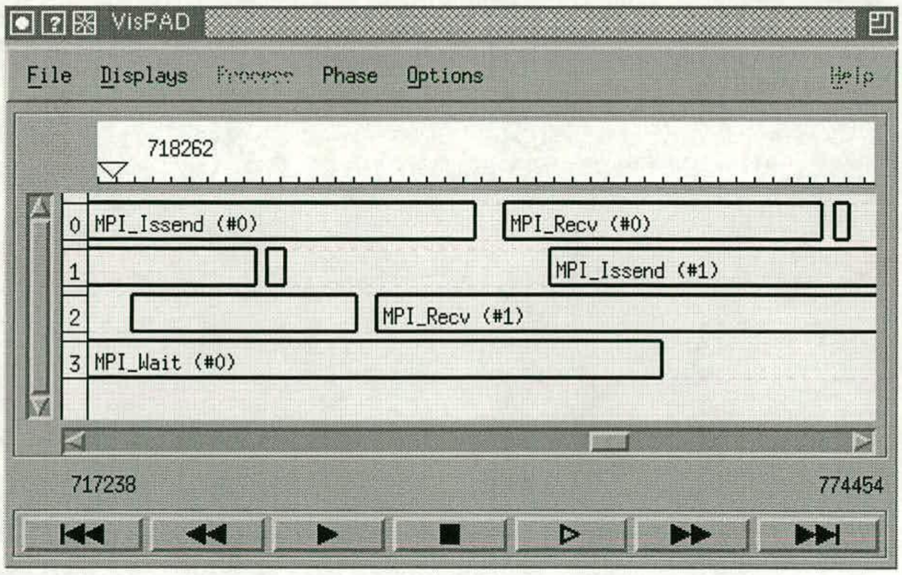


Figure 7–6: Navigation display

These requirements were realised through a single user interface mechanism – the **Navigation** display – a form of Gantt chart (see Figure 7–6). The Navigation display determines the context of data visualisation because it has a central role in accordance with another design requirement. VISPAT's other displays should render data only over the time period and parts of the program that are currently visible within it, Figure 7–7[12]. Apart from the Navigation Display the current set of VISPAT's displays includes the Communication, the Statistics, the Membership Matrix and the Profile displays.

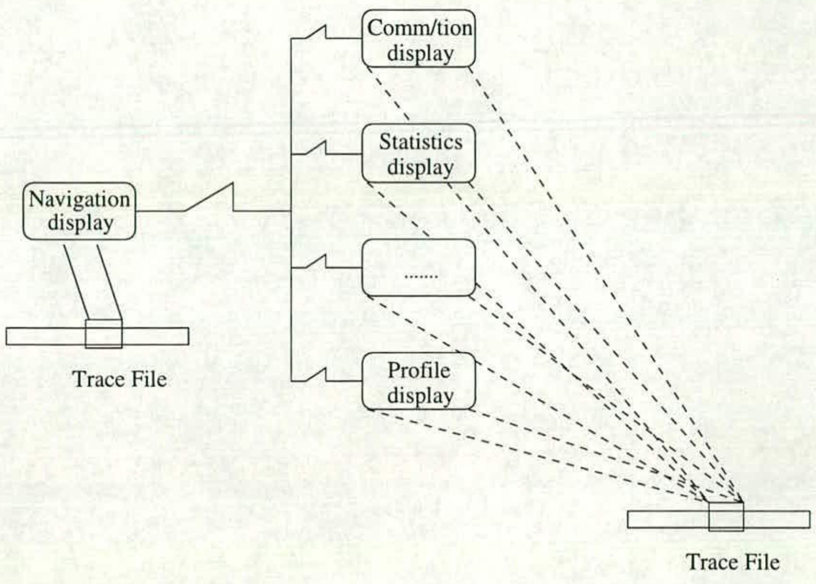


Figure 7–7: The central role of the Navigation display

7.5.1 Navigation display

The Navigation display renders the parallel event histories of the processes in the program. It is a Gantt chart where the time line is depicted on the horizontal axis and the set of processes on the vertical axis. Each process occupies a horizontal strip of the display where all the interesting events of that process are depicted as they take place over time. The Navigation display provides the context for

[12]This feature has not yet been implemented fully.

the visualisation of trace and performance data. When used in combination with VISPAT's phase abstraction mechanisms, this context also provides the means for achieving source code reference. The Navigation display enables the tuner to browse or systematically move around within the space of program phases. Navigation can be performed in both a horizontal fashion, i.e., forwards and backwards over time, or in a vertical fashion i.e., up and down the phase hierarchy. Traversing the various levels of the phase hierarchy is achieved through unfolding and folding, with the additional option of filtering. This not only allows for the transfer of the focus of visualisation to a higher or lower level of abstraction, but also for the view to be limited to specified phases. The other data visualisation displays can show events only for those phases visible on the Navigation display after all filterings and abstractions are applied. In this way tuners can relate trace data to specific parts of the program. Providing visualisations of meaningful abstractions, allows for looking for the *effect* of poor performance at a higher level of abstraction instead of examining the large space of the program's low level trace data. Apart from finding out faster where the program botttlenecks are, filterings assist tuners to establish easier the *cause* of these problems by controlling the amount of detail in which performance data is presented.

The function of the current version of the Navigation display was cooperatively designed with users during VISPAT's design meetings and can be categorised into two distinct groups: operations on phases and operations on processes.

The first group consists of the **Selection**, **Unfolding/Folding** and **Filtering** mechanisms:

Selection — when phases are chosen for expansion or filtering. A selection can be global or local. Global phase selection ensures that the current instance and all subsequent instances of a phase will be selected. Local selection means that only the current instance of a phase is selected. The view of selected phases can then be either enhanced or removed. Selection is possible through the support mechanism of TPE which provides functions which traverse the phase trees and mark a phase as either selected or unselected. An **Unselect All** option makes all the selected phases unselected.

Unfolding/Folding — where selected phases can be unfolded into their sub-phases (or sub-phases folded into their parent phases). There are two ways of unfolding a phase. Unfolding is either *simple* or it is *specific*. In the former, a phase can be unfolded into all its sub-phases independently of the class to which they belong. In the latter, the parent phase is unfolded only into instances of a nominated phase. This phase may be either an MPI or a programmer-defined phase. For example, the user may want to examine all the MPI_Send phases included in a user-defined phase which contains other types of MPI function calls as well. By unfolding specifically the parent user-defined phase into its MPI_Send children only, a view of the program is displayed containing only these MPI_Send phases. In terms of TPE operations, this involves making a phase invisible and its children visible. If p is a phase which is unfolded, p*'s* sub-phases can be obtained by following p*'s* leftmost child pointer and then following a chain of right sibling pointers.

On the other hand, folding a number of phases into their parent phase can serve as a means of controlling the amount of information displayed on the Navigation display. After a number of phases are examined by the tuner they can be folded into their parent phases and removed from the current view using the *filtering* mechanism which is described below.

Filtering — which realises a more immediate abstraction mechanism by enabling the user to keep in the Navigation display only specific phases. For example, the user may choose to expand to the lower level of abstraction and then have a look at all the instances of an MPI_Send extracting from the current view all the phases which are not MPI_Sends[13]. In this manner, the trace data can be abstracted in two orthogonal directions and the phase hierarchy traversed horizontally and vertically. The TPE operation which supports this mechanism traverses the phase trees and marks as visible only the phases whose type is MPI_Send.

GoTop level. After a number of expansions and filterings the user may want to return back to the top level of the phase hierarchy.

Undo dismisses the result of the last operation.

The second category of Navigation display operations focuses on process abstractions. Process abstraction makes user interaction with the tool more flexible. At the same time, the scalability of the Navigation display is increased[14]:

Elimination — whereby those processes whose behaviour is not currently of interest can be removed from the screen.

Re-ordering — whereby processes whose combined behaviour is of interest to the tuner can be moved to occupy subsequent lines in the Navigation display.

One of the requirements identified during design meetings was for a means to hide uninteresting periods of time or periods of time where the behaviour of the

[13]The filtering and folding/unfolding functionalities address major user requirements as the original programming environment for which VISPAT was designed included a number of parallel programming libraries which could all co-exist within a single program. It was thus necessary to provide ways in which tuners could create views of the program corresponding to the different kinds of library functions used.

[14]VISPAT targets programs running on clusters of workstations where usually the number of processes is small.

program has already been analysed. For this, a mechanism for *marking* periods of the execution time was designed[15]. A set of n markers divide time in $n+1$ periods. The user can assign names to the markers and characterise the periods within them as interesting or uninteresting. An uninterrupted animation of the trace files will skip all the periods marked as uninteresting. When the animation is stopped (interrupted), the **Fast Forward** and **Rewind** buttons move animation time forwards and backwards stopping at the nearest markers.

7.5.2 Communication display

The Communication display presents an animated graphical representation of the communication events in a parallel program. The numerous communication events in a parallel program necessitate filterings that reduce the complexity of the communication space. MPI communicators provide the means of separating the communication space since each communicator specifies a communication context for a given set of communication operations. Communications that happen within one communication context do not interfere with communications in a different context.

Viewing the communications as belonging to separate contexts led to their spatial separation by assigning one instance of the communication display to communications relevant to one communicator. The abstraction achieved has been extended to allow for filtering over the processes that participate in the communications of a particular communicator. Hence, each time a new instance of the display is opened the user is presented with a list of all the processes and a list of all the communicators in the program. From these lists, the user is then able to select any two sets of processes and communicators which define a (possibly empty) subset of communication events to be visualised in the newly created instance.

[15]This mechanism has not yet been implemented.

As the animation proceeds, the next communication which satisfies both a process set and a communicator gets displayed on the corresponding instance of the Communication display.

The animation of the many instances of the Communication display can be driven by the animation of the Navigation display or alternatively the user can step the animation. The latter allows the user to browse through the sequence of the communication events on all the instances of the Communication display. The necessary control interface is provided in a separate window called the communication browser. Step-by-step browsing is suitable for the close examination of the ordering of communication events in time. When the user steps the animation for the Communication display instances, the Navigation display is updated as well.

A list of single line textual descriptions of each communication event (Communication event description display) is also provided along with every instance of the Communication display. The textual description aims at:

- resolving any ambiguities that might be present in the graphical representation,
- providing timing information about the beginning and completion times of events and
- supplying the user with a history list of communication events, so that the current state of the display can be related to previous ones.

Participants in VISPAT's design meetings determined the graphical encoding scheme used for point-to-point operations is illustrated in Figure 7–8. The various components of the MPI Communication display are shown in Figure 7–9. Processes are represented as spheres and communication operations (sends and receives) as arrow heads attached to the peripheries of the spheres. For the purpose of visualisation, each point-to-point operation is assumed to have a source

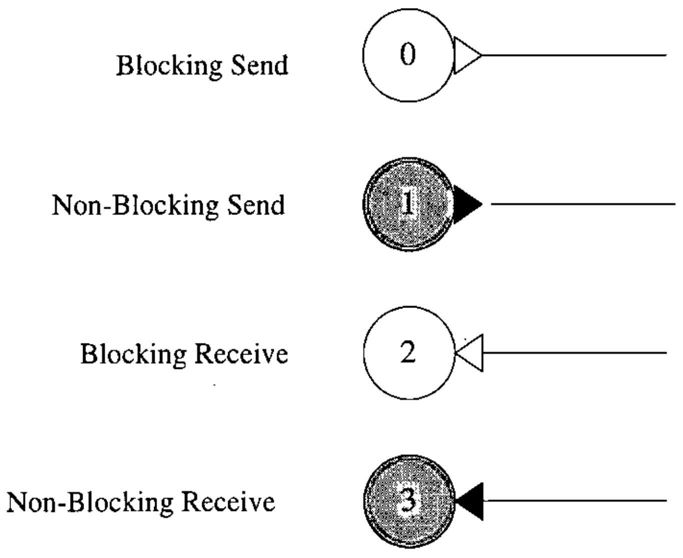


Figure 7–8: Graphical encoding of MPI point-to-point communications

and a destination which map onto spheres on the display. As long as any of the two processes corresponding to the source and destination spheres is engaged in communication with the other, a line is drawn between the spheres to identify them on the display. Additionally, spheres are coloured as either empty or filled to indicate whether the corresponding processes are currently being blocked by a blocking communication event or not. Collective operations are represented as symbolic labels drawn within spheres. For those collective operations which distinguish a particular process as being the root of the operation, the label in the sphere corresponding to the root process is coloured differently. Table 7–1 lists all the instrumented collective functions and their corresponding symbolic labels.

For animation, each instance of the display maintains information about the various processes and the set of communication events associated with them. At initialisation time, each instance obtains a copy of the single communication event list supplied by the TPE and which contains all the communication events recorded in the trace files. When the user finishes the selection of processes and communicators for this instance, TPE functions are called to mark the appropri-

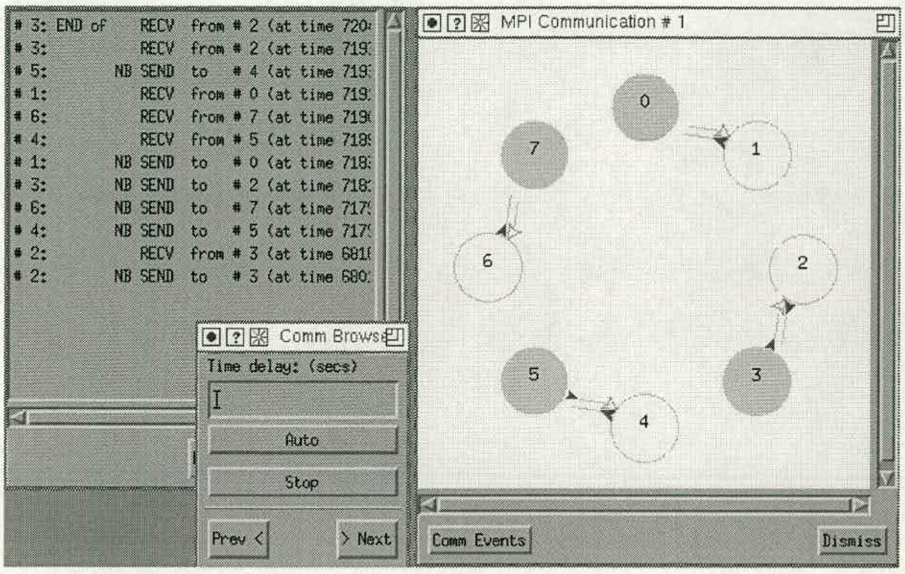
Barrier	X
Broadcast	B
Gather	G
GatherV	GV
AllGather	AG
AllGatherV	AGV
Scatter	S
ScatterV	SV
AllToAll	AA
AllToAllV	AAV
Reduce	R
AllReduce	AR
ReduceScatter	RS
Scan	SC

Table 7–1: MPI collective operations and corresponding labels

ate events in the list as selected. Animation skips unselected events and updates all the instances in such a way that all appear to be synchronised by a common clock whose value at each step equals the time stamp of the most recently displayed communication event.

7.5.3 Membership Matrix Display

The Membership Matrix display shows process memberships in the communicators of the program. Process numbers are displayed along the top of the matrix, whereas communicator names are listed vertically on the left hand side. If process j is a member of communicator i, then a circle is drawn in position (i,j) of the matrix. Within each circle, the rank of the corresponding process in that communicator is shown. Figure 7-10 shows an example of the membership matrix display. When the display is initialised, the list of all communicators in the program is traversed



```
# 3: END of     RECV  from # 2 (at time 720
# 3:            RECV  from # 2 (at time 719
# 5:         NB SEND  to   # 4 (at time 719
# 1:            RECV  from # 0 (at time 719
# 6:            RECV  from # 7 (at time 719
# 4:            RECV  from # 5 (at time 718
# 1:         NB SEND  to   # 0 (at time 718
# 3:         NB SEND  to   # 2 (at time 718
# 6:         NB SEND  to   # 7 (at time 717
# 4:         NB SEND  to   # 5 (at time 717
# 2:            RECV  from # 3 (at time 681
# 2:         NB SEND  to   # 3 (at time 680
```

Figure 7–9: Components of the Communication display

by the TPE functions in order to obtain all communicators of which each process is a member. These memberships are initially drawn as described above. This is the static representation of the memberships. Animation is performed by progressively filling in the circles with colour as the animation clock goes past each of the join events.

7.5.4 Statistics display

The Statistics display provides a list of metrics for each process. These metrics are communication specific quantities.

The list of metrics supported in the current version include amongst others:

- the time spent in collective and point to point communications,

- the number of messages sent and received by each process and

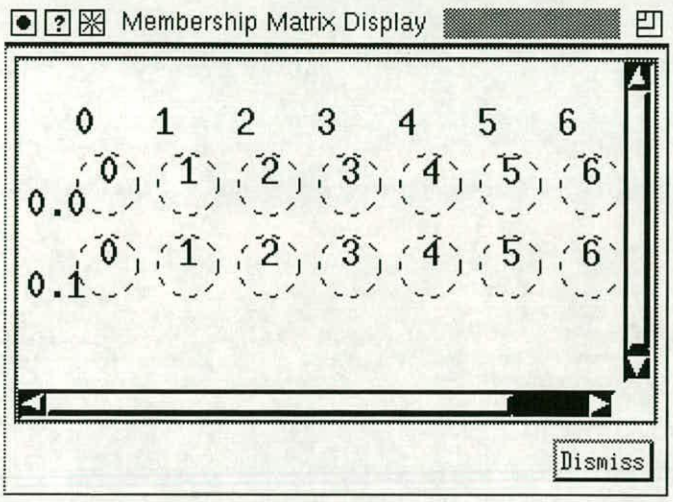


Figure 7–10: Membership matrix display

- the minimum, average and maximum times spent by each process on blocking communication and calls to MPI_Wait and MPI_Test functions.

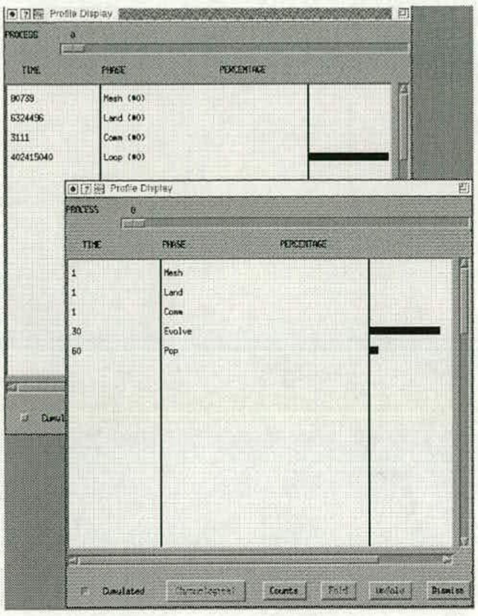
The above metrics may be used in performance analysis to reveal communication bottlenecks and can be presented graphically by using a graph plotting tool[16]. The graphical representation of the statistical information indicates at a glance which processes are engaged in heavy communication.

7.5.5 Profile display

The Profile display presents quantitative information about the various execution phases in a parallel program. The information provided is mainly concerned with the execution time of a particular instance of a phase in the program or the total time spent in all instances of a particular phase. In addition to absolute numeric values, a bar-graph scheme is also employed to clearly indicate the most time

[16]The current version of VISPAT uses Gnuplot.

consuming phases and facilitate comparisons between groups of different phases. Figure 7–11 shows two instances of the Profile display.



TIME	PHASE	PERCENTAGE
90739	Mesh (#0)	
6324496	Land (#0)	
3111	Comm (#0)	
402415040	Loop (#0)	

TIME	PHASE	PERCENTAGE
1	Mesh	
1	Land	
1	Comm	
30	Evolve	
60	Pop	

Figure 7–11: Two instances of the Profile display

7.6 An example of VISPAT use

A simple application program can be used to demonstrate some of the interaction features of VISPAT. The example program solves a simple predator-prey model. The two animal populations modelled are rabbits and foxes which live on a piece of land. The animal populations are represented by two two-dimensional arrays which are decomposed regularly across an array of processes. The program consists of a number of phases whose hierarchy is depicted in Figure 7-12. A number of initial phases – *Mesh, Land, Comm* – are responsible for defining the process topology, the geometric partitioning of the problem and the MPI data types for exchanging rows and columns of the arrays among nearest neighbour processes.

The next generation of foxes and rabbits are computed according to some pre-defined parameters of the model during a phase called *Evolve*. The population of one animal species within a stretch of land depends on the population of the other animal species in the same stretch of land.

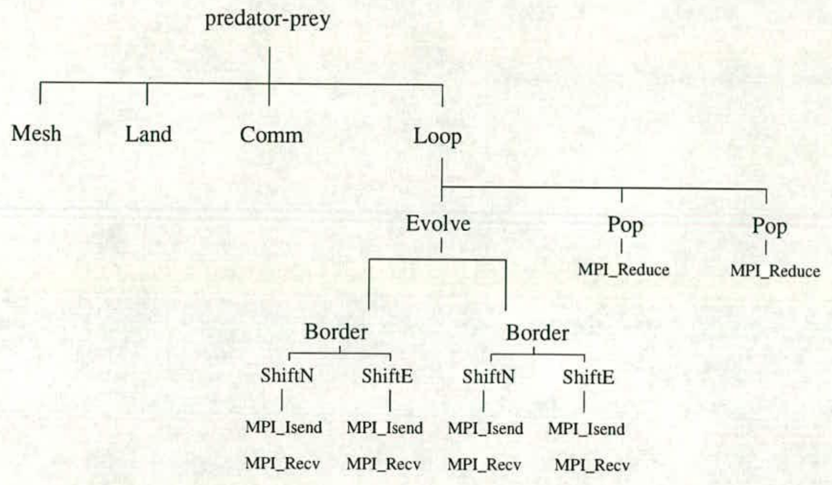


Figure 7–12: Unfolded phase hierarchy of predator-prey program.

Information about the land boundaries needs to be swapped among nearest neighbour processes. These communications are performed during the phase *Border* which takes place within the phase Evolve. The eight processes, which in the program are arranged in a Cartesian topology, exchange data to eastern, western, southern and northern neighbours. After the model is solved in each iteration, a phase called *Pop* summs up the population of the two animals across all processes.

Initially, the phase tree of the program looks like the one presented in Figure 7-13. Figure 7-14 shows the phase hierarchy after phases Mesh, Land and Comm have been *filtered out*.

In Figure 7-15, the *Loop* phase has been *unfolded* to its sub-phases; the Navigation display (Figure 7-16) shows some instances of the user-defined phases Evolve and Pop for the first four processes of an eight processes run. When phases Evolve

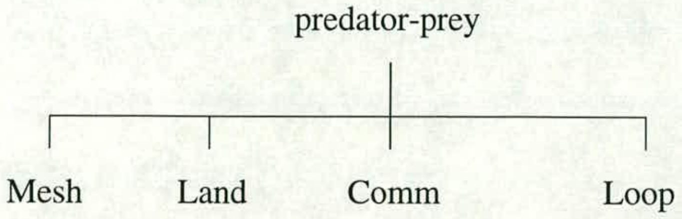


Figure 7–13: Top level phases of predator prey program.

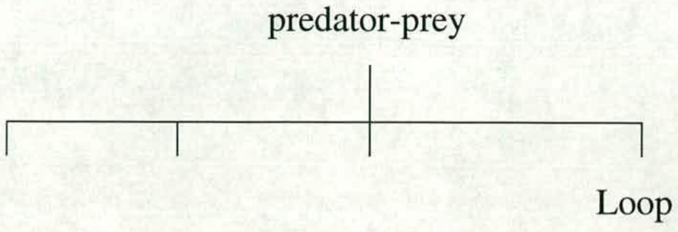


Figure 7–14: Phases Mesh, Land and Comm have been filtered out.

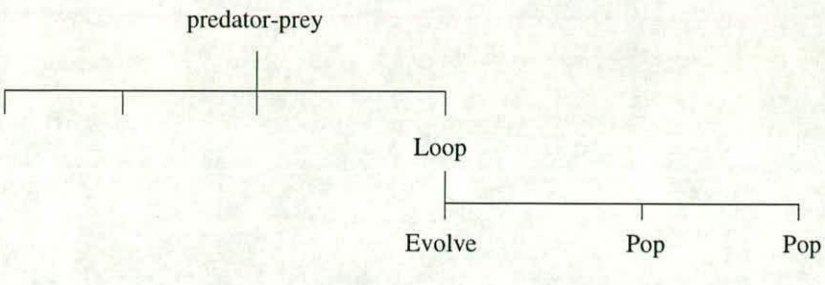


Figure 7–15: Phase Loop has been expanded to its subphases.

and Pop are expanded to their subphases, the phase hierarchy is the one shown in Figure 7-12.

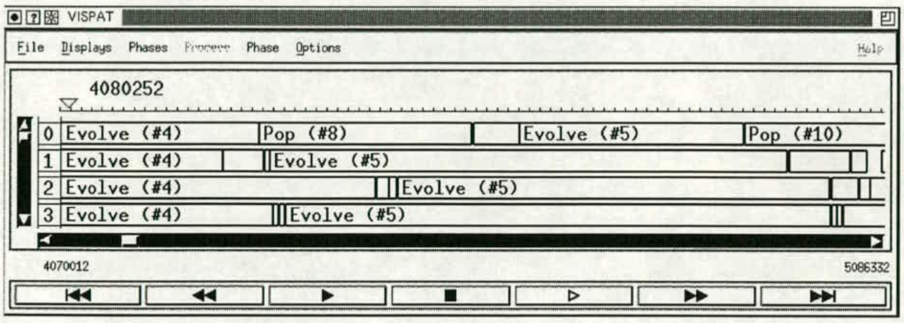


Figure 7–16: Phase hierarchy of predator-prey program.

A glance at the Profile display in Figure 7–11 reveals that Evolve and Pop are the most expensive phases in the program. Evolve's sub-phase Border (Figure 7–15), which exchanges the populations in the borders, and Pop, which calculates the total population at the end of each iteration, are called twice, once for each of the two kinds of animals. Foxes and rabbits live in the same piece of land, so the arrays that hold their populations have the same number of dimensions and shapes. It was thought that the phases Border and Pop could be called only once, halving the number of communications if both the populations were communicated in one message. In this second version, the creation of two new data-types for the communication of the data at the boundaries (in the first version a number of non-contiguous and contiguous data elements were organised in the form of columns and rows) was omitted. The data was packed before the communication and unpacked after the communication was completed in order to support the combined transfer of the fox and rabbit data. The corresponding phase tree is depicted in Figure 7–17.

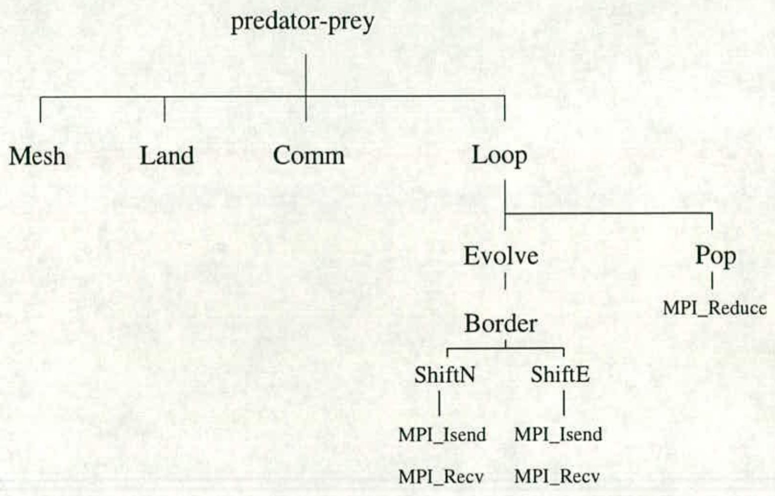


Figure 7–17: Phase hierarchy of predator-prey program (version two)

7.7 Conclusions

A prototype tuning tool has been implemented which through the principle of hierarchical phases, allows tuners to relate low-level events in trace data to program source code and reduces the amount of information they have to consider. The latter is achieved through an interface which enables the traversal of the phase hierarchy and the creation of different views of the program for which performance data is displayed. The hierarchical grouping of events in VISPAT reflects the hierarchical grouping of the library functions in the program and in this way provides the abstractions necessary to limit the focus to the areas of interest whilst maintaining a close link to the structure of the source code. Moreover, the performance data query mechanisms of TPE can be used to query the performance data and in conjunction with the source code reference that the concept of phases provides can assist in eliminating the impact of the cause/effect chasm which has been identified as one of the major sources of difficulties in tuning (Chapter Four).

The current implementation of VISPAT addresses only a small part of the user requirements and the ideas raised during design meetings and informal conversations with local tuners over the last three years. Further work will have to focus on:

- instrumenting more MPI functions,
- reducing the intrusiveness of the instrumentation,
- introducing more advanced data structures in order to speed up the trace analysis performed by TPE,
- enriching TPE with other filtering mechanisms,
- investigating ways in which to improve the scalability of the tool,
- and making VISPAT extensible by providing a well defined interface to the output of the TPE functions in order to drive other displays or tools.

Further development of VISPAT should be guided by constant feedback from tuners. As an example of the benefit that user involvement can incur, an informal evaluation of the second version of VISPAT was attempted. Some results of this evaluation are presented in Chapter Eight which describes user participation in the design process of VISPAT.

Chapter 8

The participative design process of VISPAT

User involvement in the design and evolution of computer systems has enjoyed growing interest within the human factors and the systems engineering communities in recent years. The term user-centred design has been used to refer to a number of methodologies and practices which aim to make the end-user participate actively in the design process of computer systems. Such techniques include: contextualist design [70]; joint application design; [29]; participative design [49]; cooperative requirements capture [108]; usability engineering; iterative design; and others.

Among the most widely established techniques, participative design is a term attributed to design approaches first employed in a pioneering project in Scandinavia, which succeeded in acquiring knowledge about the users' work practices by allowing users to participate in the system design process [19,18,146,100,49].

There are various definitions of participative design, but they all have in common a number of basic principles, found in [42]:

- **The focus is on the users' work.** The design of the system aims to improve the quality of the work life of the people who will use the system. This goal is shared by users and developers who take part in the design process. Technology design is only a secondary objective of the design process.

- **Design should not be done "for" or "by" users.** When design is done for the users, knowledge about the users' tasks may be missing from the design. On the other hand, design by the users excludes developers from the design group; most often, though, users lack the technical skills necessary for technology design . Clearly, design with the users implies a framework of collaboration and mutual learning, where designers are willing to learn about users' work practices and users are willing to become familiar with technology.

- **Design is iterative.** The design process should be based on tools and techniques simulating the system and its use under real work conditions as users should be able to evaluate and determine future enhancements.

Local tuners were invited to actively determine the design of VISPAT. This participative approach aimed at acquiring additional knowledge about the tuning task and at building a tuning tool, which would address to a large extent tuner requirements. The first part of this chapter describes how users participated in VISPAT's design. When a working prototype of VISPAT was developed user participation was carried on through an evaluation of the tool which in turn guided the design of a subsequent version of VISPAT. The second part of the chapter describes how VISPAT's evaluation was conducted and presents some of the evaluation results.

8.1 Participative design of VISPAT

environment using very limited resources. The volunteered to take part in the design meetings and were motivated by a genuine interest in performance analysis tools. No group member had any authority over any other member. People were not accountable to anybody for the meetings' progress and the project's success. They had not been taught any formal design methods. They had three to five years of experience in parallel program development. The initial meetings focused on requirements specification while the subsequent ones were concerned with design issues. The design meetings occurred within a period of three years with a version of the tool being the outcome of each year's meetings. During the initial meetings of the first year, the participants started from a very general description of the tool and the meetings were a mixture of requirements specification and high-level design. During the subsequent years' meetings, the design was altered. The number of participants ranged in size from three to eight and the meetings typically lasted approximately 90 minutes. The participants were sitting around the table with minimal tools such as pencils, paper mock-ups of the design and a drawing board. The paper mock-ups were necessary for the users to be able to envision the design and contribute to it. Paper prototyping fosters an atmosphere of exploration and experimentation[1]. Solutions and variations can be tried easily and quickly. The design sessions of VISPAT were recorded and transcribed.

8.1.1 First year meetings' overview

The design sessions started on February 19 in 1994 and were scheduled to be finished by June 26 of the same year, when the tool development would start. Figure 8–1 shows when the meetings took place. The initial intention of the participants was to have a meeting each week. Instead, the team met almost consistently during the initial period of the first seven weeks, but the frequency of meetings dropped after the sixth meeting.

[1]one of the many techniques which can help users acquire knowledge about the proposed system. Card games, computer prototypes [112], think-aloud experiments [82], working analysis walls [43], [169]. Users are offered an equal opportunity for participation since they are not intimidated by the use of computer technology. The design sessions can be video-recorded, so that design can be implementors.

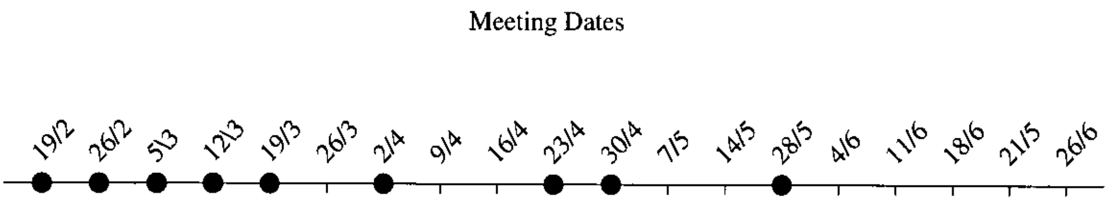


Figure 8–1: Meeting dates in a period of four months.

The first two meetings were brainstorming sessions, where people shared domain knowledge and expressed ideas, random requirements and alternatives. Many of these requirements were materialised in the form of paper mock-ups produced by the designer and discussed during the next four meetings. This second series of meetings elaborated on the requirements and were centred around the paper prototypes, which became quickly more and more detailed. A number of new functionalities and displays were introduced at the same time and people could criticise them. The discussion gradually focused on lower level design issues. The seventh meeting attempted to summarise the proposed displays and also tried to prioritise them. The eighth meeting had purely a management purpose. The activities of the people who would undertake the development of VISPAT were outlined and scheduled in time. The ninth meeting concentrated on low level user interface issues.

The first meeting focused almost entirely on identifying the knowledge and requirements necessary to lead to the design specification. Almost half of the meeting's twenty nine technical issues raised were concerned with articulating new requirements and half with new domain knowledge.

The second meeting overlapped significantly with the first. The discussion focused on old and new requirements and domain knowledge.

During the third meeting, twenty two issues were discussed of which three were concerned with meeting and project management issues. Six of the remaining issues were about design approaches, which addressed previously mentioned user requirements. Four issues were design approaches contributed by the designer.

Five issues revisited design approaches proposed previously in the current meeting with the purpose of altering it. Sometimes, a design approach would be partly inspired by a previously mentioned user requirement and partly devised by the designer. Three issues were concerned with identifying a new requirement. During the remaining issues, knowledge and requirements were articulated in the context of a design approach. These statistics indicate that during the third meeting attention shifted to design. People were very critical to the design solutions presented by the designer. Five issues discussed variations of the current design. In one case, the design solution offered by the designer was rejected and substituted by a new one proposed by one of the users.

During the fourth meeting, eight out of twenty one issues were concerned with design approaches targeting previously defined requirements. Four issues were variations of the current design and were presented by the designer. Participants tried to change several design alternatives proposed by the designer and discussed about them in the context of knowledge they had about the implementation of the PUL and CHIMP libraries. Four issues repeated some old knowledge and requirements which had been articulated in the past. This was considered necessary for decision making: old knowledge and requirements were used during the process of evaluating the proposed each time design solution.

During the fifth meeting, only one issue introduced and examined a new requirement. The rest of the issues introduced design options which attempted to address previously expressed requirements.

During the sixth meeting, nineteen issues were discussed of which one was concerned with project management. Only three issues dealt with identifying new requirements or knowledge. The remaining sixteen issues evaluated the design solutions presented in the mock-up. During the discussion about an issue, group members would occasionally share some knowledge which was relevant to the way people go about tuning or to the specifics of the PUL or CHIMP implementation. Again, these requirements or knowledge were oriented towards the context of the current each time design approach, in order to prove the appropriateness of the design solution in hand. Only once were previous design approaches questioned by participants and discussions were minor clarifications when the mockup did not successfully convey the right impression.

The seventh meeting tried to prioritise the displays so as to cut down the number of the displays that would have to be implemented. Two issues were concerned with altering a previous design implementation and three issues dealt with new requirements.

During the ninth meeting, a large number of issues attempted to introduce some minor changes and improvements to previous design approaches. Five issues were concerned with new requirements which were not elaborated further since it was thought they involved a great deal of implementation effort.

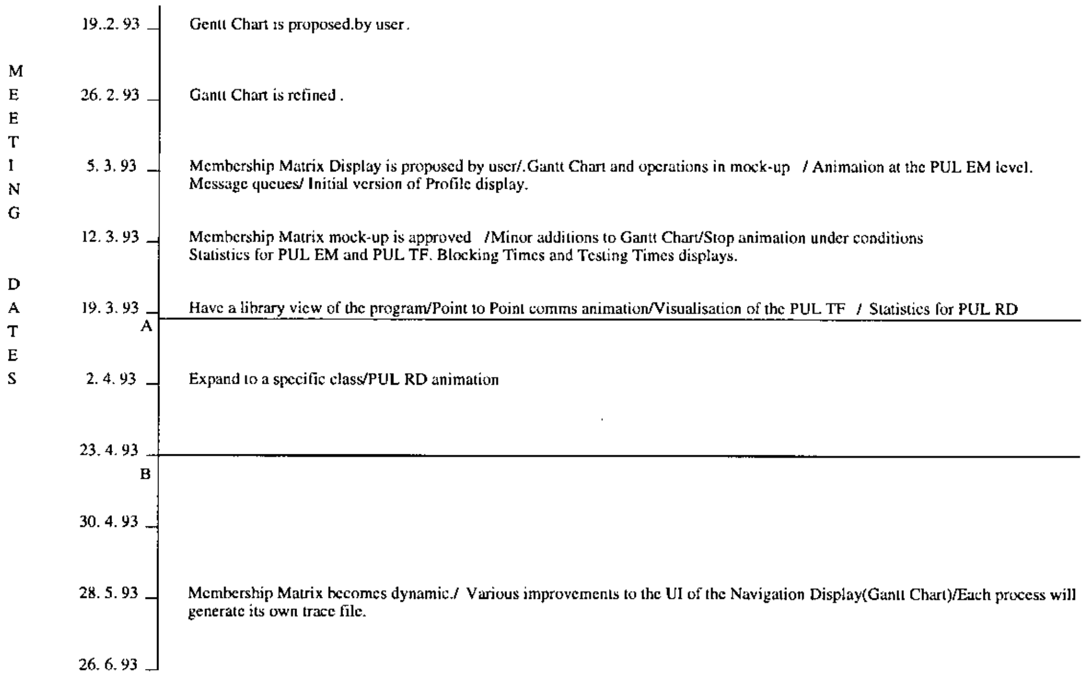

MEETING DATES
19..2. 93
Gentt Chart is proposed.by user.
26. 2. 93
Gantt Chart is refined .
5. 3. 93
Membership Matrix Display is proposed by user/.Gantt Chart and operations in mock-up / Animation at the PUL EM level. Message queues/ Initial version of Profile display.
12. 3. 93
Membership Matrix mock-up is approved /Minor additions to Gantt Chart/Stop animation under conditions Statistics for PUL EM and PUL TF. Blocking Times and Testing Times displays.
19. 3. 93
Have a library view of the program/Point to Point comms animation/Visualisation of the PUL TF / Statistics for PUL RD
A
2. 4. 93
Expand to a specific class/PUL RD animation
23. 4. 93
B
30. 4. 93
28. 5. 93
Membership Matrix becomes dynamic./ Various improvements to the UI of the Navigation Display(Gantt Chart)/Each process will generate its own trace file.
26. 6. 93

Figure 8–2: Basic tool functionalities.

Most of the tool's basic functionalities were determined during the third and fourth meetings as is shown in Figure 8–2. The 26th of June was the deadline by which the design specification of the tool should have been completed. The line marked as A in Figure 8-2 denotes the middle meeting, whereas the line marked as B represents the middle point in the period of four months that the project lasted.

8.1.2 Communicating the design

Participants were peers in the academic establishment they worked for. Generally, discussions tended not to be detailed due to the limited time people were able to spend in the meetings. Occasionally, a conversation would be very short or very long according to the spontaneous interest of the interlocutors. Discussions fell into a general pattern in that, first, the displays and functionalities of the tool would be established, and then the information requirements of these aspects would be considered. There were no formal meeting agenda and meeting minutes. Only the designer did any preparation before the meetings. The remaining members participated in the formulation of design options in an opportunistic way. The above described loose cooperation was the context of a number of remarkable phenomena such as the richness and the free expression of ideas; communication problems, e.g. misunderstandings; compromises; etc.

8.1.3 The designer's role

Among the observations of this study was that as users were given more control in determining the design, the designer assumed a more supportive and less dominating role:

At each meeting the designer would prepare and present to the team paper mock-ups which were based on previous design results. The paper prototype would organise and elaborate the ideas expressed in the previous meetings and would introduce some new issues. As it has been shown in [131] potential users are unlikely to co-operate in the development if they feel their views have not been taken into account. Providing a paper prototype, which included the participants' views, guaranteed that users would continue to participate as they felt that their participation had an impact on the system design and that progress was made. Also, it provided the framework for further discussions. The designer was solely responsible for the preparation of the mock-ups. In between two meetings, the designer had to transcribe the conversations, distinguish the different viewpoints and integrate them in a design solution. Many times the issues were interleaved with each other, or they were very implicitly introduced. Often, the discussion about an issue would not conclude to a decision, so the designer had to identify within argumentative conversations what were the winning alternatives and incorporate them in the mock-ups.

In cases where there was not a clear winning alternative within the discussion of an issue, the designer had to arbitrate and formulate the design solution herself. The design solution would be presented to the team at the next meeting in order to be approved by the group.

The designer determined the theme of a large part of the discussions by raising the largest number of issues during the nine design sessions. During the first two meetings, the designer would raise an issue in order to elicit information from the group. Later in the process, the meeting flow was determined by the paper mock-ups, which served as the agenda for each meeting. An issue raised by the designer would be either about a design solution corresponding to the group's requirements of a previous meeting or a new design solution devised by the designer. In the former case, the designer sought some feedback from the team about the degree to which the design addressed the members' requirements. In the latter case, the designer attempted to prompt participants to articulate new requirements.

During the subsequent years of the tool's development, the designer was the only person to participate in all design meetings and to supervise the tool's development. The designer acquired a deep knowledge of the application domain and the design process and this made her a valuable source of information for the subsequent development teams.

8.2 Evaluation of VISPAT

A major concern for the development of any system is whether it is acceptable i.e., whether the system satisfies most user requirements and needs. A model of system acceptability is outlined in (Figure 8–3) [128]. In particular, usability is concerned with how well users can employ the system's functionality. This can be summarised in a number of attributes such as:

- Learnability: the system should be easy to learn to use.
- Efficiency: the system should be efficient to use so that once the user has learned how to use it, a high level of productivity can be demonstrated.
- Memorability: casual users should be able to return to the system after some period of time without having to learn everything from the beginning.
- Errors: the system should not allow users to make errors.
- Satisfaction: the system should be pleasant to use.

VISPAT was evaluated co-operatively with the help of nine users. The aim of VISPAT's usability evaluation was twofold: first, to discover major usability problems; and second, to give local users – who did not participate in the tool's design stage – an opportunity to voice requirements which could be taken into account

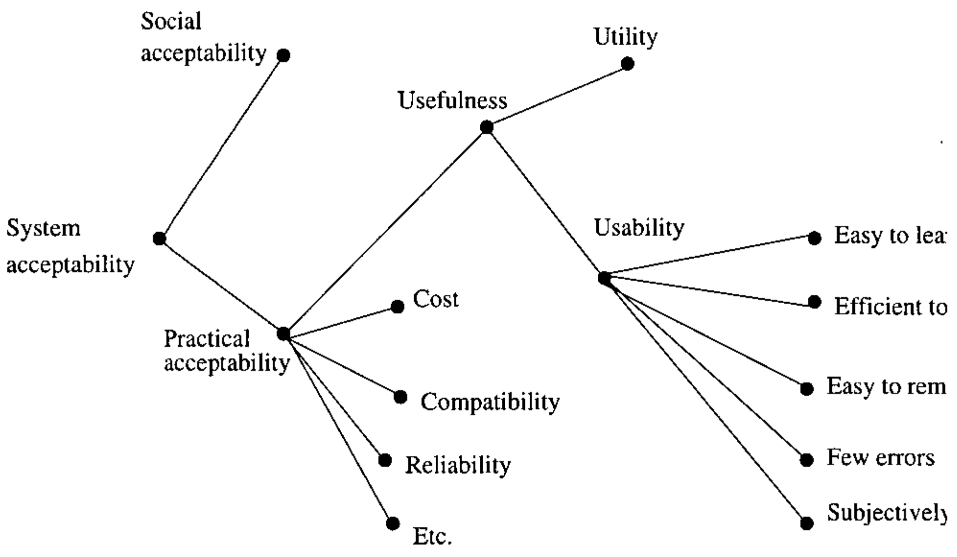


Figure 8–3: A model of system acceptability

in subsequent iterations of the design process. The evaluation was particularly concerned with guessability and the information needs of users [84].

The reason for setting the focus of the evaluation to guessability was that the system's main functionalities had already been evaluated through the previous year's iterative design process. On the other hand, the observational studies of the use of performance tools reported in Chapters Four and Five indicated that most tuners do not consult any documentation so it is vital to make VISPAT's functionality and operation self-evident. The approach to evaluation employed was based upon the Coaching Method [109]. The Coaching Method is a simplified thinking aloud study which can be used in *discount usability* evaluations[2]. While

[2]Nielsen's *discount usability testing* is based on the use of scenarios, simplified thinking aloud studies and heuristic evaluation [126,127,129]. A scenario combines the limitations of a vertical prototype (the user cannot access all aspects of the system) and of a horizontal prototype (the user cannot interact with real data). Nielsen gives a taxonomy of the various forms of scenarios and reports on the results of a comparative study of

most of the think aloud methods attempt to interfere as little as possible with users, the Coaching Method tries to direct the user while using the system and the experimenter is allowed to answer any of the user's questions. Coaching is aimed at discovering users' information needs in order to make the user interface self-explanatory and improve the quality of training and documentation. By giving users the information which they need at some point of their interaction with the system, they rapidly become expert users and are able to judge the system both in terms of its functionality and its usability.

VISPAT's evaluation was based on a vertical prototype. Some aspects of the system could not be used because they had not been implemented yet. The nine evaluators who participated in the study had a moderate experience with parallel programming and had not used VISPAT before; all but one did not take part in VISPAT's design sessions. All had used some message passing programming environment before and most of them were already exposed to programming with MPI. A toy MPI program was used in the evaluation sessions apart from two cases where the evaluators requested to use their own program with the tool. The sessions were videotaped.

The evaluation revealed a number of problems and requirements which were recorded to inform the design of future versions of VISPAT.

8.2.1 Quantitative and qualitative findings

The choice of usability measurements for VISPAT was oriented towards one of the aims of the evaluation which was to discover how guessable the tool's user interface was. For each of the thirteen tasks, it was measured how many evaluators

the effectiveness of paper versus computer based mockups in heuristic evaluation [127]. A collection of fundamental usability heuristics can be found in [128].

performed a task correctly without any help from the experimenter; by using an alternative strategy; with some assistance from the experimenter; or were helped so much that the task was actually carried out by the experimenter. Additionally, for each user interface feature which the evaluators had to use while performing a task, it was recorded how, if at all, this feature was used by the evaluators. Table 8-1 lists the possible ways an evaluator could interact with a user interface feature.

uc	used feature correctly
uw	used feature wrongly
nt	did not think to use feature at all
hu	helped to use feature
eu	experimented with using feature
gu	guessed correctly

Table 8–1: Ways of interacting with the interface features of the tool.

ne	negative comment
po	positive comment
un	unexpected user action
re	requirement
or	old requirement

Table 8–2: Different kinds of utterances.

Apart from these measurements, the evaluation was concerned with revealing new user requirements. Occasionally, requirements would arise after the completion of a task, when the evaluators were asked to express their opinion about specific aspects of the tool revealed during the task. At the end of the session, the evaluators were asked to evaluate the tool in general and this gave them the chance to utter any additional general requirements (re) or old requirements which had been expressed in the design meetings but not implemented (or). The videotapes were examined for unexpected evaluator actions (un) as it was thought that they could hint areas where the interface and the functionality could be improved. The evaluators' comments, which were positive (po) or negative (ne) about the system, were recorded. Table 8-2 lists the coding of the different kinds of utterances recorded during the evaluation sessions. Appendix D describes in detail the evaluators' utterances. The following subsections correspond to evaluation results from a subset of the tasks users were asked to performed during the evaluation sessions.

8.2.2 Evaluation tasks

Task 1

The evaluators were asked to replay the program execution and guess the function of the buttons (Figure 8–4) which control the animation of the Navigation display in a way similar to the control panel of a sound system (hi-fi metaphor).

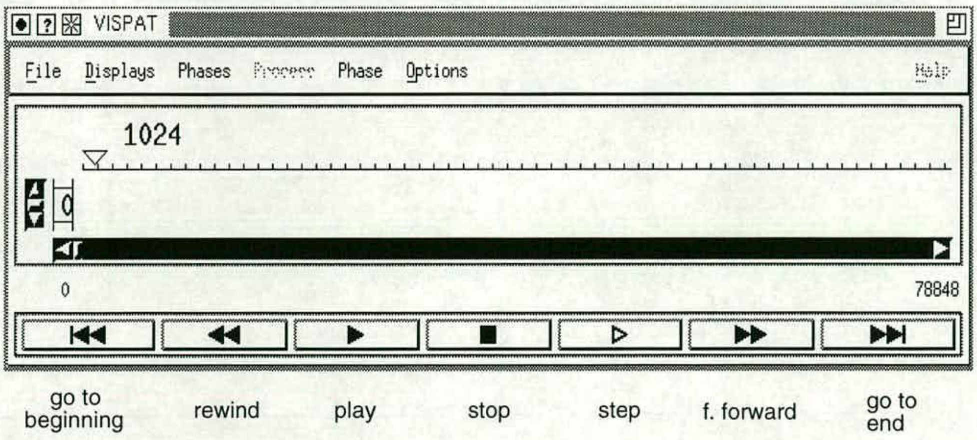


Figure 8–4: Animation control buttons

Two evaluators were not able to guess the function of the *go to the beginning* button while four evaluators failed to guess the meaning of the *step* button. All evaluators were able to make the tool start the animation. Table D–3 shows the contributions of the evaluators in terms of negative and positive comments and also in terms of additional requirements. It also lists the ways in which evaluators interacted with the hi-fi buttons and depicts how many evaluators managed to perform Task 1 on their own.

There were instances where the tool did not give enough feedback to the evaluators. In one case, animation was too slow for evaluator 4 to realise that animation time was indeed advancing. For a while, evaluator 4 could see nothing on the screen because of the very small initial value set to the *time unit* field. He said: "I am surprised. I was expecting things to happen. Instead, I have an empty display." Evaluator 4 was not helped by the continuously changing time indications in the bottom left and right corners of the Navigation display. Instead, he asked the experimenter how he could understand that animation proceeded slowly. The slow feedback of the tool was the reason that two evaluators commented on the difficulty to handle the display's horizontal scrollbar. Upon each movement of the scrollbar, the tool re-calculated the new animation time and drew the corresponding phases in the Navigation display. Re-calculating and drawing took considerable time while the evaluators perceived the delay as an inefficiency of the system and retried to move the scrollbar which increased further the delay.

Half of the evaluator requirements during this task were already implemented or had been identified during the design of VISPAT.

Task 3

Evaluators were asked to replay the execution at the second level of expansion. As is shown in Table D–4, only one evaluator needed assistance in order to select the *unfold* option from the *Phases* menu. After the unfold option was selected, a message box appeared asking: "There are no selected phases, do you wish to unfold them all?". The evaluators had at this point two options: either to press OK and to proceed with the operation or to press CANCEL. In three cases, this message caused some confusion. Evaluator 6 was put off for a while before pressing OK and evaluator 2 pressed CANCEL and required to know how he could select all the phases at once. Evaluator 8 cancelled the operation and tried to select some phases first. As it was expressed the message implied that some irreversible action may happen when the user unfolds all the phases.

Six evaluators were confused when the phase expansion resulted in an empty navigation display as most of the MPI phases took place only after a number of time frames. For example, evaluator 9 said: "I lost the phases". Evaluator 6 thought that the animation had finished and that the program execution could be replayed only once. Evaluator 2 expressed his satisfaction when he realised which phases in the program were responsible for the large execution time[3]. Evaluator 4 said that he liked the filtering mechanism with which only selected phases can be unfolded.

Among the most important requirements expressed during this task were: the requirement to see the source code; the requirement that colour should be used to identify those phases whose duration is too small to allow for their names to be displayed; and the requirement to be able to navigate to arbitrary points in the animation time.

Task 7

In order to perform task 7, the evaluators had to make the Navigation display depict all the user defined phases in the application program. The toy program did not contain any user defined phases, so when the evaluators invoked the filtering mechanism to perform the task, the Navigation display did not present any user defined phases. Five evaluators were confused by this (Table D–5). Evaluator 7 thought that the tool was still processing the user event and found the response of the tool too slow. Evaluators 7 and 9 did not like having an empty frame on the display. Evaluator 6 considered going one level up in order to find the user

[3]Evaluator 2 was using his own program during the evaluation.

defined phases. One of the two evaluators, who used their own programs, knew that his program did not contain any user defined phases, so the response of the tool did not surprise him. Evaluator 3 thought that the filter operation, which he invoked, had actually the opposite semantics from what he initially thought it had, so instead of *filtering in* the user defined phases, he thought that it *filter* them *out*. For a while, evaluator 2 thought that *filter* meant filter out.

Although users' knowledge of their own programs may seem to alleviate the need for a message in this case, the structure of a typical parallel program may contain a complex hierarchy of phases for which it is vital to provide some feedback on the user navigation actions. Feedback on navigation actions becomes more important in cases where parallel programs are tuned by experts who are not the programs' authors.

Task 9

In order to perform task 9, the evaluators had to set the animation time to zero and use the "Next" button on the Communication browser window in order to make the Communication display depict the third communication event (Table D-6). Finally, the evaluators had to invoke the Communication event description display and see the start time of the third communication event.

The Communication event description display which accompanies the Communication display lists the communication events as they happen over time. Eight evaluators found very confusing the fact that the event list scrolled downwards instead of upwards and were not able to complete the task without the help of the experimenter. Four evaluators made critical remarks about the order of the text presentation. The decision to adopt a downwards scrolling was influenced by the way the communication events are handled in order to construct the system's data structures. It was proved, though, that downwards scrolling contrasted with the traditional way humans read (from top to bottom). One evaluator noticed that having known which way the list goes it was easier for him to use it. This is because the most current and subsequently the most important event (from the tuner's point of view) is depicted at the top of the display. This design decision may have an increased guessability and learnability overhead but it can be more efficiently used once the user learns how to use it. Three evaluators explicitly said that they liked the Communication event description display. One evaluator liked the mechanism of filtering communications according to which communication context they belong.

8.3 Summary and conclusions

Local users have actively determined the design of VISPAT through a series of design meetings which occured within a period of three years. Users were highly interested in critiquing constructively and reformulating when necessary the basic design options offered by the designer. Very early in the meetings of the first year the design was determined and after that participants did not examine any new major requirement. This can be attributed to the limited resources available to the project. During the meetings of the subsequent years the design changed to address issues associated with the change in the programming environment.

Whereas the series of participative design meetings constituted a formative evaluation of VISPAT, VISPAT was also evaluated summatively with the help of nine evaluators. The evaluation revealed many usability errors which violated a number of usability heuristics such as: providing adequate feedback; natural and simple dialogue design; good error messages; and speaking the users' language.

A large number of requirements (163/92) have been expressed during the evaluation. More than half of these requirements were "old" requirements, that is they had been considered at some time during the design of the tool. Most of the remaining requirements represent valuable contributions which could be realised in future versions of the tool.

The ratio of positive versus critical comments was 98/106. This along with the large number of requirements suggest that there is still a lot of work to be done to improve the usability of the tool.

8.3.1 Future plans for VISPAT

VISPAT is registered software with Fujitsu and has been installed on the Fujitsu AP1000 system in Kawasaki, Japan.

Two years ago, VISPAT was used by researchers at ANU (Canberra) to demonstrate their MPI implementation for Fujitsu AP systems at a Fujitsu workshop hosted by them.

The software remains installed for use on the EPCC cluster. VISPAT will most probably be included in a new release of CHIMP/MPI scheduled for this year (1997).

8.3.2 Summary

This chapter demonstrated how the principles of user-centred design were applied in the case of designing a parallel program performance analysis tool. User participation throughout the whole development cycle of VISPAT created a large repository of ideas and tuner requirements. It is hoped that these requirements will be addressed either in future versions of VISPAT or will be taken into account by designers of tuning tools similar to VISPAT.

Chapter 9

The Tuner's Workbench

The results from the observational study of tuning (Chapters Five and Six) can be used to drive a preliminary specification of a tuning environment, the **T**uner's **W**ork**B**ench (TWB), that could address support for tuning in the large. The requirements for such a system fall into a number of categories:

Ease of use. Ethnographic studies of configuration management and documentation practices report that documentation processes if regimentally adhered to are likely to slow down work processes, threaten people's control over their work practices and expose them more openly to accountability [38]. Other studies report that pen and paper technology are preferred during critical projects over electronic groupware systems for software development [170]. It is, therefore, important that the Tuner's Workbench should be easy to use and not overly constrain the ways in which tuners document the task. In this way, documentation will be less likely to be considered as an additional overhead. The system should fit into tuners' normal working practices and should not impose undue constraints. Documentation should be contributed and retrieved easily and quickly.

As tuning is performed interleaved with parallel program development it is necessary that the Tuner's Workbench[1] shall not be simply another level of software and will not introduce another degree of complexity to the process of producing parallel software. Rather, TWB should be driven by the software development process and requirements. It should be usable in a range of different development contexts and parallel programs from the complex, large scientific codes which are developed by a group of people, to the toy parallel codes of the novice programmer.

Policy-free. TWB should be "policy-free" in two important ways: it should not force the use of specific tuning tools or impose any particular strategy for performance analysis upon the tuner. Rather, it should be intended to provide an environment in which the tuner's choice of tools can be used more effectively in support of whatever strategy the tuner feels is most appropriate to the particular problem.

Support distinct aspects of tuning in the large. TWB should support the management of tuning by-products, the assessment of tuning progress and the sharing of knowledge within a cooperative development environment.

Some of the proposed functionalities of TWB can be found in many Configuration Management (CM) tools and team programming environments [53,98,139, 149]. Use of CM tools is not yet widespread and most users employ local mechanisms provided with the operating system because of their ease of use and the low software cost [37,46]. These mechanisms are also appealing because programmers do not have to change to more elaborate work practices. Most CM tools, however, have general functionalities and do not address tuning in any particular way. A notable exception is the configuration management tool described in

[1]Herein after referred to as TWB.

[31]. Programs whose performance is being tuned may have components which differ slightly, thus forming a family of instances for the specific component of the program. Instead of maintaining several instances of the program, each corresponding to the different component, this system maintains one instance of the program containing links to the component family. An important contribution of that work is that it addresses the requirement for tracking the changes of the operating system and the compiler, which is a major issue in parallel program development. However, the attempt to address tuning is incomplete. The granularity of tuning changes may be even smaller than that of the procedural level. Two program versions may differ from as little as in the ordering of the indices of a number of loops or as much as in re-organising whole parts of the code. Also, the implications of collaborative tuning are not examined. For example, in large programming teams, a mechanism is needed to check whether the development efforts of group members do not worsen the performance of the main code when their individual modules are integrated with the main version. Moreover, it does not cater for recording any tuning by-products or the rationale of design decisions.

The repetitive nature of tuning is taken into account in [24] by adopting a software engineering approach for trace file and performance metrics management. This work does not consider, however, tuners' work practices, nor does it examine how such a system should fit into the software development process. In contrast, the requirements for TWB are based upon the results of the observational studies of tuning described in Chapters Five and Six.

9.1 Preliminary specification of TWB

According to the results of the investigation of tuning in Chapters Five and Six, TWB should address tuning management and tuning knowledge sharing (Figure 9-1).

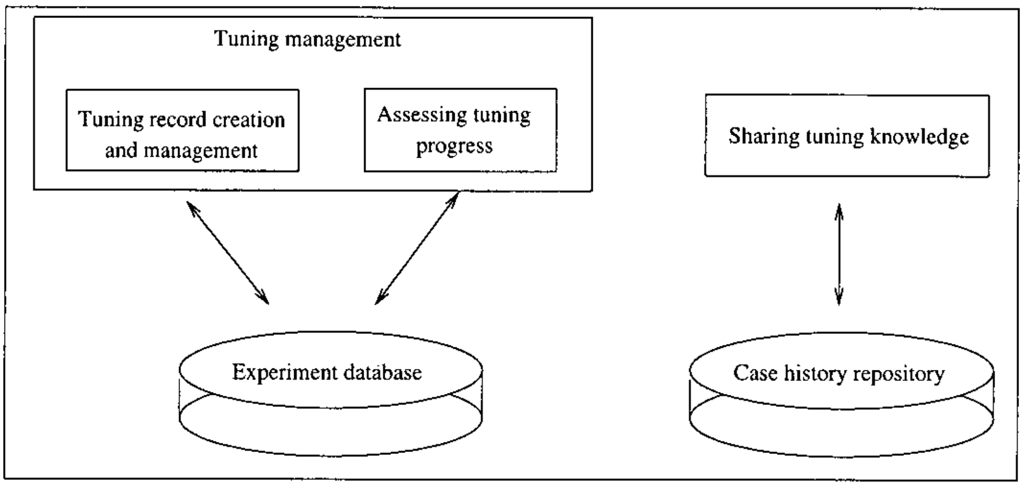


Figure 9–1: Components of TWB

An *experiment*, the main entity in the experiment database, is a record that represents a single unit tuning cycle. It includes a description of: why it was done (the hypothesis); what was done (the set of parameter-value pairs); and the results (trace data and set of performance metric-value pairs). Thus, the record of an experiment conveys both the rationale for a code design decision and the impact the decision has on program performance. According to the investigation of tuning in Chapter Four, the two major difficulties associated with the task are the difficulty to find the cause of a performance problem (cause/effect chasm) and the difficulty to know a lot of details about how the machine operates (faulty assumption/model). An experiment record can contain information about the cause of a specific program behaviour and its effect. At the same time, it can demonstrate the relationships among performance determining parameters, helping the tuner to establish correct models of the program behaviour on a particular parallel machine.

The implementation of TWB could be based upon existing hypertext, and/or database technologies. The experiment record should contain information in a variety of formats, including *textual* (e.g. lists of variable-value pairs representing parameters and performance metrics), *static graphical* (e.g. snapshots of displays generated by trace data visualisation tools) and *dynamic graphical* (e.g. animated displays from trace visualisation tools).

As program optimisation effort continues a number of experiments can be generated which should be stored to provide a record of the program tuning process. The structure of the tuning experiment repository could be hierarchical. Experiment records could be simply nodes (documents) within a larger hypertext document that would constitute the complete database. A sequence of experiment documents should define the path followed to tune a program and they could be organised as a hierarchical tree structure with branches that would correspond to experiments that would share a common hypothesis. For any hypothesis, the most recent experiment could be located at the leaf of the branch corresponding to this hypothesis.

The experiment documents which can result from documenting tuning should be handled to cater for tuning tasks documented in Chapters Five and Six (see Figure 9-2).

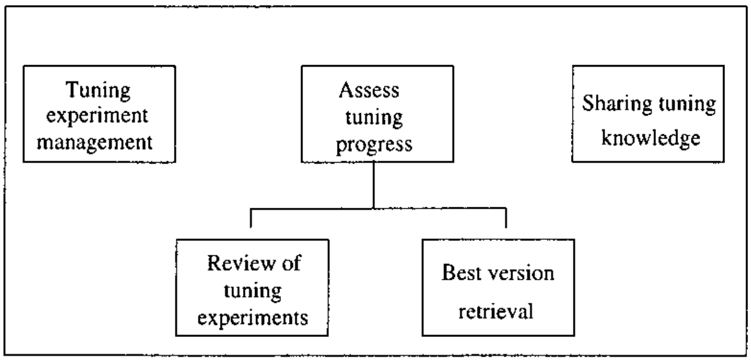


Figure 9–2: Tuning in the large

9.1.1 Tuning record creation and management

TWB's functionality should provide for the creation and management of tuning experiment records. In order to meet the requirement for ease of use, tuning record creation should be integrated with the process of program development and tuning by providing a common interface (through the experiment document) with tools for program development, performance assessment and tuning record creation and management. The "policy free" requirement should be met by the ability to use any tools which can support these activities (Figure 9–3).

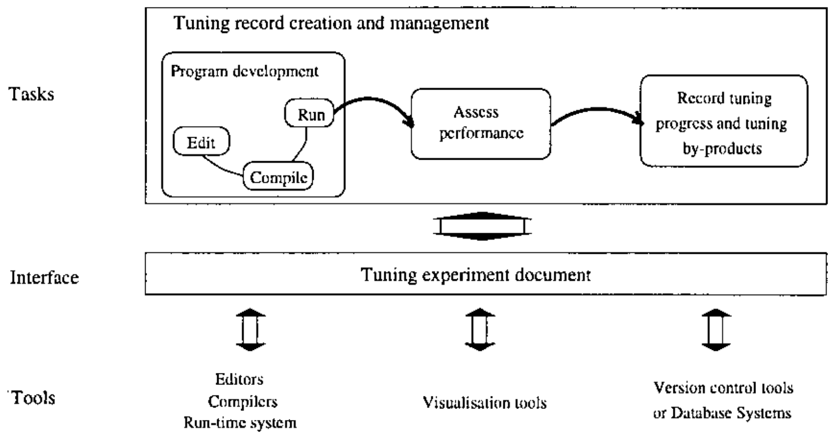

Tuning record creation and management
Tasks
Program development
Run
Edit
Compile
Assess performance
Record tuning progress and tuning by-products
Interface
Tuning experiment document
Tools
Editors
Compilers
Run-time system
Visualisation tools
Version control tools or Database Systems

Figure 9–3: Tuning record creation and management

Figure 9-4 sketches the practices associated with a unit tuning cycle[2] as they were identified by the investigation described in Chapter Five. After editing, compiling and running the code, the tuner may assess its performance based on timing results provided by a performance analysis tool or by using timer functions at appropriate points in the program execution. The program performance may be compared to previous results. After assessing the performance there may be a

[2]The notation employed in this figure is borrowed from Hierarchical Task Analysis [4].

number of possibilities as to what the tuner's next action may be. For instance, the scalability of the performance determining changes may be tried by running the code on different sets of processors. The tuner may need to describe important aspects of the particular run, the values of performance determining parameters and the execution times of the program. If the performance is still not what the tuner expected a few more changes may be attempted. In case the required changes are too drastic the tuner may decide to work with a separate version of the source code files. Alternatively, the tuner may decide to investigate the relations amongst performance determining parameters by a few more runs which may differ by one or more factors such as the structure of the input data.

The scenario described above should be addressed by functionality attached to the experiment document:

Edit - Compile - Run cycle.

Program development tools could be accessed through the experiment document.

Performance assessment should be facilitated by calling the appropriate performance tuning tool.

Evidence of program performance could be gathered by capturing images from the performance analysis tool and attaching them to the experiment document. This evidence could later be used in report generation or could be communicated to colleagues.

Textual information corresponding to hypothesis information should be held along with information representing an account of results which convey the performance of the program. Results from runs on different sets of processors could be added to the document and depicted graphically at a later point in time. The documentation functionality should be designed to correspond as much as possible to tuners' own documentation practices.

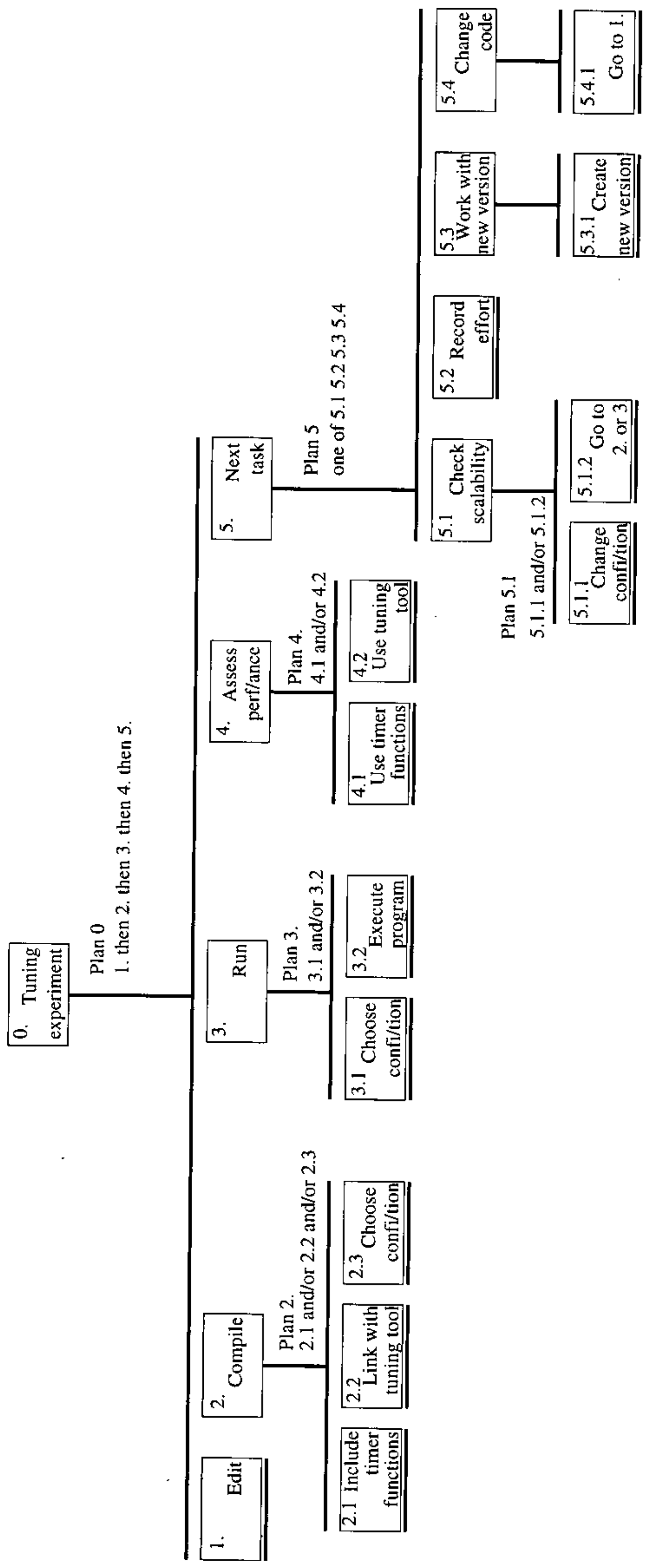


Figure 9–4: Tuning experiment (as HTA)

Tuning experiment creation.

The tuner should have the flexibility to determine the granularity of the tuning experiment. For example, a tuning experiment may differ from another by as little as the number of processors on which the program has been run or as much as the program's numerical algorithm.

The source files and tuning by-products (textual and graphical information) should be registered with every instance of a performance experiment. Version control could be supported by providing access to a suitable version control tool e.g., RCS [163,164]. In the process of creating a new experiment document the tuner could specify an existing experiment on which the new experiment could be based. The system could extract all the files associated with the particular instance of the experiment; the tuner could then continue the program development and finally create a new experiment record.

Multi-people program development should be facilitated.

Tuning experiments should be grouped into projects and operations should be supported to create and access documented tuning projects. The tool should support the documentation and communication needs of groups of people involved in parallel program development.

9.1.2 Assessing tuning progress

Operations to facilitate tuning progress assessment should allow for quickly browsing through the experiments. Within a hierarchical structure of experiment documents, each experiment could be linked to its predecessor and successor nodes and these could be immediately accessed by the activation of the appropriate links. The most successful experiment(s) could be retrieved. Searches could be narrowed by specifying one or more performance metrics and/or parameters.

Alternatively, the experiment structure could be directly queried. Users could retrieve the experiment having the lowest or highest parameter or metric value of interest; or they could retrieve a listing of values of a certain parameter or metric over a range of tuning experiment records.

9.1.3 Sharing tuning knowledge

Chapter Five and Six demonstrate that tuners' documentation efforts are motivated by the need to communicate ideas and design decisions in a collaborative work context. TWB should attempt to provide tuning documentation and training for less experienced tuners and a framework within which tuners can share their tuning knowledge and experiences.

A *case history* document is an extension of the concept of the tuning experiment document in that it represents a prototypical and interesting tuning problem made available for other people to see. These problems may not need complete application programs to be demonstrated but could instead be incorporated in small experimental codes. The technique to test alternative solutions in small pieces of code has already been identified in Chapter Five [Bel53]. The case history repository should be organised and accessed by the types of performance problems and the types of parallel machines, programming environments and tools used. Once a case history document has been retrieved, the tuning project could be reviewed by examining its trace files with the appropriate tools.

The availability of tuning knowledge of this type can be extremely useful, not only to others within a particular organisation, but to the parallel software development community at large. In this way, tuners could benefit from others' experience and save time and effort when faced with a similar tuning problem. The investigation of tuning (Chapter Six) has already shown the advantages of an on-line tuning repository. It is expected that such a system could improve the effectiveness of informal knowledge dissemination and electronic mail technical assistance. As the investigation of tuning in the small has shown, tuning knowledge is often organised in the form of heuristics or cliches. A repository of such cliches based on a set of case histories could help novice tuners who lack tuning strategies. Another positive outcome of such an organisation of information relevant to tuning problems is that the impact of cause/effect chasm identified in Chapter Four can be reduced if tuners record cases of exceptional tuning problems for other tuners to consult. Similar practices are already commonplace. For example, tuners routinely use Internet news bulletin boards to seek out advice from others who have solved similar problems.

9.2 Conclusions and future work

A specification of a tuning environment, TWB, has been proposed which demonstrates how some of the considerations raised by the observational study of tuning could be addressed. As was identified in Chapter Four, one of the major difficulties in tuning stems from tuners performing tuning without having a correct model of the system software or hardware parameters which can affect program performance (faulty assumption/model). As a result, experimentation is employed to try to establish such models and relations among performance determining parameters. The repetitive nature of this experimentation has led to a number of problems which were documented in Chapters Five and Six e.g., program versions proliferate and become difficult to manage; tuners have to keep track of the changes and tuning progress; and documentation and communication needs are increased when tuning is performed in collaboration with others e.g., results and program changes have to be communicated to colleagues. TWB should address some of the above mentioned problems by providing support for:

- managing the large number of program versions that the repetition of tuning produces,

- associating each unit tuning cycle with the concept of a tuning experiment and accompanying it with documentation of the experiment hypothesis and result i.e., the rationale for the performance determining changes and their outcome,
- automating the experimentation process and correlating its results,
- accessing information about the design process of the parallel code e.g., the reasons that lead to design decisions,
- sharing this information within group-based development projects and
- sharing tuning knowledge and experiences resulting from many projects with other people within and across organisations.

Two issues were regarded as particularly important in the specification of TWB. First, the system should not impede tuners' work practices. The requirement for such a tool already exists as parallel programmers spend considerable effort and time in devising techniques to document and manage the tuning process. The system should facilitate some of the documentation and experimentation practices described by the investigation of tuning in Chapters Five and Six. Second, the system should be designed to be independent of implementation details. The storage of experiments could be implemented to be file system dependent. Alternatively, any Database Management System or any hypertext tool could be used to store and organise experiment information. In the same way, any version control software could be used to support management of tuning by-products. Finally, tuners should be able to choose from a number of available visualisation tools to use with their programs. In this way, TWB can integrate functionality for addressing both tuning in the small and tuning in the large: it can include a number of performance visualisation tools which tuners can use to bridge the cause and effect of poor performance (cause/effect chasm); it can provide support for tuning documentation and management, thus helping tuners to establish correct models of relations between performance determining parameters through efficient, controlled and well documented experimentation; and finally, it can assist with tuning knowledge dissemination.

The implementation of TWB should adopt the tool development model and processes proposed in Chapter Four. Prospective users should be involved in determining the design of the tool. Further observation of tuners' practices should attempt to identify patterns of collaboration in managing and documenting tuning in the context of large programming teams [150]. The technology to support collaborative processes already exists. A number of systems have been proposed which address issues such as the different kind of collaborations that may exist within software development projects, for example, collaboration may be loose or tight, within the group and/or among different groups [76,148]. Capturing the rationale for design decisions has been investigated by many researchers [30,33,34, 111,106,110]. In [106], a system was described for providing rationale for maintenance by implementing a hierarchical annotation mechanism for managing the changes to a piece of code of varying granularity. In [33,34], a hypertext tool was proposed to capture design policies and discussions. Such a method is useful as a means to capture organisational memory, but the system proposed was not actually integrated with the actual artifacts. Richer and more flexible protocols for capturing software development processes were presented in [10,87]. A closer study of these environments combined with input from more observational studies of tuning can be used as the basis for developing a tool to document the design process along with the design artifact.

A possible extension of the concept of sharing one's tuning experiences with others is the idea of using tuning knowledge to provide technical assistance to tuners. The findings of the observation of the user support service, described in Chapter Six, showed that a large number of user questions are relevant to the same issues to the degree that performance experts could reuse some of the answers. Tuning knowledge can be organised in a hierarchical way according to the categories of problems, machines, languages and others[3]. A very efficient search mechanism can be provided which can direct users to the advice suitable for their problem. A successful example of Web-based technical assistance can be found in [83,?].

Certainly, there is need to examine how an environment such as TWB could be used by tuners. Studies have shown that when new technology is implemented and adopted without a concurrent examination of how processes and coordination might change and evolve, it has little impact on enhancing the productivity of the users [28]. TWB's design process should be iterative consisting of cycles of evaluation and design phases.

[3]Frequently Asked Questions (FAQs) of various news bulletin boards are another example of knowledge organisation.

Chapter 10

Conclusions

Despite the large number of tools described in the literature, the results of tool use are disappointing. Users complain that tuning tools are hard to learn and use and do not provide the information they really need. These problems can be attributed to the fact that tuning tool design is often not informed about tuners and their tasks. This work has conducted an investigation about tuners, their tasks and their requirements of the tools they use currently.

In the absence of studies of parallel programmers, an initial investigation of tuning resulted in the adoption of a framework for further studying how tuning is performed. The framework distinguished two main aspects of the tuning task which were studied further. The first aspect of tuning is relevant to cognitive issues and the impact that available software and hardware tools have on the tuning process (*tuning in the small*). The major difficulties that tuners face are pertinent to having:

- to know a lot about the operation of the underlying software and hardware ("faulty assumption/model").

  In tuning, a lot of knowledge about the specifics of the underlying machine is required and the programmer has to know how a large number of low level entities operate. There are many models of parallel programming and they often involve a large number of inter-related performance determining parameters.

- to eliminate performance bottlenecks when the symptoms of the problems are far away in context from what caused them ("cause/effect chasm").

  It is very difficult for tuners to try to find in large volumes of performance data and tuning information where performance is degraded in the program and establish the reasons for performance bottlenecks.

- and to use immature system software or not fully configured machines ("inadequate tools" and "change").

  Tuning is often performed when hardware and system software are still unstable. Porting the program to a new parallel machine is very common and this introduces many more degrees of freedom in the search for bottlenecks.

Tuners informed this study with their requirements of tuning tools. These requirements were incorporated in a model for tuning tool development. The model is concerned with assisting tuners with problems:

- that stem from the cause/effect chasm

  Tools should provide reliable mechanisms to capture program behaviour and meaningful visualisations of aspects of the program behaviour which can be related to the source code. Abstraction and filtering mechanisms should be adopted to enable tuners to control the level of detail in the search space so as to locate easily the manifestation and the reason for a performance problem.

- that arise from the faulty assumption/model

  The role of tuning tools should shift from merely presenting performance data to performing some of the diagnosis on behalf of the tuner and demonstrating the relation of inter-related performance determining parameters.

> The tuning tool should provide training in the form of documentation or in the form of exemplar codes whose behaviour can demonstrate the factors that affect program behaviour.

Apart from the model, processes through which tool development should take place were proposed. User participation in tool design is the fundamental principle on which tool development should be based. User participation can occur at two levels: first, it can ensure that tools can assist both the experienced and the inexperienced tuners with their tasks; and second, it can provide tool design with information to support the shift of role from presenting performance data to providing performance diagnosis and tuning training.

VISPAT was designed with the help of local tuners to address the issues revealed by the investigation of tuning in the small. It attempts to reduce the impact of the cause/effect chasm by allowing tuners to relate performance data to the source code and to reduce the volume of information by viewing the application program as a hierarchy of phases. The design of VISPAT demonstrated the advantages of user participation in tool design for generating a repository of ideas and requirements for a tool to support tuning in the small.

Performing tuning without a correct model of the performance determining parameters (faulty assumption/model) increases the experimentation needed to tune program performance. The second aspect of tuning revealed by this investigation is relevant to issues which arise from the repetitive nature of tuning (*tuning in the large*) such as the need to document and manage tuning by-products and tuning process and outcome (*tuning rationale*) often in a context of collaboration with others. Further observational studies of tuning have shown that currently tuning management is done in an ad-hoc way that can benefit from a system which can assist tuners in organising the tuning process. Ethnographic techniques were used for an investigation of tuners' work practices and were analysed to see the implications for the design of tuning management systems. A preliminary specification of a system (The Tuner's Workbench (TWB)) was proposed to support documenting the design process of a parallel code by allowing tuners to record the reasons for any design decisions and their outcome. By supporting integration with a number of performance visualisation tools, TWB could address both tuning in the small and tuning in the large:

- it could include a number of performance visualisation tools which tuners could use to assist them with bridging the cause/effect chasm and with the other sources of difficulties in tuning.

- it could provide support for tuning documentation and management.

  Efficient, controlled and well documented experimentation can help tuners to establish faster correct models of relations between performance determining parameters and use this knowledge in future tuning projects or share it within the context of the organisation where parallel program development occurs.

- and it could assist with tuning knowledge dissemination.

  The investigation of tuning has shown that tuning knowledge is often organised in the form of heuristics or cliches. A repository of such cliches based on a set of case histories could help novice tuners who lack tuning strategies by providing them with examples of correct models of how system software and hardware operate and of the relations between performance determining parameters. In this way, TWB could become effectively a tuning training tool.

One of the most important requirements for such a tool is not to interfere with the working practices of those who will use it, in other words to be cost-effective where cost is taken to mean the effort the user will have to put in learning and using the tool. What is considered to be ad-hoc documentation practices may be a very acceptable way of working for many tuners given the time constraints and the difficulty of the task in hand. As Anderson says in [3], "Brokenness is in the eye of the beholder. If you cannot be sure it is broken think twice before setting out to mend it." For this reason, TWB should be designed to be integrated with any tuning tool and without imposing any tuning strategy. At the same time, it should be attempted to tie the system's functionalities with the tuner's programming environment and work practices.

10.1 Future work

It is believed that this work has shed some light on the barriers which parallel programmers face in their effort to tune their programs successfully and it is hoped that the observations made will spur more systematic investigation:

- A number of programming models can be selected in order to study differences and similarities in the manifestation of difficulties stemming from the cause/effect chasm and the faulty assumption/model problems.

- Ethnographic techniques could be employed to study more closely the impact of "change" on cause/effect problems.

- Expert tuners could be observed while performing their tasks and changes in the skills and tuning knowledge of novice tuners could be analysed over time. The findings of this investigation could be used to create user models which in turn could be used to drive the design of tuning tools.

- A more systematic investigation could also refine the model of tuning in the large and expand it to cater especially for requirements that arise in large multi-people development projects. Such an investigation would have to use ethnographic techniques and deal closely with a number of organisations where parallel software development and tuning are performed.

- User participation and a refined model of tuning in the large could be used to drive the design of TWB.

- TWB could be evaluated by local and non-local tuners and tuning case histories could be contributed.

The storage and access of exceptional tuning case histories could be made available for tuners to use as a repository of tuning knowledge and expertise. Apart from helping tuners to eliminate the impact of the cause/effect chasm such example cases could be studied further to reveal more aspects of tuning in the small.

Appendix A

Tuning in the small

Appendix A contains a number of excerpts of discussions with parallel program developers. Parts of some of these excerpts are included in Chapter Four to support the arguments presented there. The selection of these excerpts has been based on their ability to convey briefly and effectively practices which are most representative of the ways parallel program developers work. Other excerpts included in this appendix do not appear in the corresponding chapter. They repeat and support the information given in the representative excerpts and are included here for the benefit of the reader in case they can be used to drive further research on parallel program development.

A.1 Difficulties of tuning

[Ae41] I suppose the number of variables. It is difficult to conduct controlled experiments. Even things like the compiler version make a difference ...having to know machine specifics, how a T3D is. *Informant 45*

[Ae43] Finding algorithmic methods to overcome causes of poor performance such as load balancing and data distribution. *Informant 3*

[Ae44] The domino effect of changes of parameters with respect to performance is quite challenging.

Informant 5

[Ae45] It is difficult to try to keep changes clean and portable. In my program, I assumed that the shared variables were implemented by replication. So, I assumed read operations were fast and write operations involved one or more messages. As an aside, since the network used was an ethernet, I also wanted to have different messages sent at different times to avoid congestion. So, I tried to keep the number of write operations as small as possible. On a point to point network or without replication, this should be implemented differently. *Informant 6*

[Ae46] Tuning is fairly easy with the proper tools. Another hard part is figuring out exactly what gives the performance improvement when some optimisations are performed. For example, when you rearrange basic blocks at compile time based on some heuristics, performance improvement comes from better cache locality (reduce cache misses) and from better branch prediction. The hard part is figuring out exactly what percentage of improvement came from which change. Knowing where to fix and knowing the solution for best performance are two separate issues. Knowing the solution comes from experience and it can take one to many tunings. *Informant 69*

[Ae47] ...I find difficult the question of tuning the program from a macroscopic or microscopic perspective. That is, it is difficult for me to know whether a global big change (algorithm restructuring) might be better than making many minor changes. *Informant 41*

[Ae50] The difficulty is to understand the way of thinking of the programmer who wrote the program. *Informant 7*

[Ae51] The interactions of changes throughout a program can be a problem. Particularly, if you want to work on vector codes. *Informant 11*

[Ae52] Usually, we have enough time for tuning, but this time the hardware became available very late, so the system was developed in one hardware and will be used with another. *Informant 8*

[Ae55] Parallel program optimisation is difficult; even with `apprentice` you don't know where to look. Getting to optimise the serial program can be hard enough because you have to know the things about the RISC architecture that you don't want to know. I know them by reading the *BYTE* and listening to colleagues. If your basis is an unoptimised program, then your measurements of speedup will not be taken seriously by colleagues.

Informant 44

[Ae54] Tuning itself is kind of game, finding strategy, slowly progressing inside a foreign wild programming world. More you know about the writer, faster you will find your way. Indeed, good tools are the weapons you need to survive. *Informant 7*

A.2 Cause/effect chasm

[Ae25] In the Maspar, you have two types of communication, one of them is a global router and the other one is a ...next neighbour communication; because the processors are so small, this neighbour communication is nearly as fast as a local memory access. But, on the other hand, if you go too far by that neighbour communication, you worsen the performance and also there's a chance that too many processors try to send a message through the same route ...so you have to know when to use the global router and when to do next neighbour communication. There are a lot of degrees of freedom, sometimes you change the algorithm in order to get performance and you loose somewhere else. *Informant 42*

[Ae108] I used it initially, to check that B's perception of the problem was actually correct. That we could see that there was a load balancing problem. I mean, `Apprentice` may not be the ideal tool to investigate load balancing because it sums up across all nodes. But you can see that a proportional large amount of time is spent waiting to receive messages. That is basically what happened. You could see that overall, fifty% of the time, was spent in PVM receive or MPI receive and then by determining where the PVM was happening and taking a lot of time, you could see that ...you could guess that this was a result of load imbalance. *Informant 45*

A.3 Faulty assumption/model

[Ae101] Meiko CS-TOOLS offers you all four but the meiko guys are now actively discouraging the use of asynchronous messages because people just don't understand; because if you use an asynchronous message, you have to be able to guarantee that there will be a buffer waiting for it to the other end. And that can require some kind of sophisticated parallel programming to ensure that independently of the relative speeds of the processors on which things are running is always going to be a buffer there. So generally people screw up. Lock up the machine. *Informant 67*

[Ae12] Having to work with constantly changing machines ...Regarding the latest machines, there are certainly two things I am not familiar with. Some years ago, I used to work with `prof` but for parallel codes you don't have that and in terms of...and the latest architectures are RISC, which I am not familiar with, for example, with vector architectures you know what to do, you know with the inner loops and things. In terms of cache use, I am little bit lost. The real problem was the data decomposition, the load balance problem. You know, you cannot optimise the inner core algorithms. If at a level prior to the core algorithm a process is waiting idle for other processes to finish, then we found out that that was where most of the time was held. Because we did the data decomposition on a memory basis for each process, we tried to save memory, in fact then we found that once we were modelling computationally intensive parts of the problem where the bulk of the work was, we were always waiting for these two or three processes to do most of the updates, then the rest could continue. We did not suspect this because we did not have an apriori knowledge of the problem, that the data was irregular. We suspected that something is wrong because the transputer has a number of flashing lights, showing when you are busy, when you are communicating. *Informant 47*

[Ae20] The T3D's performance is not what it should be. We started trying to understand what the problem is. We have understood more about the architecture and IFS's performance on that system. Still we aren't getting performance much higher. On that machine we are not going to get 15% higher performance than it is now. Compiler unrolling should theoretically help, but we 've got some data recently, which shows that it may in some cases do harm. Due to the limited instruction cache on that machine. By unrolling large loops, you may find that the code runs slower. It depends on how aggressive the compiler is in unrolling. A large loop should not unroll anyway. Because it turns not to be a benefit. *Informant 50*

[Ae38] On the C90, if the code is vectorised properly then you cannot do much about performance. On cache machines, we have tried to reorganise the data structures ... it is much more complicated to get good performance on the cache machines. The problem, mainly, is that it is too difficult to optimise for RISC architectures when you have a code that is initially written for a vector machine. So it is more basic the problem we have. The code has been written for vector machines since 20 years ago and all the data structures and the design of all the do layouts were optimised for the C90 vector machine. Now, we are trying to run it on RISC architectures and so this is where we spend most of the time. We have tried to optimise it by rearranging the data structures and by introducing locality, but it doesn't always work the way you believe it should work. It seems that it is more a design problem, on the T3D than our design problem. Because it runs quite well on other machines. We are at the level where it is very difficult to optimise the code without having to rewrite everything and we have only tried to rewrite subtrees and substructures and changing all the loop in index order, things we wouldn't like to do for all the code and it didn't really improve, on the T3D. *Informant 52*

[Ae41a] I mean, one of the problems was that B had an idea about what the costs of the different parts of the model were. For example, we were modelling the airplane which consists of air, metal and cfc and we had figures of what the complexities were, but it turned out that these estimates were completely invalid on the different architecture, because working with air requires a lot of memory copying and working with cfc means that a lot of floating point operations are needed. And T3D is very good at the flops. While memory copying is expensive. *Informant 45*

[Ae21] The strategy for parallelisation we adopted wasn't the only one. But the other options we had would make the code so complicated that it would be unmaintainable, so we had to put up with less performance but with a more maintainable code. Parallel programming is difficult anyway. So, I tried to implement the simplest approach I could think of, to section data in a regular manner. I didn't actually at that time envisage that things could be so computationally heavy. In respect to the actual core code. And the transputer has a number of flashing lights, showing when you are busy, when you are communicating. So many processors were sitting idle. Because experience with parallel programming is so little, most of the people have a sequential background. And so if you get a non regular decomposition that increases the design so much that it is not worth it. And that was were most of the mistakes with the TLM code were made. For every design decision, we asked what is easier to switch to this path or to the other path. We took always the easiest option. The overhead of going another way was just too much. Any problems with the code, the project leader was not there, the learning curve to understand the code was too large. The other two projects were developed in such a hurry, we were trying to parallelise them, that performance optimisation was never a issue. We just got it operating in parallel. *Informant 47*

[Ae7] We started with a traditional task farm. The farmer maintained a grid and the workers had to calculate some values, which were later added to the grid. The workers would gather the results for a number of tasks and then they would send everything back to the farmer. But it turned out that the result phase was taking more than the farming and the calculation phase because of the congestion at the farmer. So we thought that it would be nice to do a combination of a regular domain decomposition and a task farm. So the result solution was decomposed among the workers and because some of the calculation at the borders required communication with the neighbours, we had to exchange some data between the workers but still this reduced the communication a lot. *Informant 55*

A.4 Inapplicable tools

[Ae21] is mentioned in section A.2

[Ae53] In my first serious attempt to program the Intel Delta, I tried using a library routine for a global operation. It turns out that this machine has no performance analysis tools. Which made it very difficult to see what was taking all of the time. I, finally, arrived at the conclusion that it didn't scale well. And I wrote a version that outperformed Intel's version by a large factor. Had I been able to gather reliable profile information up front, it would have prevented me from a lot of guessing over a period of maybe two weeks. *Informant 18*

[Ae74] Paragraph+ is not flexible; you cannot define a set of processors that you want to have performance data about. On the other hand, apprentice gives you data only averaged over all processors. *Informant 51*

[Ae75] Apprentice gives too much condensed information. *Informant 47*

[Ae76] We tried paragraph, but it was very difficult to use and contributed nothing. Tools were too slow in dealing with megabyte files and they didn't allow the analyst the freedom to scan the data freely both forward and backwards and to easily relate different views of the data. *Informant 15*

[Ae90] I think the main thing was a sort of per process specific information, as opposed to the sums given across all processors. This doesn't give a great sense of load balancing. *Informant 45*

[Ae89] For the other project we are doing now, we rely on another company serfax to do the profiling of the code, simply because there is nothing available like, the parallel side of prof on the unix machines. It is something we keep floating because I don't have the time, or the technology you know I am going to write the best parallelisation and hand it over to them. Say you are doing a profiling of your code written in Parmacs and it doesn't add up to 100%, so you are loosing some information somewhere. But we have got a tool like paragraph+ and I think in some ways it gives you too much information, it is showing what is being going ,but in terms of where the time is spent it is not clear where the time is spent in. Cray have a tool that actually shows you where the time is spent, but it is cumulated over all processors, so in some sense it is too much condensed information. As what it would be the ideal solution I don't know except that I would like to have a prof facility for every node in the system. *Informant 47*

[Ae100] We did try using Parasoft Express and PICL/Paragraph towards the end of the project but they were very difficult to use and essentially contributed nothing. *Informant 15*

[Ae85] Apprentice always shows this global view, it shows statistics. You have the total time spent in one subroutine, but you cannot see how much time was spent in this subroutine over the processors. So you can't have a histogram of this subroutine over all the processors to see where the time is spent, that would be nice and also the communications; it would be nice to have it in apprentice as well. *Informant 51*

[Ae78] If you think about doing a reduction operator on the Connection Machine, then there is going to be some idle time on the processor. Now, Prism at the moment doesn't bother to explain that. Prism shows the time from the moment you start, till you finish the operation and it can apply the idle time to the execution time of that instruction. Generally, you 've got some sort of communication operation involving lots of processors. So you think umm ...this is taking a long time and you want to know why this is actually taking such a long time. And maybe it takes a lot of time because you have got particularly inefficient communications pattern that it doesn't map well on the hypercube and sequentialises somewhere. That's usually the truth. And what you 've got to do is sort of fish around some alternative ways to try to do that more quickly. One of the problems with Prism is, that knowing how to make things go quicker, is still a bit of a problem. Because you need to start develop quite a bit of understanding of the actual execution on the machine rather than the semantics of the program. I think that's a gap there that Prism still leaves. *Informant 67*

[Ae102] I could not interpret what apprentice was telling me; I needed better observations and guidance. *Informant 54*

[Ae84] I mean, apprentice gives information like the amount of time spent loading and writing to memory. So you can use that info. But I suppose, I was helped by the knowledge given in the course. I guess in the beginning, when I started using it, I would not be able to suspect what the number of memory accesses really meant. In order to use apprentice fully, you need to be able understand what it is telling you. *Informant 45*

[Ae82] I 've seen these guys from Southampton running Paragraph. I 've never seen them succeeding explaining how they managed to improve the performance of their application on the basis of the communication patterns, the gantt charts and the utilisation terms. *Informant 65*

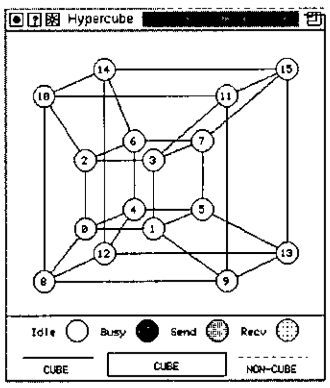


Figure A–1: Communication display in Paragraph

[Ae79] This sort of notion of topology (figure A–1) scares me. Having done quite a lot of teaching of parallel programming to people I came across a lot of people who are obsessed with things being nearest neighbours because they are really important for performance. Really really really important. And I think that I really feel passionately that there is so much for people to take on board, in view of writing parallel programs just to get it right functionally that the topology and the mappings are pure performance issues that they should be stashed until to get them working. So I am very very strongly against inheriting a view of machine topology definitely in the early steps and possibly right the way through. *Informant 67*

[Ae83] We tried to use Tatoo but it was not good enough, it did not work with traces from the target machine. *Informant 12*

[Ae76] is presented in the beginning of this section.

[Ae88] I could tell the problems I had with the Meiko. I wanted to produce a profile of the code, a very simple thing, you know I am not asking for a lot of things. Just a profile on where the time is spent, I had one vector node so I used prof, grpof and the ...profiler. All failed, for three different reasons. So what I did ? I could not get a profile on the CS-2. So I took the routines I vectorised already on the CS-2 and I vectorised the same set on the C90. And I produced the profile on that. So I could see what routines stood out as being the ones I should look at next. But that's the problem. It is great to have a utility, but it must be robust. Because if it fails for whatever reason, it is of no use to you. And the users will never use it again. And profilers are the most basic tools. You couldn't even consider them as tools necessarily. They are basic system utilities. But all of them must work. But you come across things like those failing or the compiler ...maybe you can compile 95% of the routines, with the high optimisation on, but one routine gives you wrong results, if you compile with the highest optimisation. How do you find that one? Maybe on the Cray there is a facility to do that, but you try by the old way by creating directories with optimised and unoptimised objects, you move sets of objects into this and it takes an amount of time, but it is something it could be automated, especially when it is not the only routine that creates the problem but several. And maybe it is not that it is a bug. Maybe it is just an optimisation issue that now your code becomes insensitive to certain compiler optimisations. You have to understand this as well. My experience is that computer industry is one that lacks formal .... there is not real push for standards. You know manufacturers. You have to look what has happened to message passing. The average user just gets confused by how many message passing libraries there are. Why are there so many? Why hasn't MPI been developed earlier? That is a major thing. It takes time to develop new languages, new message passing libraries. I think there should be a greater emphasis in computer industry, say from a certain day all applications (future) should be written in MPI. PVM will be destroyed. With these environments, I think we reached a situation in parallel computing, that the number of problems that we can forsee with the usage of the environments have exploded. Because they are very complex environments that we are working with. Vectorisation is very much localised, rewriting the code from scratch in order to make it vectorisable, where it wasn't before is a very complex process. What are the applications that we can use? It is common phenomenon for example on the Cray that people find that have often to puzzle themselves with a peculiar bug in their code only to find that it is a compiler bug which caused the problem. *Informant 50*

[Ae91] It isn't that bad. The Maspar tool sometimes had a bug so sometimes the timing was pretty rubbish, but you could figure that out easily because it was so far off the mark that you could understand it. *Informant 42*

[Ae99] EXPRESS has got a debugger, but it was completely useless at the time. I tended to break Express and I used to take the machine down and still I don't know why. *Informant 65*

[Ae92] It was easier to use simple timing statements than to try and get the performance tool to measure very specific regions of the code. *Informant 3*

[Ae97] You asked about slow down: This is 256x64 grid and it runs at 27 MFlops/s whereas without the tool (apprentice) it runs at 52 MFlops/s. It runs at 2.7 secs and it runs now at 5.8 wall clock time. That is one of the reasons I use it when I don't know where things are bad. For fine tuning I don't really want to use it. *Informant 44*

[Ae107] When I was using the workstations, I used XPVM but I didn't try to optimise the communications. I used to see who is communicating with whom. Getting to run was great enough. XPVM was unreliable, so I used to get an understanding of the code. *Informant 44*

[Ae104] I built my own tool for a parallel database program. The company was thinking of building a parallel machine to run programs like this, but they made the mistake to use a functional language and it was bad because they could not see where something was wrong or why it was going slowly. So, my tool gave hundreds of different performance metrics about the processors but we would not see where these metrics applied. *Informant 59*

[Ae103] My own tool gave the load balancing for a set of processors. And the problem was that we could not relate it back to the source code. The display was too slow for the rate of the changes at the load balance of the application. *Informant 65*

[Ae106] Because we have Transim and Gecko here. The tools are geared to how you can map data on processors and processes on processors. And it shows communication levels within the system. And processor utilisation. So maybe all the processors are busy all the time but still the program doesn't have any speedup. You cannot simply put the question: Are all the processors busy? Are all the links busy? While Gecko would say that if everything is busy, everything is red, so everything is ok. But this doesn't apply in the case of this simulator. Because there are many cases that the simulator can be working really hard. Everything working red hard. But the simulation doesn't progress. It progresses in very tiny steps. It is like busy waiting times. *Informant 43*

The remaining excerpts in this section are not commented in Chapter Four.

[Ae80] This is the number of tasks done per processor or the number of processes on each processor? The problem with the task is that is a tremendously overloaded term. It means many things. I didn't understand this graph at all. If this is just a number of instances of this task type you've got a graphics worker or slave then that's a static measure that is not going to change over time because you have told the system. If it is maybe the number of tasks that the processor executed for example in the task farming model ...I didn't understand these displays at all as general models. I think that perhaps some of these displays are superfluous. I think there is a real danger especially for a novice or a less experienced programmer to be mislead by which information to use to guide him for the work. Certainly in the performance optimisation I think there are some real red herrings like being obsessed with the .. topology is one of them I mentioned several times. I think there is a bit of a danger. *Informant 67*

(Informant 67 was then shown the message queue display of Paragraph and a communication map between sender and receiver nodes. (figure A–2, A–3)) [Ae81] I don't recognise whether the messages are waiting to be received or be sent. Maybe it's the number of outstanding communications for which you have said "start" and you are waiting to complete. What affects these communications? What prevents them completing? Well other processors may not have been ready to receive them, ah ...what affects that? At that time they are doing compute or other things or there might be other traffic. So while this is kind of important and eventually you are going to hit some sort of limit, or you are gonna have to wait for that message to complete, is really the idle time you are interested in. It's how much time am I having to sit about waiting for outstanding messages to complete. And while that might be related to the number of outstanding messages that you have off going, again its a kind of, sort of, second level metric. I don't think it' s actually something you can directly act on. It kind of suggests a model where you have a processor with some sort of fan out through which all messages are being pushed and you can imagine a battle going on by the rate which this fan out absorbs the messages, and it's not entirely clear to me how you would interpret that. I don't have a good feeling of how this fits in with the other displays. And as I said....... And the important thing is that this is making some very serious assumptions about the message passing model. Because if you are only having a synchronous message passing system and blocking messages so I have to wait for you to receive then there is no need for this display. So that's getting very specific.

Ah .. this notion of mapping of message lengths for sender and receiver combinations, I have no idea of how I could use it. You see the length of the message in most of the messaging systems is almost irrelevant. Because the startup costs are sufficient enough to dominate the communication costs so the issue is the number of messages rather than the size of messages. And the usual optimization trick for reducing that we were talking about earlier is to try to wrap as many messages into one as possible. *Informant 67*

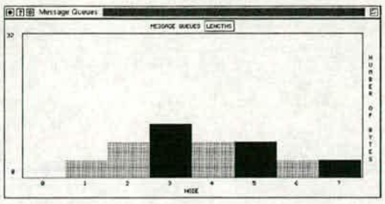


Figure A–2: Length of message queue per node

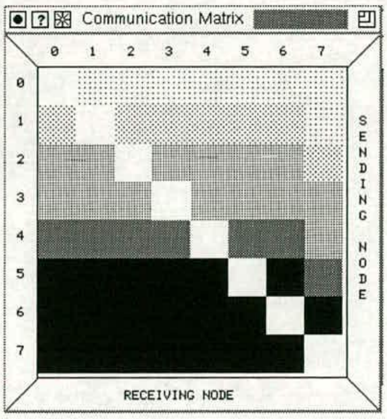


Figure A–3: Communication map

[Ae93] The problem with TMC timers is you have the cmps command giving you some timing of the process and within Prism, CM elapsed, CM busy, and so on CM busy is pretty straightforward to understand but CM elapsed varies from one run to the other if you don't have a dedicated system and you are interested in this timer since the time spent on the front end is not included in CM busy. On MasPar it is pretty more easier; you either use the MPPE profiler or even unix time command and you know where you are. *Informant 13*

[Ae95] I have tried to use the parasoft tools (ctool and xtool) for evaluating communication performance and measuring time to complete certain phases of the computation. I used these on the nCUBE-2 and the iPSC/860 to instrument the entire code by just setting some command line switches or linking ot a library and letting the library dump the information to disk. The parasoft tools then process this dump to provide a visual indication to performance. I also used the library routines to instrument various parts of of a code (marking the time that it got to different points in the code by calling a library routine) In my opinion the data provided was never very reliable, but showed me that I was spending too much time in global communication (no surprise here). *Informant 18*

A.4.1 Tool requirements

[Ae66] Prints on PRISM, better prints on MPPE . . . To print profiling information, without getting thousands of lines (only the meaningful ones); at TMC people usually use CM timers they set themselves in the code! This is not my idea of standard software. *Informant 13*

[Ae67] Visualisation of user-defined traces would be nice .

Informant 12

[Ae68] Profiler is probably a very important tool even though there is a penalty associating in using it. But there are even nicer tools on the ksr for example we also have a tool from BBN called *profview*. It is like *gprof* and *prof* but now you can click on the summary line if you see a particular routine it stands out and you can click on that line and this pops up the window with the source. And it's got little graphic at the side showing how much time has been spent on a per statement basis, of course you have to recompile this routine for it to make sense. To have that detail. And you probably have to compile with the debugging flag to get accurate data on the statement level. Of course, you no longer have exactly the same profile and it slows down your program. But you can see where the time is being spent. *Informant 50*

[Ae69] Where you can define user defined events, so now we can time say how much time we spend in the grid point bits. So we can say what time the different processes are spending in that part. Perhaps in the area of load balance, where we have some special problems with the weather systems, some weather systems are more expensive, than others so that gives you load imbalance, between the processes and you would like to understand this load imbalance. How different is the load balance? Can you do anything about it? So in that area I would like a 2d picture of the processors and the different time spent in different parts of the code something like the typical temperature map. I would like more communications statistics than those we get now . . . such as statistical variational like minimums and maximums. Because if you are running your code on 512 processors how are you going to represent these ? Make tools like paragraph more dynamic for example to be able to specify a certain amount of processors. Paragraph is very static in a way. To change to another subset of the processors dynamically. It could be nice if you could say now I want to look more specifically at ...zoom in in a way. Whereas now you have to stop and also you have to .. it has to be flexible. Now you have to stop and sometimes it doesn't stop, ...where you want to stop it. It would be nicer if you could rewind and change scales and things like that. When you have 512 processors you don't want to look at 512 processors. You would like to see a certain subset of them. And usually the subset would be the outliers of some behaviour. Like which processors didn't behave as the rest. If you could combine it with something like a profiler or if you have a very slow process you would want to look at that process. Why is it that slow? It would be a nice thing to have it and you can do it on some systems. You can do it on Convex and the T3D. It is nice to have the global view in the beginning to see where you are spending your time using the profile but when you come to see that you want to know the load imbalance, the time spent in communication is also included in the load imbalance. Because all the processors they come into the communication routine and then they wait until all the other processors are finished with the calculation so they are waiting, they send out some data and they stop and they receive. So you see all the time spent waiting for the other processors inside the communication. And you can't use apprentice for that, otherwise I could say that the concept of apprentice is quite nice you have this hierarchical presentation, you can go and see the subroutines. And you can either have the subtree or you can only have the subroutines. And it is a nice context to work with. Making the tools reliable and able to work with all sort of problems I guess this is something we should in private companies. where better quality control can be done. But it seems that nobody understands that most of the codes in the real world are big codes. So when you make your tools you have to consider that My requirements are simple and modest and most of the vendors can't give me a debugger a dbx to be able to work on more that one processors. You can do that in totalview I guess. I should try that. *Informant 51*

[Ae70] Synchronisation, I think, is actually more important than load balance. Processes are waiting to get messages and don't do any actual work. I guess you can see load imbalance, when you 've got excessive synchronisation. To get rid off the synchronisation you have to look at the sort of control structure which is more difficult you have to relate the load imbalance to a control structure. The problem you have with load imbalance and measuring load imbalance is relating to it to actual sequences of instructions. So you ask yourself why a particular communication isn't being received. I mean we 've got statistics you can even work out what is it that you are waiting for. Unless you 've got a good understanding of what is going on with your code. So you need at some point to make a connection to the actual code. I 've never seen that done quite successfully. You need trace events. So you need to be able to say: You are running a debugger thing and the monitor system would tell you are waiting an awfully long time for this message. I 've never seen a debugger that would actually trace you back to the sender of that message and it will sort, of wind the thing back to the point at which this message was sent. How come I was waiting for so long to send this message you see what I mean. *Informant 65*

[Ae71] You can only look at profiling data after the code has run completely. It would be nice to have profiling data for an incomplete code. At a break point to have profiling info.

Informant 14

[Ae72] I would like better suggestions on how to improve performance. And also the tool should give quickly accurate profiling info.

Informant 41

[Ae73] I would like graphs to show statistics of my program because currently the statistics are presented in tables.

Informant 43

[Ae94] To print profiling information not getting thousands of lines (only the meaningful ones). At TMC people usually use CM timers they set themselves in their code. This is not my conception of standard software!

Informant 13

A.4.2 Case study I: tool evaluation

[Ae112] ...So you have to have a barrier. This is interesting ...MPItime is spending relatively a lot of time in a barrier. MPItime is one of the control routines. That is something I am missing from apprentice. I would like to tell it show me the routine MPItime (He clicks many times on the name of the routine to show what he would like to be able to do) because it is not always trivial to find your routines. *Informant 44*

[Ae114] Trace based tools like Paragraph tend to be useful only for relative number of processors. I think we have used Paragraph up to 60 processors only for a subset of the scaled down version of the code ...and in a limited way whereas you would like to use it to its full extent. And you can't. On 64 processors we ran our T32 version and we ran out of disk space for the trace file. All these trace files, I created here, are about 2Mb in size and this is only a small version and a limited problem and only 16 processors. And it explodes, if you increase the number of processors but the size becomes a real issue; this is a major limitation here. With this tool you can select your processors but I mean you still create a full trace. *Informant 53*

[Ae117] You should be able to select things that give trace. Because sometimes you are not interested in communication or you would not like to have everything. It would be helpful to be able to control the part of the code that gets instrumented. So the way it is now you either instrument everything or you don't. And the only exception are these user defined events. But, for example, I would like to focus on one transposition at the top time, in one particular timestep, and then simply, I would like to say at the beginning of this section of the code, now, I want to instrument. Which could be of help since I could work with a small section of the code.

Informant 53

[Ae116] Yeah ...So we suggested to replace this one with some easier and more comprehensible displays, like you see event number a started at this time and the last processor finished at this time. And have a distribution of these events over processors. Which would be very useful in load balancing investigation. Which you ...currently is not easily available ...you see the load imbalance in this time displays like gantt charts and you can look at it at one time step and you get a distribution of the different execution times of these events on different processors at this time step. But this is only part of the thing because it could change over the execution and you would also be interested in the statistics over the whole run. As how it performed. It could be accidental the times you got from one time step. It may take twice as long on this processor than on others. So you would be interested in statistics. *Informant 53*

(He invoked the statistics display.) [Ae125] This is the statistics. And the question is do we use it? Not really. And it might be useful but I mean these are things we have checked in a way before. Well it might be useful as a reassurance but it is nothing that you need really. Because you have done these calculations before hand. When you design your communications. How big a buffer need be, what are the ....

[Ae123] A ...yes this is another bug the displays don't get refreshed. And it is very disappointing because more or less the same problems that exist in the public version continue to exist here. Because they haven't solved the underlying problems. Also the refresh here is maybe an IBM problem. We use IBM workstations and Intel thinks that there may be a problem with that. I think IBM workstations need special memory and special software for this refresh. And this isn't a default in IBM systems. I haven't used Paragraph on SGI. But I think they claim that they have solved it. The last time I spoke to people in PALLAS they think that now is only an IBM problem. Which is simple I suppose because we are using only IBM workstations. and we can't test it. We can't prove it." The tuner experiences a tool crash trying to invoke the statistics display. *Informant 53*

[Ae124] The things I mostly used are this colour scan and the feymman display. You see for demo purposes this is great. You see this is another problem .... The scalings are useless. Usually you have messages that are of a little space here and you have the scaling .... I don't know how they get these scalings. I mean these displays are nice to look at you know. And they make a good demo. Which is also an important use of these tools. Because it makes sense for people not used to the idea of parallel programming more understandable what is happening . By giving a presentation and saying that this process is communicating and here you get some animation on how the communication really forms and these things so this is also a useful aspect in some way. But it is not so useful for you who are working on it. Also I think the load to use them would be quite high because in a sense you have to relate these displays together." Second tuner: "Yes yes this is true. I mean the animation here is far too fast if you have not a good idea of what is going on. ...To comprehend.

(The tuner invokes the communication animation display). [Ae126] Well this is the most representative from all that I showed you so far. My personal view is that it is of no use at all. These are indispensable if you want to explain to somebody what is going on. Even if you know. You know it is encouraging. You should use it anyway to confirm that everything is ok that you think it should be. Well I mean it is some kind of reassurance. Lars is quite happy to look at things. Once in a while. Because I mean he really knows what is going on there. But it gives reassurance the fact that its actually happening. like you expected to. And this is also important. and in PPPE there is some body from ESA a company developing the crash code and he says that the best use he can make from such tools is in demonstrating for customers. Because these people don't know about hte details of these programs. and it is hard for them to understand what is going on. Which is not the intended use of the tool s in the first place. Well I think it is a lot. What displays you prefer it is up to your personal taste. in a way because there is so much redundant information. What I like is the spacetime display. and these gantt charts. *Informant 53*

[Ae127] I think these tools would be really widely used if they weren't so many of them. If there was some acceptance that this is the tool to use right? The manufacturers would support them and if they were robust they could go along the way and solve the problem.

[Ae128] You saw this thing today as well. If the tool is not robust it is not going to be used. You know there should be some industrial strength applications to be used with these tools for quality assurance. After this small codes will be run really with no problem. I think people use these tools when they are desperate. You know if the people, having a big problem, try to use a tool and during the learning curve the tool crashed then that is it. They are never going to use it again. It is a big issue.

Informant 50

[Ae119] These tools are useful when you don't know everything about the code. And you don't know exactly what communication patterns are taking place ...that somebody else wrote you know. Some other module, and you are trying to understand why it doesn't scale to a 1000 processors from 500. What is the bottleneck. You want to know what do you have to do to make it scale? You don't want to use the performance tool during the production runs but you should have a quick way to turn on instrumentation when you want it.

Informant 50

[Ae118] At this scale now, you can hardly see what's going on here ...You can zoom in, but it is not very user friendly. You can zoom in, you can configure and ...now, here in this tool you have a very difficult way of zooming in. What you essentially want to do is something like ...click here and ...and mark this time and this time. And it should then be loaded up to the full scale and ...But here, you specify your start time and the end time, and the time unit and these all control what is going on. So it becomes a bit more complicated.

[Ae120] See? It should ...there is a bug obviously. It didn't start so ...I ...as you can see here ...ok .... Here, no, this was probably because I didn't have the right environment set up for this tool. One difficulty with this tool is the way you can control it. (laughters) It can (he is trying to make the tool display events for a particular period of the whole time i.e to zoom in time)...but let's assume that we want to focus now on one of these transpositions here and we can get some info about the time of the start and stop time and then I can set it in this configure menu say stop time is ...and start time is .... and then we want to increase the scale like that. Now it simulates forward in time until it gets to this start time. (The tool replayed the trace file while without presenting anything until the execution time of the events were equal to the start time set by the user. All this time the tool's windows did not display any information.)

[Ae121] It would also be nice to go backward in time. Which it can't also do. A lot of things need to be improved in this particular tool. It is also, I understand it is not so easy to manage the .... What you really want to do is not to mess with these strange time units and you want simply information about how long the run is, and then say I want to go to this time and also it has a too complicated way to control the simulation speed. You have these buttons here but the simulation speed also depends on the choice of these smoothing intervals or stepping command and on all the other things. So it is a more trial and error approach to get to something satisfying and I think it could be a bit more straightforward.

[Ae122] But it is not working obviously. I don't know why. And this is a commercial tool. This is what Intel gives you if you buy a Paragon or what you can buy from Pallas. (Trying to do it again.) But the way it works is that it starts reading this trace file and does all the things. So the proper way to do that is to close these windows, and so that the program thinks that it has nothing to do on the displays, and then only open these windows shortly before you go to the start time. But this is not they way you want it to work is it?

[Ae113] You tend to use these tools as a debugging tool and it doesn't help you if the program doesn't complete. *Informant 53*

A.5 Change

[Be628] is presented in Appendix B.

[Ae15] A bug in CMAX was detected preventing vectorisation of certain reduction loops. Temporaries were inserted to allow vectorisation of these loops, and the bug has been fixed in CMAX 1.0. 6 CMAX-NODEPENDENCE directives were inserted to allow vectorisation of 6 dependence-free loops that CMAX thought contained dependences. CMAX 1.0 vectorises the loops without the directives At this point, when we executed ARPS on a problem of size 35x35x35 on a 32 processor CM-5, the code took 9.9 seconds per iteration of a simulated 6 second time step. The data distribution was to parallelise along the three spatial dimensions, but not along the time dimension. This distribution was determined by directives in the input program, as described above. CMAX obeys user directives, but in other cases it is not sophisticated enough to optimise the data layout. The serial time dimension of length 3 (past, present, future) was the rightmost dimension. Unfortunately, the current version of CM Fortran, CMF 2.1b1, is more efficient when non-parallelised dimensions are gathered at the left. We modified our makefile to call a CMAX utility tool to automatically permute the arrays in ARPS, moving the time dimension to be leftmost. This reduced the per iteration time to 4.9 second per iteration. ARPS had been performing a nearest neighbour computation in one part of the code, which CMAX translated into an expression containing a number of EOSHIFTs (End Off Shift). Unfortunately, the latest version of CSHIFT (Circular Shift) in the CM Fortran run time library is currently more optimized than EOSHIFT. We modified the Fortran 77 code to perform a circular shift, but made this code conditionally compile only for the CM, since it could slow down execution on other platforms. We switched to the latest version of the CM Fortran run time library (which contained the fast CSHIFT) and the iteration execution time dropped to 4.0 seconds per iteration. Next, we switched to the latest version of the CM Fortran compiler itself, to try out the improvements made to its optimiser. We gained about 5% from this change. We were surprised that the impact of the new compiler's improved optimiser was so small. We investigated and discovered an explanation: As mentioned earlier, high level array operations in ARPS (such as differencing) are performed by separate subroutines, each of which loops over entire arrays. CMAX is able to vectorise the loops in each subroutine, and CM Fortran then implements the vectorised loops. However, CM Fortran is unable to fuse the loops between different subroutines, since it does not operate interprocedurally. To overcome this problem, we implemented a simple tool that looked for directives of the form "C-INLINE function from file" and performed the requested subroutine inlining. We added 15 INLINE directives to the program (placing them based on our profiling results) and recompiled. This improved execution speed by 12%. Another problem that profiling pointed out was MAX/MINLOC handling. We found that 3% of overall execution time was being wasted in a line that looked for the location of minimum and maximum values in an array section; CMAX correctly translated these lines to perform MINLOC and MAXLOC on the array section. However, the CM Fortran compiler currently has a deficiency that causes it to perform communication to redistribute the array section across all processors before performing the MINLOC/MAXLOC. The array section happened to contain most of the array, only leaving out a band 1 element wide along certain edges. We inserted Fortran 77 code to mask out this band by inserting large positive or negative numbers, respectively, and then performed MINLOC and MAXLOC on the whole (masked) array. As with the CSHIFT change, this change is conditionalized to only compile for the CM, since it is a pessimization on most platforms. The MAXLOC/MINLOC line executed each time the status of a run is printed out, and the frequency of status reports is controlled by the input file, so this optimization turned out not to be too important for production runs which do not need to print out status reports after every iteration. The final change we made was to perform all computations in double precision rather than single precision. Double precision uses twice as much memory, and thus reduces the largest solvable problem (and the vector length) for a fixed size memory by a factor of two, but switching to double precision still speeded the run up by 15%. The reason is that the CM-5 memory system is optimized for 64-bit operands, which are the standard in the scientific computing community.

Informant 17

[Ae86] You know of Vienna Fortran. It is a very complex system, but it is still a university product in the way that it can't handle real codes, we have some problems of using it with more than 2000 lines code. And we have five hundred subroutines. It is not something that you get academic credit from. Maybe designing tools, but not implementing robust tools. There is a big gap between a tool that is developed at a university and something that can be used by us. And that is understandable . When you are at the university you want to make new developments.

Informant 51

[Ae11] For an extremely fine grained application such as the active chart parsing, it is impossible to have any speedup. Many tricks have to be applied to get performance as good as possible. During this stage many problems concerning Orca and Amoeba were uncovered (mainly bugs and optimisation problems of the Orca implementation). So I avoided using clean high level but extremely expensive Orca constructs. I used a profiler to profile the sequential version of the program and discover what were the expensive Orca statements. *Informant 46*

[Ae22] The parallel version of the program was slower than the serial Orca version for the same test case. As explained this was due to the extremely fine grained nature of the application. First of all, the Orca compiler translates the Orca code to ANSI C and adds the calls to the Orca run-time system. It was the ANSI C serial version which was profiled. Using the profiling information we discovered that Orca does a lot of mallocs and frees which take up a lot of time. Someone in the group found out that this could be implemented more efficiently (which was actually done). Furthermore, we discovered just by trial and error that using the Orca Object construct (comparable with a Modula-2 MODULE) was very slow. We used an Object to implement an ADT and then we implemented in the "Modula-2" like way (using Orca's Module construct) showed that the "Modula-2" was much faster than the Object construct. (This can be partly explained by the fact that Objects are atomic entities and that its operations are executed individually (i.e. all operations on an object are serialized). I didn't use timing functions to time parts of the code. Just the execution time of the program to find the solution. The profiler could not be used for the parallel version. As explained above, mainly the Object construct. Furthermore the GRAPH construct (comparable with POINTERS in Modula-2) caused a great deal of pain. As said before, where possible I changed the Object implementation in a modula-2 like implementation (i.e using no operations) and sometimes letting the user directly access the data structures of the ADT. Seen from a software engineering point of view this was a big NO, but hey it was faster.

Informant 46

[Ae48] The Maspar tools provided most of the relevant information. On KSR, the tools are somehow too cryptic, they did not really give all the information, they showed that you had a lot of cache misses, but they would not really say in which statements.

Informant 42

[Ae17] Recently, we took the last three months to port IFS. Which has already been running on the Meiko, but what he had to do is to tune it for the vector node. And we tuned it for the vector node and we found that there were certain things that run very badly. The compiler wouldn't vectorise certain things. So we did a certain transformation on the source code so it did vectorise. Maybe the loops were too big, or there could be if statements in the loops which don't vectorise so one has to rewrite these things. These are the techniques that we do use. In fact, this example that we used here, is a technique that it was used over 10 years ago. And so the

Meiko is maybe 10 years back in terms of the sophistication of its vectorising compiler. If constructs vectorise on the Cray, the state of the art vectorisers should do exactly the same. Whatever the architecture it is running on. As only a single vector node was available on the CS-2, it was only possible to test the IFS model at the T21L19 resolution, which is two orders of magnitude smaller than the current production T213L31 resolution problem. While it was clear that an increase in resolution would improve the Mflop/s performance it has not been possible to quantify the improvement. To consider optimal performance of the IFS model on the Meiko CS-2 one has to investigate the performance on both the Scalar and Vector PEs independently. This was not done as it was felt that optimal performance should come from the Vector Pes whose 64 bit peak performance is rated at 200 Mflops per node as compared with Scalar PEs, which are peak rated at 40 MFlops. In addition, the IFS model had a history of performing efficiently on vector processor systems so the expectation was that it should do likewise on the CS-2. The approach to optimise the IFS model for the Meiko CS-2 was to use the vectorisation facilities of the pgf77 (Portland Group Fortran 77) compiler. An initial test to use the *-Mvect* compiler vectorisation option for all the 650 routines unfortunately produced wrong results. This was not surprising given the size and the complexity of the IFS code. The approach was therefore modified to vectorise routines, one or a few at a time, checking for both correct results and successful vectorisation. In total, about one fifth of the routines were vectorised. Further vectorisation was deferred until the availability of a reliable profiling tool would enable vectorisation to focus first on significant time consuming routines. Those loops which could not vectorise had to be manually transformed. A loop would not vectorise when the VPu program would be too large to fit in the mVP instruction buffer. Loops containing if then else statements were not able to get vectorised as well. Certain intrinsics were unvectorisable. Unix utilities, such as *prof*, *gprof* and Portland Group utility *pgprof*, all failed to produce a profile for the IFS (these problems were reported to Meiko). The less satisfactory was to produce a profile on the Cray C90 and use this to direct the order in which routines were vectorised. Clearly it made sense to vectorise the most time consuming routines first and not to waste time with routines which contributed little to the wall time.

Informant 50 and Informant 53

A.6 Training in tuning

[Ae31] You are working for a manufacturer. If you have read the manual you may have to give a performance optimisation talk.

Informant 50

[Ae33] The best courses are the ones given by vendors.

Informant 51

[Ae32] Being in a small porting group as above, we often discussed techniques among ourselves. For example, I developed the twin debugging technique that I mentioned, so I presented a seminar on it to our group. Also, every software package we worked on was fully reported. With reasons and source code changes and we could read what had been done with other similar software. There is ultimately no substitute for native cunning. I have, for example, encapsulated the general principles of optimisation in a parallel environment that everyone should know into an one hour seminar which I give in an introductory CM-5 course. I usually use one of the participants programs and I offer approaches which may be tried to optimise. Unfortunately, time constraints mean that people cannot try the optimisations there and then. *Informant 1*

[Ce503] C rnp 21/3/95
C rnp removed if statement from do loop
C rnp modified do loop order
C rnp halved work done in do loop and added transpose copy

Informant 40

Appendix B

Tuning in the large

Appendix B contains a number of excerpts of discussions with parallel program developers and their tuning notes. Parts of some of these excerpts are included in Chapter Five to support the arguments presented there. The selection of these excerpts has been based on their ability to convey briefly and effectively practices which are most representative of the ways parallel program developers work. Other excerpts included in this appendix do not appear in the corresponding chapter. They repeat and support the information given in the representative excerpts and are included here for the benefit of the reader in case they can be used to drive further research on parallel program development.

B.1 Problem solving

[Be158] I divided the task in logical sections and attacked each one in turn.

Informant 49

[Be159] I build a version and tune it, build a version and tune it. We try different algorithmic solutions to the problem.

Informant 4

[Be160] I would see that the problem is really a problem. And then worry about what caused it. *Informant 5*

[Be161] I changed the communication harness and then it ran faster.

Informant 16

[Be162] I use print statements and comment old ones out.

Informant 39

[Be142] I would try to reduce the communications.

Informant 13

[Be26] Problem size is always important in a parallel run. Some very large runs will look really bad on one or two processors, thus making the runs on a larger number of processors look very good. *Informant 19*

[Be13] During the design phase, the bottlenecks were identified (mainly due to previous experiences with previous systems). Since the performance of the hardware was known the system was designed in such a specific way which should hopefully guarantee the performance required. In our system the problem is usually the communication, not the performance of the individual processor. Thus, we have a small code part measuring at a transputer link how much time the processor waits until a sending/receiving actually takes place. This gives us a clue as to where to change the code to use the links more efficiently.

Informant 8

[Be129] I would address the most costly statements.

Informants 41, 44, 45

[Be133] So, maybe somebody would tell you or you could know that the program should run with a certain Mflop performance. Or, it should run with a certain elapsed time and you run it on your new machine and it runs slowly. Maybe twice as slow as what was expected. So, it is really that feeling that certain codes should be better than they are. So, you do the obvious thing. You start by running the profiler and you say is there any routine that it stands out as being abnormal? Typically, you know what we call architectural bottlenecks. Something that runs well on a vector machine may nor run well on a RISC machine. And you have to do things slightly different. One of my early experiences on a RISC machine was to code a routine which on the KSR was running at 20 Mflops/s out of a peak of 40 and we knew, it should be much higher so we rewrote that particular routine, it became very large, but it ran at 35 Mflops/s. You know real peak, because we understood the architecture, we knew what the compiler could do and we just rewrote it. No case is like any other one. They are all different but you build up experience and you understand where problems arrive. *Informant 50*

[Be135] I look at the profiler and start from the most time consuming routines. It really comes down to lots of experience. I use a profiler which gives an indication of the important routines and then I start on them and then hopefully you can find something to do there that helps you in increasing performance.

Informant 42

[Be136] You see whether a routine stands out and you click on that and see whether it has any statements that could be fixed.

Informant 50

[Be137] I address the most costly module first and try to tune it. Rearrange software structure to avoid unnecessary computation and communication.

Informant 12

[Be138] I would track down the big bottlenecks and try to rewrite these lines. *Informants 14, 18*

[Be142] was presented in the beginning of this section.

[Be144] I would like to see how much the program is computing and how much it is communicating, and also to see the most expensive routines in my program. First, I would try to see if there was something in my algorithm that was holding the program. *Informant 62*

[Be131] In terms of optimising for the Cray it is possible another week's time and it could be indefinite. *Informant 45*

[Be132] See where the bottlenecks are and try to achieve the same results with less costly means. *Informant 1*

[Be163] I would profile the sequential code to see what improvements should be made in the parallel version. The first thing I would like to see is the average number of messages sent by each process. If the averages are different significantly I got the internals of my distribution wrong. The code complexity is such that I don't have any strategy, the load, for example, is different in every run. *Informant 61*

[Be140] I optimise costly statements or change the algorithm to get a different method of solution.

Informant 40

[Be628] is presented later on in section B.3

[Be134] One of the main points that became apparent in the tuning course was that optimisations are not necessarily additive or monotonic. That you might try one thing and actually it decreases the performance. And then you try something else and you discover that the first thing you did was right. And you cannot do it very systematically if you like. So it is an ad hoc search. But say it is first hierarchical and then it is trial and error. I suppose you do come with this intention but the tool (apprentice) reinforces that. I mean because of the way it presents the info. So it is hierarchical. I ran apprentice after each change ...to see what the effect of various things were, but then I did it more systematically and I made more extensive changes, as I gained more confidence. The frequency decreased if you like, initially I made a lot of small changes and then, when I understood what the problem was actually, I made more extensive changes. Recently, I stopped using apprentice and I used time calls, possibly, now that I finished the load balancing and the tuning I may go back to apprentice to see at a high level how much the original version without the load balancing varies from the current one. Because the access to the T3D is limited, it motivated us against the idea of changing a lost of small things and see what was happening as quickly as possible. Perhaps, that was one of the reasons, that the frequency of using apprentice decreased, because I realised that it is not possible, or appropriate. *Informant 45*

B.1.1 Experimentation techniques

[Be26] was presented in the previous section.

[Be139] I do a coarse profiling of the application to see which parts are the most expensive ones. And then, I look at the load balance, so at that stage you are more interested in specific detailed characteristics, say, which particular parts of the routine are the most costly ones and they are costly because of problems with the load balance or with the general communications. Or whatever has caused that really and then you are going to figure where in your program was caused this and what has caused it. Was it the data or the algorithm. So you are actually tuning load balance against communication, you would be actually tuning some other parameter and that would have an effect ...to the overall characteristics. That's obviously the one model that comes in mind, I am sure there are other models where that wouldn't be appropriate. Yeah, you could imagine different processes doing completely different work in which case the actual size of the task is undefined you certainly can't determine it beforehand and ...so your work is actually determining how tasks vary, how tasks are characterised in terms of how long they can take, how many they are, what is the typical computation time of a task. Characterise one task in terms of its communications and its computation and the distribution of tasks within the application or within ...how that varies over time, all these parameters can have an effect and the actual question is how you actually go about tuning these, all these divert things are very problem dependent as far as I can see. In many ways, you will need actually to have complete information about the profile or the task balance or whatever in your problem and therefore you would make certain assumptions that you will then try to investigate and you can either go about investigating in more detail the characteristics of your problem or you can try a strategy whereby you attempt to address one part of the problem, so you can address the load balance in isolation from the communication cost and try to observe the effect of the other, if you can actually measure it exactly and measure it correctly. So, in many ways you are not actually looking for the most optimal solution. It would be nice if you could get the optimal solution to your problem in terms of load balance and communication cost etc.. But in general that wouldn't happen, and you look at ...you identify a problem and you address that problem and along the way you uncover other problems, so ... by addressing a problem like load imbalance, you would discover that the different load balance techniques would have a communication cost associated with them. And they may have other costs associated with them as well.

Informant 66

[Bel41] I will start by looking at the computation/communication ratio and I would try to increase it by minimising the number of messages sent and the volume of the messages sent. Or, you make a hypothesis that my program is going slowly because of this ...and you try to get rid of this and see if your hypothesis is true. You also predict which changes are going to work well together because you cannot try everything. *Informant 55*

[Bel45] I would like to see which processors are actually working all the time and which ones are idle waiting for others to finish. I would like to see whether all the processors have to communicate with distant processors on the CM and I would try to fix this. I would try to fix the worst statements first. *Informant 64*

[Bel46] I look at the workload of the processors and then I try to redistribute the tasks if the tool shows me something which is not what I expected.

Informant 58

[Bel47] So the strategy is to start from a high level problem, see the time line diagram (perhaps a gantt chart utilisation diagram) and then see which processes are better than others and then you try to see what is happening to individual messages so I guess you zoom in then. And at that point you could also be helped by a statement profile. *Informant 59*

[Bel48] It would be useful to be able to refer that to ...you have some sort of representation of where processes have been idle ...to refer that to the source code to find out if there are particular regions of the application that are causing problems. Well, you could say that it is always that particular region that the process always seems to be idle. That, given some knowledge of how synchronisation is done, you can decide, you might be able to decide whether it is actually necessary for the process to be idle, if it is waiting for some synchronisation, that it doesn't in fact need, before it can carry on doing some useful work. And maybe there is some part of useful work to be done at that point.

Informant 68

[Bel39] is presented in the beginning of this section.

[Bel52] I parallelise small chunks of code and then I look at the profile of those and the nice thing about these tools is that they are sorted by their sizes from the top. Then I look at ...some portions must take a lot of time obviously because they are the work, and I try to look at those that shouldn't take that much time. For example, when you have a big group you expect that communication will take a large time whereas a short communication would be abnormal to take a lot of time. In order to see why something is taking so much time, I am trying it on half as many processors and you know that something is really going wrong if it still takes a lot of time. I mean tuning is what you do at the end.

Informant 42

[Bel51] I look at the speedup and if things are not good I look at the data distribution. *Informant 18*

[Bel50] It depends a bit. If you are doing an inner product where the number of processors is an important parameter, ...yes, you will run on different numbers of processors. If you are making a change, where you say well, this is either going to speed things up or not, then you would just run for one representative set of data. Because I know how the program behaves. The number of calculations is not strongly dependent on data sets. And I can tell how many calculations it is doing and what the pattern of calculations is. So on then basis of one data set, I can fairly predict what the results on other data sets will be. In fact, the only meaningful parameters for the solver is the number of iterations and the restarts. And the data set as long as it is not all zeroes it is not going to have a major effect. *Informant 44*

[Bel49] I would first run our code on the highest number of processors and look at how much time we spend on communication compared to computation. And then, I would say, ok, I am satisfied with the communication, then I would look at the lower level Cray puts and gets and see if I could do things faster than message passing and I would take a single node and I would look at a profile and try to optimise it as I would optimise any other program for a workstation. *Informant 51*

[Be153] When I started with the parallelisation, I would do some experiments on the machine to get an idea of how I want to do it and get it efficient then I would just put it in the code and then I would fine tune at the end. Well, on the Maspar you have a front end and then you have the back end, which is the machine and on the front end you have one large processor with the sequential data program and the data parallel program will be executed by the distributed processing units. For example, some of the data exchange was done globally, global broadcasts and global gathers were done by the memory of the front end. It turned out it was faster reading the sequential memory and when you had a total read of the sequential memory. There was some synchronisation on that but apparently it was automatically broadcasted and distributed in a form of a tree or something like that, it was really the fastest way to get there. I mean, the other option that you would have would be to send something to the sequential processor and this one would act as the source of the broadcast. So I've worked on the Maspar before, so I knew, but in the very beginning, I just tried a small piece of code. And the way I solve these problems is that I develop two small programs, which have these two kinds of communications, and I see which is the faster one. *Informant 42*

[Be154] I tested ideas on code and then I experiment on real problems.

Informant 11

[Be155] That is true and another nice thing about it is the way we have coded the communication declaration we can split tuning into two tasks. Optimising the communication and optimising the computation. And they are really totally independent and two people can do that. Some of us are trying to get serial performance and some of us are trying to make more clever communication. We usually look at different subtrees. If you look at a profile, i.e where you spend a lot of your time, and then try to divide it out and we haven't really spent much time on optimising ...we are trying to understand more why certain things that you optimise don't give what you expect. Usually, we divide it so we can get different subtrees. Try to see how to optimise those. *Informant 51*

[Be37] Implementing the optimisations is limited by the edit-compile-run cycle. For example, in one case, the solution was easy to come with. I had to move an outer loop into an inner loop. This, though, involved making simultaneous changes to 30 different subroutines. Running old and new codes under debuggers side by side on a workstation with X windows was the only way this was possible. Because the change was so global, one needed to compare the correct behaviour of the program against the altered code in order to eliminate bugs introduced in the recoding.

Informant 1

[Be628] is presented in section B.3

B.1.2 Case study I: verbal protocol analysis

[Be600] I have to see all the routines I have ... (he is searching for the file which contains the version of the routine he wants to test). This is just a wrapper (the source code which contained the call to the specific routine) for ... PVM can go out. This is just a wrapper for the sum routine which is down here (at the end of the file) and I have several versions of that. To see which one is faster (he removed from the main file some calls to PVM functions which had been used in previous experiments). I'll just compile the original routine, this is the main program and I think this is the routine I was working with before (he linked and ran the program). I run it on one processor just to see whether everything is still working; compiling and linking and running as it should do because it has been a time since I worked with this routine and this is the last version ... (a lot of time was spent trying to eliminate the compilation errors and while the program was being compiled the tuner opened the file which was being linked with the program). This is actually to make sure that I am running the file I want to be running.

[Be601] I would use `apprentice`, but as you can see, *this is just one routine and it is a fairly trivial routine, it is really, a really small part of the program so there is not much apprentice could tell me about it.* I prefer to take timings (by inserting timer calls) and check if that is ok, and the one thing I want to do here is I do want to print sum. This is a sanity check. ... You can write beautiful programs but if the results do not compute, then you are in trouble. So I am now putting a line that gives me the summation in one direction. And the result I should have got. Because in inner products timings can be very critical. So the slightest error in timing can give you very strange results. You could for instance have written a very smart routine that 99% gives a right result but if it is called twice in succession it might use an odd result because you forgot to put in a barrier. There are situations when you want to compute 100 inner products in a row, but with a little work in between. If there is too little work in between it might catch one of the old intermediate results instead of doing what it is supposed to do. I guess, I should not have tried for the 32 processors because I am not going to get them.

[Be602] So this is acceptable ... (he runs the program on 16 processors and writes down the result on a piece of paper). And now, what I have to do is change it to something that uses shmem calls to see if this is faster. But before I do this, I want to see what the results are up to now. Now this is 16 processors, (looking into the program log file) with one type of tree – here actually I use an eight way tree for as far as possible – working on the 16 processors the four tree wins; if you are working on more processors, the eight way tree will win because you have large overhead because of synchronisation. Not data transfer. Nevertheless, I want to try the four way tree with shmem_gets, instead of with shared arrays. In order to do that I need a common block, otherwise I am not allowed to use shmem_get.

I have these results somewhere in the file, it is just to remind myself what I was doing and to see if my program is still running. I want to check if it is still ok. This file is already part of a report. In general, if I take the effort to write something on a file or a paper it will turn up in a report at some point else it will be a waste of time.

I run this version here to see if it still works. And I know approximately what the results will be, so this looks ok, so what I want to do is compare this one with the new version. And as libraries and compilers are regularly updated, if you want to compare things it is better to run them one after the other; on the same date, the same machine, the same circumstances. The compiler changed and the library changed and it had a min or effect on my program but it may have a major effect on minor details like this routine. (He introduced the shmem_put calls and he had to compile to see if they would work.)

[Be603] I am not sure whether this will actually work, there might be a synchronisation problem. I hate on-line manuals. ...It is very difficult to find what I want (he attempted to consult the manual pages for shmem_put). Compiler directives, where should I look for those? (The author suggested that he should use the search function in he on-line manual.) Now, the last one I tried was 1f, this one is 1x, and now we get lots of compiler warnings. Oh yes, I have to declare these variables. Aaaaah yes ...(He checked the arguments of the routines.) And I must remember to change all the lines in the routine. The answer is correct but the routine is slower. That is a bit depressing. Aaah ...wait a minute. (He had made a typo mistake.) It was transporting a lot more data than I wanted it to (he fixed 2 typos). Now it should not be slower. That would be really disturbing news. It is slower. I think it is the cache invalidation that is causing the problem. (He un-set the cache invalidation.) Let's see. ...I was running on 16 processors. Right that's it. It is the cache invalidation that slows things down, the problem is, that if don't invalidate the cache, I cannot guarantee that I get the right answers. And as there is a difference of $880\mu s$ vs $937\mu s$ and of 350 Mflops/s vs 372 Mflops/s that is not justifying living dangerously, I will stick to the shared arrays version. To be sure that this is safe, I would have to go into the documentation. I do not want to risk it and I do not want to waste time reading manuals. Actually, I should look at the BLAS routines. BLAS tends to be very highly optimised. *Informant 44*

B.2 Documentation methods

[Be610] After a few experiments, soon I noticed that it is impossible to keep them in mind. So I have written a suite of `Perl` scripts for experiment make log with what binary resident on where, and the compilation condition for each binary. I make graphs semi-automatically. I have also written a tricky makefile to pack compilation flags and cpp-definitions into the binary. So every binary can print out its compilation options when executed with runtime option. I am trying to use the version control (CVS) but retrieving old version is still unnecessary (fortunately). Recently, I noticed that keeping the same binary is not enough to compare with other people's result. Because everybody changes the source code from time to time (including bug fix) and simply comparing new result to old (some people's conditions) result leads to incorrect conclusions. Instead of keeping binary, checking out the latest version, recompiling all the binary with some compile option (cc -dxxx) and comparing make things better. (But it is also cumbersome and it takes rather long time).

Informant 36

[Be611] I had two of these variables against time and you could see what was going on (in the diagram). What I was doing is having a batch file taking times for all these runs. I don't see how you could get away from that. Perhaps if you can build a useful user interface. *Informant 63*

[Be612] You might make a modification to the code that would have to be ripped out later. SCCS and RCS can take care of this handily, so the issue of documentation is not as important as it might seem. *Informant 18*

[Be613] I try to use RCS. Typically, I maintain only one version that I work from continuously trying to improve that version for all runs (i.e. single processor and parallel execution). It is too much to carry around more than one version. It's enough to just maintain program integrity (i.e. obtaining the correct answers when attempting to run via code modifications). This is probably my weak spot. We are usually too busy to do extensive documentation. It is really an important area to maintain good documentation and some of us use RCS to note the changes and then use an online readme file or notes file to hold the intermediate results.

Informant 19

```
 -- Made Makefile, based on CMAX example Makefile.

 -- Bus error after:
    Starting pass 2-3
    Updating primary database
    Updating secondary database
    Vectorising routine <ARPS31>

-------------------------------------------------------------------

April 8

 -- Fixed bug that was causing bus error, ran.

-------------------------------------------------------------------

April 13

 -- Started looking at results of run.  Lots of stuff ends up on front
    end.

 -- Will go through Jonas's changes and integrate them into this
    version, but using #ifdef CM so that portability will be maintained.

 -- After Jonas's changes are in, next thing is to add the axis
    permutation stuff and timing stuff (latter with ifdef's).

 -- NODEPENDENCE's in chksym3d.f possibly out of place; possibly
    unnecessary.

 -- Blew off the I/O changes in dump3d.f.

 -- BAH!  Gave up fooling with the DIFF's.  Found and grabbed Jonas's
    sources.  Highlights:

 -- Split statements for reductions:

<           dkew = 0.5*wrho(i,j, 2  )*w(i,j, 2  ,tlevel)
<      :         + 0.5*wrho(i,j,nz-1)*w(i,j,nz-1,tlevel)
<           kew = kew + dkew
>           kew = kew + 0.5*wrho(i,j, 2  )*w(i,j, 2  ,tlevel)
>           kew = kew + 0.5*wrho(i,j,nz-1)*w(i,j,nz-1,tlevel)

 -- Linear memory sleaze:

<         CALL cpyary_3d(tem1,u    (1,1,1,tim), nx, ny, nz)
>         CALL cpyary(tem1,u    (1,1,1,tim), nx*ny*nz)
```

```
<       CALL flzero_4d(u, nx, ny, nz, nt)

 -- Max of ABS (which shouldn't be needed anymore):

<         tmp = abs(w(i,j,k,tlevel))
<         absmax = max(absmax, tmp)
>         absmax = max(absmax,abs(w(i,j,k,tlevel)))


-------------------------------------------------------------------------

April 14



 -- Commented out Jonas's datavault I/O for now, got things running
    again.


 -- Next thing to do is put the axis permutation in the F77 source and
    add permutation stuff to the makefile.

 -- Another thing to do is the tweak the dimensions so that we're
    running a bigger problem, and so that we're mapping it onto the
    machine better.  Currently, each axis has three slop elements in it,
    so a 32x32x32 problem uses arrays of dimensions 35x35x35.
    Dimensions are in dims.inc:

      parameter(nx=35, ny=35, nz=35)

    When we get around to timing things we should time both as-is and
    CM-tuned problem sizes.  Be sure to adjust cell size in arps31.input
    when we do this.

 -- Trying to run on a 4 PN machine, getting this error:

    CMOST: User yellow interrupt.

    *** RTS-FATAL-UNIX: IOT Trap
    Traceback follows:
       pc = 0xf78bc120    ???
       pc = 0x28c2d4      _CMI_panic
       pc = 0xf7892c4c    ???
       pc = 0xf80f300c    ???
       pc = 0x285190      _CMNA_bc_send_msg
       pc = 0x2725e8      _CMCOM_pe_my_grid_coordinate
       pc = 0x25fad4      _CMCOM_my_grid_coordinate
       pc = 0x251b1c      _CMRT_my_grid_coordinate
       pc = 0x6c2c4       _advw_, line 1238
    pc = 0x5a00c        _advuvw_, line 190
    pc = 0x37780        _frcuvw_, line 320
    pc = 0x11c50        _tinteg_, line 924
    pc = 0xc8a4         _cordintg_, line 392
    pc = 0x8360         _MAIN_, line 688
    pc = 0x24f80c       _main
```

The indicated line is:

```
  FORALL (k = 2:nz - 2, j = 1:ny - 1, ueb100(k - 1,j) .GT. 0.0)
 &    wadv(nx - 1,j,k) = ((urho(nx - 1,j,k) + urho(nx - 1,j,k - 1))
 &    * 0.5) * (w(nx - 1,j,k) - w(nx - 2,j,k)) / dx
```

Will try compiling this file -nonewforall without -O to see if that helps.

-- Yes, that does the trick, but at some cost in performance. Will try just -nonewforall.... Yeah, that works, too.

-- Added timing code conditionalized by #ifdef CM to arps31.f.

-- Added permutation directives to source files and associated hackery to Makefile.

-- Screwage can't do the axis permutation before CMAX'ing because then the axis elision doesn't think the slices are contiguous pieces. So, rework Makefile to do it afterwards.

-- Okay, with that done, things run at speed comparable to the hacked-by-hand CM Fortran.

April 15

-- Added cshift stuff in conditionals, but not in DIFXX, DIFYY, or DIFZZ, because there was no temp to do this right. Could do it halfway in those places.

-- Recompiled with larger problem size (67 x 67 x 35) but don't have quite the right input file for it. Results look promising, however.

-- Preliminary numbers (pre-cshift) per iteration:

	4PN	32PN	64PN
35x35x35:	18.2	4.5	2.8
67x67x35:	62.6	10.6	

```
-- Reconverting for cshift in all but DIFXX, DIFYY, DIFZZ.

-- Bug in CMAX_CSHIFT processing found and fixed.

                        4PN            32PN
   67x67x35:            59.6           10.6

-- After bounds trimmed (so no masking in AVG and DIF ops):

                        4PN            32PN           64PN
   32x32x32:            11.0            2.0            1.33
   35x35x35:            17.2            3.7
   64x64x32:            43.4            6.0            3.36
   67x67x35:            57.5           10.1
   128x128x32:                         22.0           11.42

   Using exact powers of two sure helps.

-- We don't know the number of iterations used to get the RS/6000
   cluster results.  If it's something like 10, we're in good shape.
   If it's something like 60, we're in sad shape.  Jonas said something
   like one processor would do about 90 seconds per iteration, I assume
   of the small problem.  Looking at Figure 5 in the paper that tells
   us that the 900 second figure for the small problem on one processor
   represents 10 iterations.  Let's go with that.  This is the mail
   from Jonas where the 90 second figure comes from:

     From: Jonas Berlin <berlin@Think.COM>
     Date: Mon, 22 Mar 93 12:42:45 EST

     A little update of what we are doing. Ken lost quite a bit of
     interest for a while, but when version 3.1 of ARPS came out I managed
     to get it running in 4 days. Most of the problems I encountered were
     known CMAX bugs or deficiences. The update time on a 16k cm-2 is 8
     seconds, compared to 90 seconds on a slow version RS600. The biggest
     problem now is to get the memory usage down. CMAX wastes quite a bit
     of temporaries. Next thing will be to get some speedup.

-----------------------------------------------------------------------

April 16

-- Turns out the above 90 second figure was wrong, or at least doesn't
   have much to do with what's in the paper.  The graphs in the paper
   are from ARPS 2.0, so we're basically comparing apples and oranges
   here.  Compiling and running on a Sun for some means of comparison.

   Compiled -O with f77, run on Ukko (user CPU time):
```

```
   32x32x32:           31.2
   64x64x32:           75.6

-- Mail on RS/6000 performance:

   Date: Fri, 16 Apr 93 17:14:12 EDT
   From: johnson@vaxdad.scri.fsu.edu

   I do not have presentative timings at this point for v3.1.  Rough timings
   on an RS6000 model 320 (20 mHz clock) is about 2.5 hours CPU for
   the input file supplied with the ARPS distribution (35x35x35 grid
   points, TSTOP=3600 min).  In the coming weeks I should more precise
   timings on various models of RS6000's, a PVM version on a cluster
   of RS6000's, and a Y-MP.  I will send them to you when I get them.

-- So, that's 2.5 hours = 150 minutes for 600 iterations, or one per 15
   seconds.  About twice the speed of a fast Sparc system (Ukko), which
   is certainly more than twice the speed of a CM-5 Sparc.  Hmm.
```

```
April 19

-- Trying without the VU's, just for kicks.

                       4PN            32PN            64PN
   64x64x32:           458.8          39.7

   Oh, foo, those times were with -cmprof, so not quite comparable
```

```
April 20

-- Can't run F77 version on a node because:

   Unsupported unix system call (getrusage) called.

   From:

   cmaml_internal_dispatcher() at 0xb5b40
   CMMP_send_block() at 0x8150c
   CMMD_request_service() at 0x9a814
   cmmd_lseek() at 0x8c2c8
   lseek() at 0x8ec5c
   .fseek.fseek() at 0xece18
   now_acc() at 0xcaae8
   t_runc() at 0xc71a8
   f_clos() at 0xc6c08
   dtadump_() at 0x77ed8
   initout_() at 0x3b3fc
   MAIN_() at 0x30a0
   .main.main() at 0xb9704


-- Trying CMF version.

                       1PN (compiled -node; 4 copies run on 4PN machine)
   32x32x32:           40.2

   This is in line with the other CM5 numbers, so I'll just add it in.

-- Can't run sparc version when compiled without -cmprof:

   CMOST: User segmentation error on PN.

   *** RTS-FATAL-UNIX: IOT Trap
   Traceback follows:
      pc = 0xf782c120   ???
      pc = 0x23e13c     _CMI_panic
      pc = 0xf7802c4c   ???
      pc = 0x22f3fc     _CMCOM_pe_my_grid_coordinate
      pc = 0x21aac4     _CMCOM_funcall
      pc = 0x2107e0     _CMRT_funcall
      pc = 0x1a57f0     _soundg_, line 1048
      pc = 0x1a41f8     _zprofil_, line 571
      pc = 0x199148     _inibase_, line 282
      pc = 0xee91c      _initvar_, line 2771
      pc = 0xd9a5c      _initial_, line 380
      pc = 0x4284       _MAIN_, line 629
      pc = 0x208cd4     _main

   Tried compiling inibase3d.fcm with -cmprof, and that does the trick.

-- Times:
                       4PN             32PN
   32x32x32:           93.0            8.8

-- Since the profiling data reveals that no one routine is being a
   terrible hog at this point, it seems that new approaches could be
   worth looking into.  Changing the decomposition to just X and  Y or
   just X might help a lot -- could take advantage of serial axis
   optimizations.  Might get tricky.  Will try serializing the Z axis
   first, then the Y.

-- Looks pretty simple, actually.  Also changing the layouts in
```

```
    inibase3d to :serial, since that seems to be all I/O.

 -- Here's a problem with this: the axis elision transformation puts
    things in canonical layout, which is not what we want.  We want to
    preserve the serialness of axes:

      SUBROUTINE frcuvw(nx,ny,nz,u,v,w,ptprt,pprt,qv,qc,qr,qi,qs,qh,ubar)

      [...]


\small
CMF$  LAYOUT u(:NEWS,:NEWS,:SERIAL,:SERIAL)
CMF$  LAYOUT v(:NEWS,:NEWS,:SERIAL,:SERIAL)
CMF$  LAYOUT w(:NEWS,:NEWS,:SERIAL,:SERIAL)
      [...]
      CALL mixuvw(nx,ny,nz,u(:,:,:,tlevel),v(:,:,:,tlevel),w(:,:,:,&tlevel))
      [...]
      SUBROUTINE mixuvw(nx,ny,nz,u,v,w,ptprt,pprt,qv,qc,qr,qi,qs,qh,ubar
      [...]
CMF$  LAYOUT u(:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT v(:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT w(:NEWS,:NEWS,:NEWS)
      [...]

 -- Further tuning possible in getting rid of contextualization by
    detecting if loop bounds equal to array bounds, then having two
    separate loop nests; this is worth putting in training and doc
    materials.  For ARPS, look at a3dmax, aamult, advcts_v1,

------------------------------------------------------------------------

April 21

 -- Could try using interface blocks, perhaps?  Or have axis elision
    transform emit them, and do the right layout propagation?

------------------------------------------------------------------------

May 4

 -- The CMF compiler won't be up to passing sections without VPMOVE's
    until 2.1 Beta 2, so we'll wait until then to push ahead on this
    front.

 -- Will try 2.1 Beta 1 (have been using 2.1 Beta 0.1) to see if that
    helps in any noticable way.

------------------------------------------------------------------------
```

```
May 5

 -- Running a big problem on 64 PN's:

                           4PN              32PN             64PN
    32x32x32:              11.0              2.0              1.33
    64x64x32:              43.4              6.0              3.36
    128x128x32:                             22.0             11.42
    256x256x32:                                              43.57

-----------------------------------------------------------------------

May 18

 -- With things tuned, some times:

                           4PN              32PN             64PN
    32x32x32:              6.23             1.15             0.76
    64x64x32:                               4.87             2.06
    128x128x32:                                              7.07

    Unfortunately, we can no longer run 64x64x32 on 4 PNs.

 -- Time from Ukko on this serial version for complete run, with
    initialization and 600 time steps:

    12272.9u 655.3s 3:35:41 99% 0+-4312k 13+17io 116pf+0w

    Which is 20.45 sec/it average.  The CM time is not too hot compared
    a bare Sparc, although Ukko is perhaps twice as fast as a node
    processor, and the node is double double precision math while Ukko
    was doing single precision....

 -- Can't even run 32x32x32 on 1 PN, which truly sucks.

 -- Will move these arrays to :SERIAL layout to try to cut down VU
    memory usage:

    CM    QVBAR(:NEWS,:NEWS,:NEWS)<Directive><Warning: No vector usage>
    CM    QS(:NEWS,:NEWS,:NEWS,:SERIAL)<Directive><Warning: No vector usage>
    CM    QV(:NEWS,:NEWS,:NEWS,:SERIAL)<Directive><Warning: No vector usage>
    CM    QI(:NEWS,:NEWS,:NEWS,:SERIAL)<Directive><Warning: No vector usage>
    CM    J3(:NEWS,:NEWS,:NEWS)<Directive><Warning: No vector usage>
    CM    QH(:NEWS,:NEWS,:NEWS,:SERIAL)<Directive><Warning: No vector usage>
    CM    J2(:NEWS,:NEWS,:NEWS)<Directive><Warning: No vector usage>
    CM    ZP(:NEWS,:NEWS,:NEWS)<Directive><Warning: No vector usage>
    CM    HTERAIN(:NEWS,:NEWS)<Directive><Warning: No vector usage>
    CM    J1(:NEWS,:NEWS,:NEWS)<Directive><Warning: No vector usage>
    CM   Z(:NEWS)<Directive><Warning: No vector usage>
    CM   Y(:NEWS)<Directive><Warning: No vector usage>
    CM   X(:NEWS)<Directive><Warning: No vector usage>

 -- Nope, the Q guys should still be :NEWS....

    Conflicting directives for array QS in ARPS31 [arps31.f]
    Conflicting directives for array QV in ARPS31 [arps31.f]
    Conflicting directives for array QI in ARPS31 [arps31.f]
    Conflicting directives for array QH in ARPS31 [arps31.f]

    Must be a bug somewhere in the "no vector usage" detector, hmm?

 -- Running short on single and double on ukko:

    238.7u 39.4s 5:26 85% 0+8816k 10+7io 117pf+0w    single
    261.5u 51.7s 8:09 63% 0+8784k 11+7io 162pf+0w    double

 -- Running short on single and double on smitty-cm5-n3 (cmix):

    271.6u 15.7s 5:07 93% 0+7744k 3+5io 123pf+0w     single
    271.2u 15.7s 5:12 91% 0+8052k 3+5io 154pf+0w     double

    Hmm, is that right?  No time difference between single and double?

 -- These numbers are not all that impressive.  It looks like we'll be
    lucky to get 160 MFlops out of a 64 PN machine, which is 2.5% of
    peak.  That's about a factor of 10 from where we should be.  So
    where is all the time going?  Profiling doesn't help much at this
    point; there's some very basic problem, I think.

 -- Well, the version with all those arrays serial dies with a bus
    error, so I guess we back out those changes.

 -- Data for complete runs on a dedicated machine:

    32x32x32 on 64 PNs:
        CM Elapsed time: 511.029 seconds.
        CM busy Time: 451.358 seconds.
         FORTRAN STOP
        497.0u 9.5s 9:33 88% 0+2508k 9+130io 318pf+0w

    64x64x32 on 64 PNs:
        CM Elapsed time: 1309.239 seconds.
        CM busy Time: 1200.621 seconds.
         FORTRAN STOP
        1260.6u 16.3s 23:51 89% 0+2536k 7+132io 317pf+0w

-----------------------------------------------------------------------
```

```
May 19

 -- Data for complete 32x32x32 run on-a-node:

    % time arps31-single < arps31-bench.input
    ...
    15916.5u 803.0s 4:39:43 99% 0+-3004k 3+7io 166pf+0w

 -- So, CMF on a 64 PN machine is 4:39:43/0:09:33 = 29.29 times faster
    than F77 on one PN.  The speedup for the larger problem would be
    more.  Unfortunately, it would take about 16 hours to run the large
    problem on one PN, and we might not get that dedicated time.  If the
    serial time scales exactly by 4.0, then we would come in at 46.9
    times faster.

 -- Short run of 64x64x32 on-a-node:

    % time arps31-64x64x32-single < arps31-short.input
    ...
    1166.6u 914.2s 2:24:46 23% 0+9504k 2+5io 175642pf+0w
                           ^^^
    The problem here is that the nodes on Smitty only have 16 Meg, and
    so it's is paging (over the network to an SDA).  Boo hiss.

------------------------------------------------------------------------

May 21

 -- Data for short run 64x64x32 in F77 on-a-node with BIG MEMORY:

    864.7u 7.5s 20:47 69% 0+880k 3+9io 53pf+0w

    This looks a little better, eh?  So: we'll run the biggest thing we
    can on the BIG MEMORY machine.

 -- Running short run 128x128x32 on 4 PN BIG MEMORY machine:

    CM Elapsed time: 1124.630 seconds.
    CM busy Time: 1033.380 seconds.
     FORTRAN STOP
    940.7u 4.4s 23:32 66% 0+2408k 10+13io 307pf+0w

    That's 103.3 sec/iteration.

 -- Rounding out with 64x64x32 time we now have:

                        4PN             32PN            64PN
    32x32x32:           6.23            1.15            0.76
```

64x64x32:	24.5	4.87	2.06
128x128x32:	103.3	13.2	7.07

There seems to be some degradation here going from 64x64x32 to 128x128x32, perhaps due to the boundary condition handling. However, whatever the reason, the speedup factor gets better with larger problems, going from a factor of 8 between 4 and 64 PN on the small problem to a factor of over 14 on the large problem.

July 8

-- Trying to figure out what runs we can make with a couple of hours of 1K processor time. For full runs, we're probably looking at something like:

	1K PN
256x256x32	:40
512x512x32	2:30
1024x1024x32	(no way)

-- So, we'll get some short times and extrapolate from those. To be complete about this, we'd need the iteration times for all the other runs, which we may be able to scrounge up somehow. Or, we can just count the setup time for a small run and extrapolate. Or something.

-- Times for short (10 iteration) runs, actually measured, in seconds. "Total" is real time, "10 its" is CM elapsed time for 10 iterations:

	128 PN		256 PN		512 PN	
	Total	10 its	Total	10 its	Total	10 its
256x256x32						
512x512x32					604	200.2
1024x1024x32						

-- For full run:

	128 PN		256 PN		512 PN	
	Total	600 its	Total	600 its	Total	600 its
64x64x32					702	487.7
128x128x32						
256x256x32					3197	2694.2
512x512x32					12512	10803.5
1024x1024x32						

-- Also could include memory usage figures:

512 PN

```
     64x64x32   1.11 G
   128x128x32
   128x128x32   1.44 G
   256x256x32   2.68 G
   512x512x32   7.77 G
 1024x1024x32

  128 PN   4 G
  256 PN   8 G
  512 PN  16 G
 1024 PN  32 G

 On 32 PN, 32x32x32 uses 94.2M.

-- Need to do short runs, obviously.  Might have time for 512x512x32
   full run.  Set limits at:

                    1K PN
   512x512x32        :10     (just like 512 PN run)
 1024x1024x32        :20     (double for good measure)

-- Submit these jobs:

   time arps31-512x512x32 < arps31-short.input     limit 10
   time arps31-1024x1024x32 < arps31-short.input   limit 20
   time arps31-256x256x32 < arps31-bench.input     limit 45
   time arps31-512x512x32 < arps31-bench.input     limit 2:30
```

Informant 17

B.2.1 Reasons for documenting tuning

[Be612] and [Be613] are presented in section .

[Be614] To have a place holder in order to be able to put a project on hold for a period of time and be able to return to that project at a later time without having to reinvent the wheel. We juggle many projects at a time and this is a key not to regress.

Informant 19

[Be615] To remind myself of what I was thinking of and why I did it this way. *Informant 20*

[Be616] I think it is probably worth keeping even the bad cases. I think they are worth keeping because there might be some particular bad thing in the configuration which has caused a . . . Identifying where the worst case is is useful as identifying what the best case is. Worst case avoidance could be a reasonable way of tuning the program. *Informant 68*

[Be617] To keep track of changes and maintain the history of the software. Also to be able to explain why the speed differences.

Informant 7

[Be618] To use as a learning aid for future performance tuning sessions.

Informant 21

[Be619] To have an idea of how these tuning changes will be different on other systems. *Informant 11*

B.3 Tool requirements

[Be522] Documentation has been a problem in the sense that the formal documentation, required for the software standards, was imposed half way through this project. So, it was never originally planned that the documents that were ultimately required would be required. But other problems we find are the continuous documentation of problems, hints tricks, things that people have found in use and have it up here (he showed his head) and they never wrote them down so that everyone else can use. Yes, the biggest problem is with the little tricks and hints, ways around the problems, the formal documentation is not so much of a problem. We do have some means to trying to document that, but persuading people to write it down to paper has been difficult. So there is a problem in documenting at that level. But I would say that this documentation is satisfactory. But other than that, the problem with the documentation was that the formal documentation was imposed half way through the project. We have never used any versioning system. *Informant 48*

[Be523] We would be interested in having something, that integrates your kind of environment, where you can query a database of outcomes of experiments. And have a kind of version management included . . . you have an application and you are going to be running thousands and thousands of data sets with this, or a large number of data sets, you have to get to record that, and maybe you have the number of processors as one factor, maybe the optimisation level of the compiler, etc. And it was exactly the idea, for example, the system would automatically record what optimisations you used or when you use a visualisation tool, the trace files would be linked to makefiles that you used. And it could associate the options for the compiler and the outcome of the experiment. It was only thought to keep track of the relationships in your development process, what you have done, what versions you created. It would certainly be helpful to have something like that. If you extend it, you can even create a database of case histories that other people can go and look up, for cases that are similar to this, and say some one has a problem on one machine and he describes a bit of the problem and then somebody could access it and benefit from this experience. You get a lot of that by user groups. Anybody who is using a new machine architecture would almost certainly want to collaborate or have some meetings with people doing similar work. *Informant 50*

[Be524] Well, I think we are going to have to have some kind of configuration control method mechanised for it because he have a number of upgrades to do, and things are relatively easy when you are in a prototype stage of a project. You can have versions proliferated. Once you get into production mode, things become more strict. You have to record specific versions of the code. So, I do anticipate that at some point we will use some form of computerised control over the versions of the code. Currently, we have a logbook for that ...but out of the design procedures every change of the software or every hacking has to be done through proper documentation and agreed by the people in the team. So, there shouldn't be any informal changes to the code.

Informant 48

[Be526] Right yeah, so some kind of database with versioning control. Yes, that would be very nice. It reduces the risks, you know, if you change the code too much that you cannot get back to a previous version. It seems very difficult to achieve. In terms of things like that we will have to spend some time in the project. But at the moment, we don't know of anything that could provide us with this kind of documentation. *Informant 47*

[Be527] Yes, I think it would probably be useful, the way we are working here today is from this point of view very much haphazard. So any specific time connection or even just going and asking people to see if they 've done it, or doing it myself, if you have a central repository for this. ldots that is a significant problem, actually keeping track of the version you have on the various machines, we do have a very good version control tool, which is `Clearcase`. In the parallel world so far, the guys have found it restricting or painful to work with, but partly or even mainly they intend to be developing on our 6000 workstations and ClearCase runs on the SGI platforms which are the platforms all the researchers work on and where the fileserver is. So we bought it only for that platform. The fact remains, if you don't have an SGI, it is a little more clumsy to use because the database is somewhere else. And therefore developing on the RS6000, they have to go through some short of NFS or some sort of network to get a ...And they find it a little bit ...it affects their developing speed. So they are reluctant to use it and therefore we have the situation where there are lots of tar files lying there, and it is slightly difficult to follow what you have. There are comment fields in many places, but they tend to be limited ...They are simple character strings appended to the version. It is a very simple level, you could of course impose your own documentation standards, you could have in the Clearcase database each change to reflect this information but it would only be a text file. *Informant 52*

[Be529] A tuning management tool proposed within the context of the PPPE project was dropped because of the persistence of most of the users and most of the vendors to implement it. The argument was from the vendors point of view, that it is one more software layer and since it is all from different companies, it comes more messy for them to manage the licence rights and royalties to pay and it becomes more complicated to market. And most of the users' point of view was that the tool will degrade the developers' performance. Saving the result of a tuning experiment in a database has some cost associated with it, which users with small codes are not prepared to afford. *Informant 53*

day	secs	mflops***1	notes
13**	18.1	132	recompiled meiko set of vectorised routines on c90, performance suggested that more routines need to vectorised and probably further optimisation of already vectorised routines. produced profile on c90 and gprof, prof and pgprof all fail to work (for ifs) on cs-2
15	181	13.2	vectorised: trltog, trgtol, had to work around compiler bug in trltog and trgtol, bad object code was produced for identical loop in both routines
16	171	14.0	vectorised: sigam, surgri, sc2fsc,radlsw radlsw failed to compile, got an internal compiler error workaround was to use cdir-novector directive
17	157	15.3	vectorised : vdsfsdrv, no connection to cerfacs possible
22**	150	16.0	recompiled with nproma=64, 128 was probably large enough
22**			recompiled with nproma=256 run failed with an addressing error
22**	11.7	205	recompiled latest meiko set of vectorised routines on c90 with nproma=64,as 220 mflop was obtained when all routines were vectorised it looks like we wont get much more performance on the meiko by vectorising more routines.
24			updated sources to latest cy12 versions
25	133	18.0	more work on :lwu, radact,swu,mainly changes x**y to exp(y*log(x)) to get loops to vectorise

Table B–1: [Be627] Diary of porting IFS to CS-2

Appendix C

Tuning across organisations, people and machines

Appendix C contains a number of excerpts of discussions with parallel program developers and material kindly given by EPCC's User Support Service. Parts of some of these excerpts are included in Chapter Six to support the arguments presented in that chapter. The selection of these excerpts has been based on their ability to convey briefly and effectively practices which are most representative of the ways parallel program developers work. Other excerpts included in this appendix do not appear in the corresponding chapter. They repeat and support the information given in the representative excerpts and are included here for the benefit of the reader in case they can be used to drive further research on parallel program development.

C.1 Different attitudes to tuning

> [Ce30] Most people don't have to. In terms of optimisations, they rely mainly on the compiler optimisations and some people don't even exploit that because they don't know that there are certain flags, you can turn on For example, a code is running on double precision whereas it can run faster in simple precision. This is talking in Cray terminology, they forget that there are a few people out there, that if the code doesn't run long enough then it doesn't make any difference. Then, there is the other extreme like here in the weather forecast, where performance is a very important issue. If we can get the operational model code run 10% faster, we get our salary increased. There is a value associated with that, but equally, we can't go and modify 90% percent of the code to gain this performance because we will get the code unmaintainable. *Informant 53*

> [Ce28] Tuning? We don't do it so much. We are always more interested in providing functionality than providing performance. We hope that we will

have the time to implement a second version of the code, but this never happens. Having said that, we try to think about performance and no do anything stupid.

Informant 56

[Ce8] No, that was to take it away from the C90 because it has been running only on the C90. And people have always coded for the C90. And uses features like memory management, dynamic arrays and pointers, so first thing is to make it portable, the first test is to run it on the workstation. And next to another message passing machine with the same level of parallelism that you have in the C90. I did that version that runs on the workstation and then I changed some of the data structures which were not so suitable for parallelism in a distributed memory; so then I rewrote the data structures for this version on the workstation, on a distributed memory machine we had to reorganise all the data structures. Well not all but some of the data structures. So the parallel part was to put all the PARMACS commands which send the data between the processors, that is what they wrote in Germany. We had to make it portable so we only allowed a subset of Fortran 77 and all the Cray specific features have been removed so it is quite different. The number of lines that I modified is quite a high number. The forecast model is around 100,000 lines. The message passing version is the same plus 5,000 for message passing; the strategy we had was one of the basic strategies, was to keep the serial code as close as possible to the parallel code. Because of maintenance reasons, we didn't want everybody here to start recoding, they don't want to recode the basic code. And we want to keep it invisible to many people but a few who have to know how the parallel code is organised. The code is a spectrum model because we have three different space models. We have three different data and distribution layouts, so what we do is we move the data around in memory between those three different data layouts, and then the data is how each processor want it to be so, so all the computations are done serial, on a subset, so it is SPMD approach. We are not using any parallel algorithms at all. So for a Fourier transform we do it serially. Most of the work was also to design this message passing version so you change as little as possible in the serial code. No, we are always worried about performance. But what we did was we estimated how much how the parallelism was for the strategy we were using. And try it on some simpler models, one level models to see if this strategy was working ok, and we could tell from these estimates that it should be very efficient. The most efficient way you could do things. So it turned out to be correct. We have tried really to design it. Because we have these 300,000 lines of code, we don't want starting to do it using the wrong track then you can't afford that. It takes you a year to do the work so you cannot afford ...so you have to design your work properly. A way to improve the parallelisation strategy was to use this simplified done level model and use the same communications strategy. And also look at what other people ...a group in the states did it in another way. We have stuck to what was proposed here, by those two guys who worked here and then it turns out to be the best way. The optimisations were not that many.

We have also tried to make the code flexible so that it can run on different architectures using switches. Yes, you can specify if it is a vector machine or if it is a cache based machine. And we tried to put that in. For performance I use `Paragraph+` but not very much. I want to see whether things happen the way I think they should happen. I look at the communication patterns and not very much for optimisation reasons. And perhaps also to see some load imbalance, to see if some of the processors are doing more work than the others. I did use it for presentations. I am also using `dbx`, in the initial phase, to understand the code. See what the variables are. There are some more advanced tools that I am not using on the C90. There is this `Totalview`. You can ask about variables where they are used and defined. I also made some tools myself. Perhaps they should not be called tools but they are calling trees to see where variables are used. And understand the data flow in the program. And we have made the code flexible, so you specify how you want to communicate depending on the architecture, so everything is available in the communication. You just specify by special switches what you prefer to do and then that is also what the vendors do. They try different switches and decide what is the first choice for them.

Informant 51

[Ce29] Real time programming doesn't fit many of the usual assumptions about parallel programming; for instance, tuning is only necessary until the program runs fast enough. *Informant 15*

[Ce19] It would depend on what machine the code was going to target. We don't want too many optimisations, because we don't know what machine we are going to buy. We have to make sure, that we are not spoiling the code for the vector machine, when we do specific things for the cache based machines. So at that level, we try to make the code flexible, so it can run well on cache based machines. When we decide which machine we are going to buy, we are going to optimise specifically for that. *Informant 52*

[Ce27] We were concerned that in the production run we were going to have some very long runs. We were anticipating 10 days of run times. During a length of run, which gets so long, there is a high risk that the computer itself will go down, so we could not have a single run in a single go. So, that meant, that we had to store the state of the computation, periodically in case the machine went down, so we could start from there rather than from the beginning (check-pointing). Now, we had to make this ourselves because our computer platform didn't have this built in. Check-pointing introduced performance problems because it had to cope with all the program's data structures. For a short run, it is not worth using check-pointing, but is worth spending 2 hours in every 24 hours for a ten day run. Performance has been to some degree a problem, because the algorithm was not as efficient as originally, we hoped, we could make it. Whilst we had an optimisation plan, there were very few possibilities that we could get the funds to implement it. We had constraints in time and money. We have never used a performance optimisation tool. Partly, because we didn't have a performance evaluation tool on the Parsytec machine. We did evaluate the performance, in the sense, that we timed some of the processes within the algorithm. So we had some more information on this algorithm. It never really became an issue, because we have never been able to do much about that performance. But now, we have the ability to run the same code both on the Parsytec Explorer and on the Cray T3D. So we have a fair amount of power and optimisation is less significant than it was before. There are some things, that we need to tackle, that are relatively simple and one of them concerns the formatted or unformatted output of data. At the moment, this is extremely inefficiently done, but it is expected we are going to gain a lot of performance by changing the way the data is output. Another development will be the introduction of a more efficient way to solve the model. This will have to be included in the operational code. So the development of the code is an on-going process and performance optimisation is an on-going process as well.

Informant 48

C.2 Case study I: the LIBRA project

(Within the LIBRA project, two local tuners were involved in designing a data distribution library for the code of B. A lot of time was spent in trying to design a number of data distribution schemes, which would then be tested for performance with the parallel code. The data distribution library would get the input data set and would partition it according to the amount of work involved at each data point. They had to transform the communications within the code to be able to use the data distribution proposed by the data distribution library. After the code was integrated with the distribution library, they measured the execution time of the program in order to see how the new distributions performed compared to the regular decomposition. `Apprentice` was used heavily in the beginning when they wanted to understand what was going on with the code.)[Ce24] In the original code, because of the NSEW communication, there was a certain way of communicating in the regular domain decomposition. In the original code, Parix communications were based on the OCCAM model, where communication can take place through four channels. But here, we introduced more neighbours. The other modification, we did, was that Parix doesn't support any high level communications support so you do a broadcasting by explicitly doing a number of messages ...a loop of messages. So we introduced the MPI broadcast. One of us was analysing the code that we were given to decide whether it would be feasible to introduce this sort of communication scheme or whether we would need to stick to the regular scheme. And the other side of the work was that we tried to design different decomposition strategies like these pictures here. One of the extreme things that we thought of doing was, you typically have like these large amounts of air with some other material in little places in the middle. So one suggestion was to distribute the work on the air evenly and then you consider the left bits in isolation to distribute them on a number of processors, but looking at the code, it was decided that it would be extremely complicated. If we can get an accurate estimate for the ratios for each material, then our solution is good because it is guaranteed that each time you split the work in half. But it depends on how accurate these estimates are, in the first place. We can find out the accurate ratios by using timers to time the program on the T3D which operates on a simple data set comprised a plane of one material. Another slight complication is just that there is not only one single cost for the cfc. It depends on the fitness and various other features of the data sets. So we need to determine a range of values for cfc and then experimentally plug this into the load balancing. One of the things, that came out of that, was that the way the input has been done on the T3D it took a lot of time some ten minutes took to read the input file. So for short problems, it didn't matter how much we improved the code since the I/O was so pure. At the moment, the performance has improved 2.5 times using a standard problem structure and size. The other thing is the way memory copying is being done on the T3d and the work involved a lot of things propagating in the workspace. And the optimisation for this is to move memory copying expressions out of the loops. And scheduling things in order to make better use of the cache. And a lot of things that the compiler should do but it doesn't necessarily do them on the T3D. That became apparent with apprentice that a lot of time was spent by doing memory copies. So that was one of the improvements. *Informant 45*

[Ce506] There is a kind of use plan for this code in terms of design and use. There is also a development plan over the period of the next six years. So, there will never be a point in time that the code will be finalised. Performance to a degree is bound into that. Many of the issues in the next versions will be concerned with performance, for example, to have a time domain filter process that we want to put into it. Which will allow us to down-sample the data as we generate it so that we don't have to store every time-step, that is another performance related issue which reduces the time we spent in outputting data. *Informant 48*

C.3 Case study II: development in X

```
[Ce530] T106L19, Nproma=320 Full Grid  MFLOP=5207 Estim Seq Time = 99.54
Nproc Elapsed  Speedup  Mflops Comms MAXCPU MINCPU AVGCPU  TOTCPU TOTVU
4      26.07    3.81     1997   1.10  24.96  24.76  24.86   99.43  85.63
8      13.38    7.44     3892   0.76  12.62  12.33  12.45   99.64  85.63
:        :        :        :     :      :      :      :       :      :
54      2.13   46.73    24449   0.17   1.96   1.40   1.89  102.20  85.69
Comment:
The basic vector efficiency has improved by over 30%. Parallel
efficiency continues to improve particularly for the reduced grid case
where on 58 nodes the load imbalance and communications overhead at
0.28 seconds is actually less than in the T41L19 case.
```

[Ce39] What, I think, is hard is that you really need to know a lot about what the compiler is doing and about what the hardware is doing. And these relationships is not what the typical user is primarily interested. And the scientists, the most they want to do, is ok, I want to run this code with the optimisation flag set and that is it and I don't want to spend a lot of time in restructuring the code so that it runs faster. But you might have to do that, if you want to gain this small percentage of improvement. And it is becoming a lot harder, I mean, you have these examples of what happened on the Cray T3D, where you can do a typical optimisation with unrolling and apply that and past experience says you gain something, and then you suddenly realise you don't gain, Then, you find out that there is a clash in where the instructions are put into. This is hard to understand, I mean it is not well ...If you were talking to a typical user about this you would be talking in a foreign language. It wouldn't mean anything to him. He would say: what do I have to do in my Fortran program to stop this. Well, you can't do anything about it sorry. Just carry on doing what you are doing. Do your best. Their thoughts are to write modular code writing efficient code from the algorithmic point of view. You know we can tune things and get things running 20% faster but they can do a lot more with algorithms, if they decide to use a different solver, maybe there is another way of doing the same thing. If they could use a BLAS3 routines rather than calling BLAS2 many times, for example. Look at how solvers have improved in efficiency over the years. It is not because computer hardware has become better, it is because scientists have used new solvers.

Informant 50

[Ce510] We also deliver a benchmark to computer companies, so I spent some time setting up program runs. Essentially, it is a RAPS initiative, it is called Real Applications and Parallel Systems. Different large parallel codes are delivered to different computer companies. Yes, this is a benchmark suite. And the computer companies pay some money not to us but to a company in Germany which joins together all our codes and they distribute them to the companies. And then, all these companies can ask us come and give tutorials how we parallelised it and made it run efficiently on their machine. And it is one of the reasons we are interested, because we are going to buy a parallel machine next year. We want to see if our code can run efficiently on these machines. At the moment, the vendors are quite interested to learn about our model and they spend a lot of time optimising it for their architectures. *Informants 51*

C.4 Tuning within the group

[Ce511] Mostly on documentation, I mean if I have a colleague with a lot of experience in something I will ask him.

Informant 42

[Ce513] Normally, you look at your code and you see what is using the most time, in my case initially it was the inner product, and you optimise it as much as you can and then you start looking at documentation, and you start asking around about ...I need to do this: I need to sum over all processors one way or another what is the fastest way. And it took me until two days ago to find a way to do it faster, because shmem has a routine that does a global sum but it took me several months to find someone who could tell me which routine is best. Well, I was not going to use shmem with my own program but it turns out that there is a routine that does what my routine is doing, and then I am going to see whether this is faster and then I am going to use that ...it has to be faster. *Informant 44*

[Ce517] Courses are not the most efficient way to learn about tuning a specific machine. There are some courses which are really very very advanced or something like that but then if I just go to talk to someone who is good, who knows their stuff, just for half an hour will make more difference than a course.

Informant 42

[Ce800]

(Tuning consultation, first meeting among J (the expert) and T (the tuner)). T was able to see with the help of `apprentice` that the square root function was responsible for a large part of the execution time of the program. T told J that he was thinking of changing the algorithm so that the use of square root function could fall to half. J informed T about the existence of another library with cheaper mathematical functions. T and J discussed the structure of T's code. During all this time, T was drawing how the program arrays were distributed. He described how the data distribution affected the communication patterns in the code. J suggested that T should have a look at some sample codes using two dimensional domain decomposition. T said that he would read them and see what parts of them he could use. J explained the various ways to distribute data in the Craft programming model and why the order of the loops in a nested loop mattered for performance. T nodded but it was obvious that he had a problem understanding what J was telling him.

(Second meeting is joined by a second expert) T reported that the current version of the MPI_Allgather function took too much time. T worked in the same open office plan as G, one of MPI's developers. G, who happened to be around when T was experiencing this problem with his code, helped T by informing him that there was a new version of the MPI_Allgather function which had not yet been released with the official MPI implementation. G told T how to link his program with the new MPI library and the program performance improved. Also, G suggested to T a less expensive way of realising the same communications in his program without using MPI_Allgather. According to T, this alternative was too difficult for him to implement and even if he implemented it, it would be very difficult for him to maintain it: "In one month's time, I will forget what I did". Since E, a second expert, was attending that second meeting, T had to repeat explaining the structure of his program. On the basis of his explanation, J and E suggested how to best organise the communications. T had with him a display dump from `apprentice` in order to convey to the experts the performance of the program. The conversation focused on the most expensive routines. T complained that he could not find the alternative maths library in the manual pages. J said that definitely this time he was going to send the name of the library and the names of the functions to him by electronic mail. T said that `apprentice's` observations were too difficult to understand: "`apprentice` gives too many numbers! What can I do?" After that, E and J read apprentice's output. J started explaining to T what cache thrashing is. T asked what to do in order to avoid thrashing. T did not understand why the number of integer arithmetic operations described by apprentice was so high. J suggested that T had a look at another person's web pages to find an optimised routine for performing transpositions.

(Third meeting) The two experts discussed a problem that T was experiencing and could not find a solution immediately. It took them a while before they were able to suggest something. T said that he understood their suggestion in principle, but he was not sure whether he could realise it. Still, until that meeting T had not discovered where the library for the square root function was. J promised that this time he would send him all the necessary information. T said that he would like the experts to have a look at the code in order to discover where cache thrashing was happening. E explained why the loop order in nested loops mattered.

[Ce514] In the other case, I was aware of the standard BIAS routines, it was that I was wondering whether the extra copy would be worth it and after hearing from two people independently that it is worth it, I actually did it so it is not like ... normally, when you are optimising, you have a rough idea of what you want to do. But unfortunately, it is an experimental science so you have to try it to see if it actually works. Now, if it is something that will take a lot of time to try then you go to other people and ask if it will work. *Informant 44*

[Ce515] When we find something of interest to the "community", we often announce it in a paper or in email to our local users' groups. *Informant 17*

[Ce518] Being in a small porting group as above, we often discussed techniques between ourselves. For example, I developed the twin debugging technique that I mentioned, so I presented a seminar on it to our group. Also, every software package we worked on was fully reported, with reasons and source code changes, and we could read what had been done with other similar software. As for developing a detailed knowledge base, this would be useful, but one wonders about the cost-benefit trade off. *Informant 1*

[Ce520] Tuning knowledge is not something found in books or manuals but something that someone else may tell you. In the TLM code they have made many mistakes because they didn't know the particulars of the architecture and these mistakes are not going to be repeated now with the experience they gained. Parallelisation is not enough to do the job because even then you cannot get the performance out of that. *Informant 47*

[Ce521] Maybe somebody else would continue and I would need pointers to the most recent versions or I would leave them pointers or simply I would look at the most recent time of creation. I would enumerate all the parameters to him. The first thing would be to produce the document and sit down with him and give him a verbal explanation [1]. *Informant 45*

> [Ce504] Mostly for our own use. However, we have been talking about sharing these experiences with others. The main key is time. It takes a tremendous amount of time to document.
>
> *Informant 19*

> [Ce503] C rnp 21/3/95
> C rnp removed if statement from do loop
> C rnp modified do loop order
> C rnp halved work done in do loop and added transpose copy
>
> *Informant 40*

[Be628] is presented in Appendix B.

[Ce800] is presented in section C-4.

C.4.1 Somebody else's code

[Ce501] And in terms of documenting that is the design itself as such, because we are dealing, with people who won't have experience with parallel programming as such, so the documentation of the actual design is like a report, because we did write, started documenting in CSP. Which was fine, but nobody studied it besides Chris (the project leader) and me, so it was

[1]Ironically, this respondent had to delay moving over to his new job in order to finish with the implementation and the documentation of the project.

pointless to do it really. So we were back in writing a report of what is going on. You know do the usual functional mapping and the requirements mapping that we would do with the sequential code.

Informant 47

(Informant 51 has a interdisciplinary background in Computer Science and Meteorology. This, made parallelising and porting a large weather forecast code easier for him since he could understand what the scientific part of the application was doing. The well structured code and the ability to talk to colleagues made parallelisation easy, despite the fact that a lot time was spent initially in understanding the code.) [Ce2] Parallelising the code was not so difficult even-though there was only 10 pages of documentation for about 300,000 lines of code. It is quite well structured the program. The general layout and design have been designed by a few people and it's only one person who is responsible for the overall design so it is quite well designed. The project is new; it started in 1988 and it is written according to new programming standards. I have a master's degree in meteorology and I have always worked in numerical applications, so it was easy for me to understand the code. I know all the equations. I've never had any courses or read any books on parallel programming, so I talked to some of my colleagues, but I didn't find parallel programming difficult. It is really down to understanding the data distribution, the data layout and the data flow. That is what takes most of the time when you have a big code. To understand the serial code it takes a lot of time, you have to understand the data layout, what is globally used and what is only used locally. *Informant 51*

(Respondent 50 is an experienced parallel program tuner. He has not done parallel programming as such, rather he has been parallelising and optimising other people's code mostly on shared memory machines for the last five years. He approaches parallelising other people's code as a task where it is important to be able to isolate the pieces of code, where parallelisation can be introduced, from the rest of the code. On the other hand, it is vital to be able to obtain more feedback from the original developers of the code, when parallelisation depends on how the parallelised code can be integrated with the serial part of the application. Currently, he is employed by a large European institution to help them port and optimise their codes on a number of parallel machines.) [Ce3] The codes that I've come across tend to be very readable. Of course, there could always be some more documentation. I tend not to be interested in the whole of the code just the you know ...20 routines that you end up looking at, but if you want to parallelise code you probably need to understand a lot more sometimes. It may be located in three or four routines and then you look at those routines to understand what is going on. Sometimes, having a feedback, describing what a code does, helps during an integration problem or something like that. And you say ...oh that's good, I could parallelise that. I mean, it really depends on the application you are porting. For the overhead communication, you know, a very very simple model is the wall-clock time equals to computation plus communication. And you have to understand from the application you are porting, a parallelisation strategy, the cost of the communication, how many synchronisations you have to do, and you quickly understand if you are going to be successful in your application. You can start from the serial version and say I am going to work with the solver and just decide to parallelise that. And ignore the rest of the code. Which it runs still but it runs serial. You don't have to worry about where data is in some architectures. *Informant 50*

[Ce509] I didn't have any courses. I prefer not to follow courses. I prefer to read about it myself and it is so simple. What has to do with the message passing. If you are not using HPF or anything like that, if you use message passing you have only to know about send and receive and you have to know the syntax of these. Now, I am designing some new code and also when I did it the first time it was more reading the serial code, how it is organised and talking to somebody who wrote that code, "is this really independent so can anyone use it in parallel" and then to make sure that I am not using the same quantity at both latitudes. It was more design work than anything to do with message passing. Or HPF. For me parallel programming is to understand the dependencies in data and the data structures and the data flow in the program and when you have understood that you can cut everything into pieces and the last part is to really write the code that does the message passing. It is not more than 10% of the time. There were two people working here before I came and they were looking at different strategies to parallelise our models and trying it on simple models, 1,000 lines versions. They didn't leave any documentation apart from some papers they wrote. But they were here while I was here. We worked together here for a year. So I talked to them and it was more this way. *Informant 51*

[Ce4] I was working with a very old C program and it was so messy it was very difficult to find which parts to parallelise.

Informant 42

C.5 Case-study III: EPCC User Support Service

[Ce64] Q: I have just been reading the optimisation section of the CM Fortran manual (Version 1.0, March 1991). In here, it mentions various techniques one should adopt in writing code for the CM to speed up execution and which will not be necessary in future releases of the compiler. For example, unwinding serial loops. What I should like to know is if these techniques are now necessary - i.e. has the compiler been updated since 1991. If it has been updated, could you tell me where I can find some information on optimisation techniques for the current compiler version.

A: Since version 1.0 of the CMF compiler there have been a few releases: CMF 1.1, CMF 1.2, CMF2.1.1-2 and CMF2.2(CM-5 only). The optimisation notes are very out of date. There is now a complete manual on this topic "CM-5 CM Fortran Performance Guide", but as its title suggests is mainly of relevance to the CM-5 compiler and this has much more aggressive optimisations than the CM-200 compiler[2].

In particular, to comment on the "CM Fortran Optimisation Notes: Slice-wise Model", which I assume is what you have. For the CM-200 compiler: it is still true that communication will break a PE code block. The compiler is better at eliminating and reusing temporaries. Array sizes should be large for efficiency. The compiler is now better at serial optimisations but nowhere near as good as the CM-5 compiler, particularly for FORALL statements. So it normally pays to unwind this:

```
integer a(n,m)
cmf layout a(:serial,:news)
s = sum(a,dim=1)
```

Q6

[Ce63] Q: Do we loose a lot of efficiency if we have computation arrays which have dimensions (x, y, z) which are NOT powers of two? For example, would an array 1150X900 take a lot more computing time than a 1024x1024 array? I'm asking because it was the case in some earlier release of the CMF.

A: There may be a loss of efficiency but it is not easy to quantify. The first release of the CMF compiler had a fieldwise memory model and arrays were padded up to the next power of two on each dimension. Hence, performance of an array (300,300,300) would be similar to one of size (512,512,512). With the introduction of the slice-wise model (now the default) the situation changed. The array elements are now laid out on the processors (256 or 512 Weiteks) so that there are a multiple of 4 elements per processor. Hence, the padding is not so dramatic. However, an array that is not padded (extra garbage elements added) will be operated on more efficiently. You can tell if an array is padded by examining the output from the command call cmf_describe_array(array) which is described in the Utilities Library reference document. The situation is complicated by the fact that some communication routines and CMSSL can operate more efficiently on arrays that are powers of two. To be sure you have to try it for your particular application. *Q1*

[Ce65] I have faced recently a quite weird problem on the T3D and wonder whether somebody else ever noticed the same. I don't want to rise a fuss upon this case, but I'd be grateful if you can advise me something and maybe check my runs. I've noticed recently that the time per iteration has CHANGED for the SAME job, namely, now it is getting slower by a factor between 2 and 4! For example, this table shows several subsequent runs of the same problem on 128 PE's and the time spent:

[2]CM-200 was the machine used by the particular user.

```
run    time/iteration (s)         when
1          55.3                  26 Oct. 23:00
2          59.8                  1 Nov. 12:30
3          62.05                 1 Nov. 17:10
4          58.6                  2 Nov. 02:14
.......... but ???????? ............
5          209.4                 5 Nov. 12:23
```

The same happened on 6th of November with this job. Similar behaviour I have observed with another job which used 64 PE's. In this case, it was slowed down by a factor of 4-6 for all runs performed after 2nd of November. So, I cannot say whether there is any correlation with the number of PE's that would mean that this is a communication problem. A special investigation is probably needed which I cannot accomplish myself at the moment. The same for a 32 PE's job: 48 sec for 4 iterations some time ago and 115 sec for only 2 iterations now. In all this cases, the same source code has been used. But I have RECOMPILED the code after 1st of November with the same level of optimisation, so that this might be some compilation problem.

What do you think of that? It is extremely important to understand why this is happening since it affects the computer time.

A: The C compiler was updated on November 2nd and the Fortran compiler on September 4th. You might want to try compiling with the old compilers[3] and see if it makes any difference. If it does, let us know and we will try to find out what went wrong with the new compiler. This change cannot be due to the hardware upgrade because this was completed on October 19th and your performance change occurred on November 2nd. There were some changes to the YMP made around the beginning of November and the YMP has been very heavily loaded since then. The load should go down again when the J90 becomes available. You would have to be very IO dominated for YMP changes to make such a large change to your performance. Are you doing any IO in this loop? The key limit would be number of IO transaction rather than number of bytes. The other possibility is that you are using some other operation that involves the YMP. For example some of the timing functions operate locally on T3D nodes and others read the clock on the YMP. The CETEP code is known to require a great deal of communication so it is also possible that communication plays a part. From time to time PEs in the T3D show errors and are mapped out and replaced by spare PEs The communication performance will be slightly changed when this is in effect because messages have to route out to the spare PEs on the edge of the machine. If you put the following line in your scripts just before starting the parallel program you will be able to tell if this happened. (sleep 60; mppstat -a > mppstat.log) &. This command will wait for 60

[3]a directory was mentioned

seconds (to give the mpp job time to start) then run mppstat. This includes a report on every mpp job running at the time including information about redundant nodes. If the job runs slowly this will let you know if it was using any redundant PE at the time. *Q19*

[Ce58] A: If you are not using the shmem[4] routines, I cannot say what might be the problem. However, if you are using shmem_get or _put style routines, there may be a problem with cache coherency. Apprentice works by putting subroutine calls into your code at selected points, and if your code has not been explicitly flushing the cache before your communications, it is possible that your received data is being overwritten by cache lines before it is used. There is a shmem library cache flush routine which you should call explicitly to prevent this. *Q33*

[Ce61] I now have my code running on 32, 64 and 128 processors on the T3D but I cannot run on 256. The reason for this is that I use domain decomposition and run 256 identical copies of my code, each of which opens and reads input files. This means I need more than 256 file descriptors (taking into account the essential descriptors for standard I/O). However, UNICOS only allows me a maximum of 256. Would it be possible for the maximum number of file descriptors to be increased ?

A: In the next major release of UNICOS this limit will be under user control but an upgrade to this version is not scheduled here for a few months. The main bottleneck in doing I/O however is not the size of the I/O packets being transferred but the number of packets (ie. latency is a much bigger concern than bandwidth, due to the nature of initiating a system call on the YMP from the T3D). By buffering up your input/output data and say using one master processor to gather/scatter this data you may make your I/O more efficient. 256 pes all wishing to "talk" through the I/O gateway to the YMP cripples performance. It is much better/faster to use the internal communication network. *Q44*

[Ce60] Q: I and other members of my consortium would be very interested to know from you if you have any code, in any language (preferably Fortran) that actually runs at close to 38 Gflops on the whole machine (i.e. runs at close to 150 Mflops on one processor) or runs at 30 to 50 Mflops on one processor. We really could learn a lot from any chunk of code, however short, with such performance. *Q32*

[4]Using the optimised low-level communication routines, available on the T3d, is very difficult since users have to handle explicitly cache coherency and synchronisation issues. A large number of queries were concerned with how to get these routines work correctly. The problem is exacerbated because the available documentation is not always straightforward to understand or complete.

[Ce56] Q: I've used apprentice to see the performance of my code. The observations for my code are:
Detailed Description: The combined losses due to single instruction issue, instruction cache and data cache activity are estimated to be 10442550 μsec, or 8.33% of the measured time for this program. The combined expenditure of time for output routines is measured to be 711246 μsec, or 0.57% of the measured time for this program. The combined expenditure of time for input routines is measured to be 235400 μsec, or 0.19% of the measured time for this program. Navigation Information: The current selection may be improved by up to 11389196 μsec. The greatest aggregate improvement (6193874 μsec) may be gained by improving the children of EVOLUTION_STUDY. The synchronisation or work construct which offers the greatest potential improvement (3587656 μsec) is STMTS@26. This construct is a part of EVOLUTION.

With these comments, I don't understand if my program is good or not. It seems that I loose 8.33% of the time. I suppose it means that 8.33% of the time only one process works. But what I'd like to know is to how to improve my code. What is "children of a subroutine"? And what is the work construct? I've got no subroutines named stmts! At the line 26 of evolution there is a call to a random number generator which is not the one supplied by cf77. Can in-lining help the problem ? *Q45*

[Ce57] Q: I have a question concerned with the Apprentice tool. In the COSTS window, the bar chart always shows that Integer Adds is the dominant operation in the code. However, I expect that the code should perform floating point operations most of the time. Is there a conversion from floating point operation to integer operation inside the T3D? This will affect my judgement about how fast the code runs. In the OBSERVATIONS, two measures are printed:
1. floating point operations per second.
2. integer operations per second.
Usually, measure 2 is twice as large as measure 1. Only when I know which type of operation is dominant in the code, can I determine how fast the code actually runs. I noticed that after the new cf77 compiler was installed, the ratio of measure 2 and measure 1 has decreased for the same calculation.

A: If you have a fragment of code like

$$a(i,j,k) = b(i,j,k) * c(k)$$

This line contains only a single floating point operation but also contains instructions to do address calculations and load and store operations. The address calculations account for the integer instructions. In the worst possible case, the address calculation for a 3 dimensional array could take 4 integer operations. Integer operations can also occur in IF statements and DO loops. Floating point operations are not converted to integer operations. The ratio of iops to flops changes between compiler versions because as the compiler improves it is able to perform the address calculations using fewer instructions on average. Compilers still leave a bit to be desired. You can improve performance by making small changes to the source code like unrolling loops by hand or introducing scalar temporaries. *Q25*

(In the next case, the programmer was using a particular makefile mentioned in some local on-line document, which among other things contained the performance instrumentation enabling flag. The programmer did not even know the purpose of the flag:) [Ce59] A: The compilers were upgraded last Wednesday. The only bug we have reported on them at the moment are a few source files that crash if compiled with the -Ta flag enabled. Nobody else has reported differing results yet, but as we know there is a problem with -Ta; your problem may be related to this. The easiest way to check if this might be a compiler bug is to try with the old compiler which is still available under the name cf776102. The -Ta flag enables the apprentice performance tool. Unless it is badly broken in the new versions, it should not change the results. Apprentice inserts extra code that gathers runtime statistics and so it will have some impact on performance. You should always turn this flag off unless you are intending to run apprentice. (The -g flag has a much greater impact on performance because it disables all compiler optimisation. you should always remove -g unless you are using the debugger). When first declared, variables contain whatever values happened to be in memory at the time. Turning the -Ta flag on and off could change these values and therefore gives rise to your problem.

Q58b

Appendix D

Findings of VISPAT's empirical testing

D.1 List of tasks

This section reproduces the sheet of paper which listed the tasks which had to be performed by the evaluators during the empirical testing of VISPAT.

Navigation Display

The Navigation Display presents you with the sequence of phases for every process. Each process has a unique number which identifies it in the MPI_COMM_WORLD communication context. Execution can be replayed in various ways by using the appropriate buttons at the bottom of the display.

Every process consists of a single big phase called ring-shift at a first level. This single phase consists of several other subphases at a second level of expansion.

1. Start the replay of the program execution.

2. Find out what is the first subphase of the ring-shift phase of process number 0.

3. Replay the execution of the program at the second level of phase expansion.

4. Find out the exact time when the 3rd phase of process 3 started.

5. Change the time unit and the scale width before you proceed with task number 7.

6. Find the longest phase in the execution flow of process 3.

7. Make the display depict only user defined phases.

Communication Display

8. Animate the communications of 2 and process 3 in a continuous way.

9. Set the animation time to zero and make the communication display to show the third communication event of process 3. Which is the start time of this communication event?

10. Find the second last communication event of process number 1.

Statistics Display

11. Find the maximum waiting time of process 2.

12. Try to find out the instance number of the MPI_Wait call which corresponds to the above time.

Profile Display

13. Find out which process has the longest sum of MPI_Wait calls.

D.2 Qualitative and quantitative findings

This section is concerned with the evaluators utterances. Considered separately for each task, each evaluator's comments were interpreted and classified as negative or positive. Additionally, this section describes most of the unexpected evaluator actions and presents the evaluators requirements.

uc	used feature correctly
uw	used feature wrongly
nt	did not think to use feature at all
hu	helped to use feature
eu	experimented with using feature
gu	guessed correctly

Table D–1: Ways of interacting with the tool.

ne	negative comment
po	positive comment
un	unexpected user action
re	requirement
or	old requirement

Table D–2: Different kinds of utterances.

D.2.1 Task 1

Negative comments

1. *Evaluator 1:*"I don't see why animation is useful. It is confusing, that although the tool starts animation, we can see everything *a priori* by using the scrollbar."

2. *Evaluator 1*: "The tool doesn't give a precise idea about the current animation time."

3. *Evaluator 1*: "The fact that the current time is on the left is confusing. We are able to see the future but not the past."

4. *Evaluator 1*: "Animation is useless."

	Evaluators									
	1	2	3	4	5	6	7	8	9	total
ne	4	2	2	2	0	0	0	0	0	10
po	0	0	1	0	0	0	0	0	0	1
un	1	0	1	4	0	0	0	0	0	6
re	3	2	0	1	0	0	0	0	0	6
or	2	1	0	0	0	0	0	0	0	3

	Evaluators								
	1	2	3	4	5	6	7	8	9
go to the beginning	gc	gc	gc	gc	gc	hu	gc	hu	gc
rewind	gc	gc	gc	gc	gc	gc	gc	gc	gc
play	gc	gc	gc	gc	gc	gc	gc	gc	gc
stop	uc	uc	uc	uc	uc	uc	uc	uc	uc
step	uc	hu	uc	hu	uc	hu	uc	hu	uc
fast forward	gc	gc	gc	gc	gc	gc	gc	gc	gc
go to the end	gc	gc	gc	gc	gc	gc	gc	gc	gc

	Evaluators									
	1	2	3	4	5	6	7	8	9	total
alone	*	*	*	*	*	*	*	*	*	9
with some help										0
helped										0
alt.strategy										0

Table D–3: Quantitative findings from Task 1.

5. *Evaluator 2*: "It moves slowly. It is extremely slow. I would like to speed it up."

6. *Evaluator 2*: "I don't like the scrollbar."

7. *Evaluator 3*: "The scrollbar is difficult to use."

8. *Evaluator 3*: "The scrollbar keeps moving after I click on it."

9. *Evaluator 4*: "I don't recognise my program."

10. *Evaluator 4*: "I am surprised. I was expecting things to happen. Instead, I have an empty display."

Positive comments

1. *Evaluator 3*: "Apart from the way you can control the scrollbar, I can say I like it."

Unexpected user actions

1. *Evaluator 1*: "In the beginning, it is not clear that the tool does animation[1]."

[1]Evaluator 1 discovered the function of the scrollbar before he pressed *play*.

2. *Evaluator 3*: "I thought that what has been drawn has happened already."

3. *Evaluator 4*: "What is the ring-shift phase?"

4. *Evaluator 4*: "All processes start at the same time?!"

5. *Evaluator 4*: "I am surprised. I was expecting things to happen. Instead, I have an empty display. What has happened?"

6. *Evaluator 4*: "How can one understand that animation goes slowly? Does one have to know the maximum number of timesteps?".

Requirements

1. *Evaluator 1*: "I would like to have the moveable time cursor."

2. *Evaluator 1*: "I would like to have the current time in the middle, the past on the left and the future on the right."

3.*Evaluator 1*: "The only functionality I would like to have is to be able to zoom in and out in the navigation display."

4. *Evaluator 2*: "I'd like to be able to say what speed I would like it to play."

5.*Evaluator 2*: "The scrollbar should be able to set a new value for the current animation time once it is dragged to a new position."

6.*Evaluator 4*: "It would be nice to have some text above the hi-fi buttons for the people who don't understand the metaphor immediately."

D.2.2 Task 3

	Evaluators									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**total**
po	0	2	1	2	0	1	0	0	0	6
ne	0	3	2	1	0	0	0	0	1	7
un	0	2	2	2	2	2	1	3	1	15
re	0	6	1	1	1	0	1	0	0	10
or	0	2	1	1	0	0	1	0	0	5

	Evaluators								
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
unfold	uc	uc	hu	uc	nt+uc	uc	uc	uc	uc

	Evaluators									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**total**
alone	*			*	*	*	*		*	6
some help		*						*		2
helped			*							1
alt. strategy										0

Table D–4: Quantitative findings from task 3.

Negative comments

1. *Evaluator 2*: "I don't like the fact that the only indication of movement is the little flickering of the time indications at the bottom corners of the display."
Indeed the time unit has by default a very small value while evaluator 2's program comprised some very long phases. The default value of the time field should be set according to the average phase duration.

2. *Evaluator 2*: "The numbers and the text on the phase box sometimes get spoiled."

3. *Evaluator 2* : "Everything has disappeared. I don't know where all the info has gone. I was expecting that as everything was displayed before I would have everything chopped down.
Evaluator got an empty display because the MPI phases had not happened yet. He did not seem to understand that. He used the scrollbar and he found the phases.

4. *Evaluator 4*: "The ring-shift phase is not so interesting."
At that time, evaluator 4 did not know that he had other lower level phases.

5. *Evaluator 9*: "I don't like getting an empty display."

Positive

1. *Evaluator 2*: "Ok so you have here the exchanges. Good.[2]"

2. *Evaluator 2*: "I like here the way the name of the phase doesn't disappear when the start of the phases box disappears."

3. *Evaluator 4*: "It is very nice the fact that you can unfold only the selected phases if you want."

4. *Evaluator 4*: "That is good![3]"

5. *Evaluator 6*: "I like the metaphor for setting animation time to 0."

Unexpected user actions

1. *Evaluator 2.* Evaluator 2 thought that *unselect all* could toggle to select all.

2. *Evaluator 2*: "I don't understand what the empty space between the phases are."

[2]He liked the fact that he recognised some of the phases in his program.

[3]Evaluator 4 liked the fact that he could see the expanded phases after he rescaled the Navigation display.

3. *Evaluator 3.* Evaluator 3 invoked the phase description box and tried to click on a subphase there.

4. *Evaluator 3.* Having faced an empty display after a phase expansion, Evaluator 3 slided the scrollbar back to the left to see if there were any phases to the left of the current time[4].

5. *Evaluator 4*: "Something is wrong.[5]"

6. *Evaluator 4.* Evaluator 4 was sure whether or not the processes of his program were on the same processor.

7. *Evaluator 5.* Evaluator 5 chose *filter* and then *unfold.*

8. *Evaluator 5.* Evaluator 5 pressed the *play* button after animation had reached the end of file[6].

9. *Evaluator 6.* Evaluator 6 expanded the phases but since nothing was displayed on the screen she assumed that the execution can be animated only once. Also the message put her off but finally she completed the task.

10. *Evaluator 6.* Evaluator 6 forgot that the *go to the beginning* button can be used to set the animation time to 0.

11. *Evaluator 7.* Evaluator 7 asked about what the time in between two rectangles was.

12. *Evaluator 8.* Evaluator 8 was put off by the message and cancelled the unfold operation in order to select a phase first.

13. *Evaluator 8*: "I feel as nothing has happened.[7]"

14. *Evaluator 8.* Evaluator 8 needed help with the scrollbar.

15. *Evaluator 9*: "I lost the phases![8]"

Requirements

1. *Evaluator 2*: "I would like an automatic way to select all."

[4]This means that there is a need for an indication whether there are phases to the left or to the right of an empty frame in the Navigation display.

[5]Evaluator 4 was confused by the empty Navigation display after he expanded the upper level phases.

[6]An end of animation message is needed.

[7]When Evaluator 8 expanded the phases, the frame of the Navigation display was empty because the MPI calls did not appear immediately.

[8]Evaluator 9 did not see any phases after he expanded.

2. *Evaluator 2*: "I would like to have the function select all."

3. *Evaluator 2*: "I would like milestones so that animation can go past them and you can know where you are in the animation time."

4. *Evaluator 2*: "I would like to to have the line numbers of the code."

5. *Evaluator 2*: "It would be nice to scroll through a source code window.[9]"

6. *Evaluator 2*: "I would like the tool to hide the fact that my program consists of four processes sharing the same processor."

7. *Evaluator 3*: "I would like to select more than one phase at the same time."

8. *Evaluator 4*: "Colour would be usefull for the identification of the small phases."

9. *Evaluator 5*: "A message that animation has reached the end of file is needed."

10. *Evaluator 7*: "I would like to have a look at the source code."

D.2.3 Task 7

	Evaluators									
	1	2	3	4	5	6	7	8	9	total
po	0	0	0	0	0	0	0	0	0	0
ne	1	0	2	0	0	0	0	0	2	5
un	0	4	2	1	3	1	0	1	0	12
re	2	1	2	0	3	0	4	0	3	15
or	2	1	2	0	1	0	4	0	3	13

	Evaluators								
	1	2	3	4	5	6	7	8	9
filter	uc+hu	nt+hu	hu	uc	nt+hu	uc	hu	uc	uc

	Evaluators									
	1	2	3	4	5	6	7	8	9	total
alone				*		*			*	3
some help	*							*		2
helped		*	*		*		*			4
alt. strategy										0

Table D–5: Quantitative findings from task 7.

Negative comments

1.*Evaluator 1*: "I didn't like the stop message."

[9]He had written the program a long time ago and he could not remember it.

2. *Evaluator 3*: "I haven't got a clue."
This was outered when the stop sign appears.

3. *Evaluator 3*: "I don't understand the word filter. Does it mean out or in?"

4. *Evaluator 7*: "The feedback of the tool is a bit slow."

5.*Evaluator 7*: "I didn't like the blank space in the beginning when you unfold for example."

6. *Evaluator 9.* Evaluator 9 spotted the inconsistency between application and user defined.

7. *Evaluator 9.* Evaluator 9 didn't like that everything disappears.

Unexpected user actions

1.The stop message confused evaluator 9 and he tried to select more phases. *Evaluator 9*

2.*Evaluator 4.* The stop sign confused evaluator 4.

3. *Evaluator 6.* Evaluator 6 saw nothing on the display and thought that she had to go up one level.
She used the scrollbar to search for phases and finally understood what was happening.

4. *Evaluator 2*: "I could filter out to see what happens."

5. *Evaluator 2.* The evaluator clicked wrongly on the display menu.

6. *Evaluator 2.* Evaluator 2 thought that since there were not any phases on the Navigation display after the filter operation these phases had not happened yet.

7. *Evaluator 2* . Evaluator 2 thought that *filter* meant filter out.
Finally he found out that it meant filter in because he could select only one option. And finally he realised that his program did not have any user defined phases.

8.*Evaluator 3.* Evaluator 3 could not find *filter.* He tried *unfold* and even when he was told to keep the user defined phases he did not understand it. He thought that *filter* meant filter out.

9.*Evaluator 3.* Evaluator 3 tried to click on the empty area on the Navigation display in order to unselect the selected phases.
He said that he knew that he could do this in *idraw* and he thought that maybe he could do the same with VISPAT. He said that he had not liked that feature because in order to unselect only one phase he had to unselect all the phases at the same time. He did not discover the unselect all option.

10. *Evaluator 5.* Evaluator 5 did not think of pressing *filter* and said that he had to quit the tool to go to the upper level phases because these were the only ones that were user defined.

11. *Evaluator 5.* He tried to rescale the display to find the user defined phases.

12. *Evaluator 8.* Evaluator 8 clicked on *options* and because options had no menu underneath she got confused and said that she did not have any options. In reality, *options* had already been opened.

13. *Evaluator 5.* Evaluator 5 unselected manually all the phases.

Requirements

1. *Evaluator 1*: "I would like a message explaining more what has happened."

2.*Evaluator 1*: "*Filter* should mean filter in because users should be able to specify what they want to see."

3. *Evaluator 2*: "I would like the word application to be changed to user defined."

4. *Evaluator 3*: "I would like to be able to select more than one kind of phases to keep in."

5. *Evaluator 3*: "It would be nice to be able to filter more than one kind of phases."

Application vs user defined confused Evaluator 3 but he selected application because nothing else was applicable. He said that he was helped because he could select only one phase. After he got an empty display he thought that he had filtered all the phases out.

6. *Evaluator 5*: "It would be nice to be able to filter a phase with a particular name."

7. *Evaluator 5*: "Being able to make the tool display automatically the first phases would be nice."

8. *Evaluator 5*: "A message is needed."

9. *Evaluator 9*: "It would be nice to have two markers."

10. *Evaluator 7*: "It would be nice to use colour for the different phases."

11. *Evaluator 7*: "I would like to be able to fold phases up and down."

12. *Evaluator 7*: "I would like the line number along with the MPI calls in the popup window."

13. *Evaluator 9*: "I would like to be able to reverse all the user actions."

14. *Evaluator 9*: "I would like to be able to select and create my own collection of phases."

15. *Evaluator 9*: "It would be nice to be able to have an un-filter operation."

D.2.4 Task 9

	Evaluators									
	1	2	3	4	5	6	7	8	9	total
po	0	0	1	2	1	1	1	0	0	6
ne	1	0	2	1	4	0	0	0	0	8
un	0	0	2	0	1	3	0	0	0	6
re	1	0	1	1	2	0	0	0	1	6
or					1				1	2

	Evaluators								
	1	2	3	4	5	6	7	8	9
comms	uc	nt+hu	hu	hu	uc	nt+hu	nt+hu	nt+hu	uc
textual order	uw+hu	nt+hu	hu	hu	uc	hu	uw+hu	hu	hu
next	uc	nt+hu	hu	uc	uc	nt+hu	uc	hu	uc

	Evaluators									
	1	2	3	4	5	6	7	8	9	total
alone					*					1
some help	*								*	2
helped		*	*	*		*	*	*		6
alt. strategy		*	*							2

Table D–6: Quantitative findings from task 9.

Negative comments

1. *Evaluator 1*: "The order of the text in the display is not intuitive."

2. *Evaluator 3*: "I can't move this."

3. *Evaluator 3*: "I don't think that the textual order is natural."

4. *Evaluator 4*: "I think you expect that most of the text goes up."

5. *Evaluator 5*: "I don't think it is easy to handle the textual display in order to move it away from the Communication display."

6. *Evaluator 5*: "The order in the textual display is going to be confusing for the users."

7. *Evaluator 5*: "I don't see why time should be on the right. If the time were on the left the order of the presentation of the text would be more obvious."

8. *Evaluator 5*: "There is a lot of wasted space on the left because of the way information is presented."

Positive

1. *Evaluator 3*: "Now that I know about the textual order it is easier."

2. *Evaluator 4*: "That is very useful."

He meant the textual description

3. *Evaluator 4*: "That is a breakdown of what is going on in the communication display. Good."

4. *Evaluator 5*: "I like the communication event description display. It is very helpful."

5.*Evaluator 6*: "I like the processes layout because it is the way communication in MPI happen in contexts."

6. *Evaluator 7*: "I like that."
He meant the communication event description display.

Unexpected user actions

1. *Evaluator 3.* Evaluator 3 confused the concepts of event and end of event.

2.*Evaluator 3.* Evaluator 3 thought that a non-blocking event is always pictured in red.

3. *Evaluator 5.* Evaluator 5 thought that time in the communication event description display was missing.

4. *Evaluator 6.* Evaluator 6 confused red and green for blocking and nonblocking calls respectively.

5. *Evaluator 6.* Evaluator 6 guessed ok about the empty (waiting) and filled (non-blocking) blobs.

6. *Evaluator 6.* Evaluator 6 tried to perform this task using the Navigation display.

Requirements

1.*Evaluator 1*: " The order of the text presentation in the textual display should be the other way around."

2. *Evaluator 3*: "I would like to click on a event on the textual display and make the Navigation display to go to this event."

3.*Evaluator 4*: "I would like some relation to the Statistics display to know how much a process is waiting idle."

4.*Evaluator 5*: "The time on the textual display should be on the left."

5. *Evaluator 5*: "The textual display should resize as well."

6. *Evaluator 9*: "I would like to be able with the *next* button to go to the first communication event."

Appendix E

Publications

"Performance Evaluation and Visualisation with VISPAT". In Third International Conference on Parallel Computing Technologies. V. Malynsky et al. (eds). St. Petersburgh Russia, 1995. Lecture Notes in Computer Science v. 964, pp 180-186. Springer Berlin London.

"The Tuner's Workbench: A Tool to Support Tuning in the Large". Proceedings of the Workshop on Parallel Programming and Computation (ZEUS '95). P. Fritzson et al. (eds). Linkooping Sweden, 1995, pp 212-221. IOS Press.

"The Design of a Tool for Parallel Program Performance Analysis and Tuning". Proceedings of the Working Conference on Programming Environments for Massively Parallel Distributed Systems. K. M. Decker et al. (eds). Monte Verita Switzerland, 1994, pp 321–332. Springer Verlag.

"Applying Human Factors Techniques to the Design of a Tool for Parallel Program Performance Analysis and Tuning". Proceedings of the Sixth International PARLE Conference. C. Halatsis et al. (eds). Athens Greece, 1994, pp 749–752. Springer-Verlag.

"Performance Analysis Tools for Parallel Programs". Training and Education Series Report. Available from the World Wide Web at the URL: "http://www.epcc.ed.ac.uk".

"The Parallel Tools Consortium". Training and Education Series Report. Available from the World Wide Web at the URL: "http://www.epcc.ed.ac.uk".

"Computer Science Research in High Performance Computing". Training and Education Series Report. Available at the URL: "http://www.epcc.ed.ac.uk".

Bibliography

[1] A. Agarwal, R. L. Sites, and M. Horwitz. ATUM: A new technique for capturing Address Traces Using Microcode. In *13th Annual International Symposium Computer Architecture*, pages 119–127. IEEE CS Press, 1986.

[2] D. Allen, R. Bowker, K. Jourdaneis, J. Simons, S. Sistare, and R. Title. Data Visualization and Performance Analysis in the Prism Programming Environment. In N. Topham et al., editor, *Programming Environments for Parallel Computing*, pages 37–52. Elsevier Science, 1992.

[3] R. J. Anderson. Representations and Requirements: The Value of Ethnography in System Design. *Human-Computer Interaction*, 9:151–182, 1994.

[4] J. Annett, K. D. Dunkan, R. B. Stammers, and M. J. Gray. Task Analysis. *Training Information*, (6), 1971.

[5] R. Aydt. The Pablo Self-Defining Data Format. Department of Computer Science, University of Illinois, ftp-able from bugle.cs.uiuc.edu, July 1994.

[6] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: a Structured High-level Parallel Language and its Structured Support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[7] T. Ball and J. R. Larus. Optimally Profiling and Tracing Programs. In *19th Annual ACM Symposium on Principles of Programming Languages*, pages 59–69. ACM Press, 1992.

[8] P. C. Bates and J. C. Wileden. High Level Debugging of Distributed Systems: The Behavioral Abstraction Approach. *The Journal of Systems and Software*, (3):255–264, 1983.

[9] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore, and V. Sunderam. PVM and HENCE: Tools for Heterogeneous Network Programming. In J. J. Dongarra et al., editor, *Environments and Tools for Parallel Scientific Computing*, pages 139–153. Elsevier Science, 1993.

[10] N. Belkhatir and W. L. Melo. Towards an Integration of Software Product and Software Process Modelling. *Integrated Software Engineering*, 3(1):36–50, 1996.

[11] R. K. Bellamy. What does pseudo-code do – a psychological analysis of the use of pseudo-code by experienced programmers. *Human-Computer Interaction*, 9(2):225–246, 1994.

[12] T. Bemmerl. Programming Tools for Massively Parallel Supercomputers. In J. J. Dongarra et al., editor, *Environments and Tools for Parallel Scientific Computing*, pages 125–136. Elsevier Science, 1993.

[13] T. Bemmerl and P. Braun. Visualization of Message Passing Parallel Programs with the TOPSYS Parallel Programming Environment. *Journal of Parallel and Distributed Computing*, 18:118–128, 1993.

[14] T. Bemmerl and T. Ludwig. MMK – A Distributed Operating System Kernel with Integrated Dynamic Load-balancing. In H. Burkhart, editor, *CONPAR90, VAPP IV Lecture Notes in Computer Science, Vol. 457*, pages 744–755, Zurich, 1990. Springer-Verlag.

[15] T. Bemmerl and B. Ries. *Lecture Notes in Computer Science*, volume 794, chapter Performance Tools on Intel Scalable High Performance Computing Systems, pages 76–88. Springer, May 1994.

[16] A. Bequelin, J. Dongarra, A. Geist, and V. Sunderam. Visualization and Debugging in a Heterogeneous Environment. *Computer*, 26(6):88–95, 1993.

[17] L. M. Berlin. Beyond Program Understanding: A Aook at Programming Expertise in Industry. In *Empirical Studies of Programmers '93*, pages 6–25, 1993.

[18] G. Bjerknes and T. Bratteteig. *Computers and Democracy*, chapter Florence in Wonderland: System Development with Nurses. Avebury, 1989.

[19] G. Bjerknes, P. Ehn, and M. Kyng. *Computers and Democracy.* Avebury, 1989.

[20] W. Blume, R. Eigenmann, J. Hoeflinger, D. Padua, P. Petersen, L. Rauchwerger, and P. Tu. Automatic Detection of Parallelism: A Grand Challenge for High Performance Computing. CSRD Report 1348, University of Illinois at Urbana-Champaign, Centre for Supecomputing Research and Development, 1994.

[21] A. Bode and P. Braun. Monitoring and Visualization in TOPSYS. In G. Kotsis et al., editor, *Workshop on Monitoring and Visualization of Parallel Processing Systems*, pages 97–118, Moravanyn, CSFR, October 1993. Amsterdam Elsevier.

[22] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. In *Supercomputing '93*, pages 588–597, Portland, Oregon, November 1993.

[23] J. Bonar and E. Soloway. Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. *Human-Computer Interaction*, 1:133–161, 1985.

[24] R. Borgeest. Cyclic Performance Analysis for Parallel Systems. In *International Conference on Parallel and Distributed Processing, Techniques and Applications, PDPTA 95*, Georgia USA, 1995.

[25] R. Borgeest and B. Dimke. *TATOO User Manual.* Technical University of Munich, ESPRIT Project 6290 edition, June 1994.

[26] R. A. A. Bruce, J. G. Mills, and A. G. Smith. *CHIMP/MPI User Guide.* EPCC-KTP-CHIMP-V2-USER 1.2, June 1994.

[27] J. D. Bucher and K. L. Beck. Profiling on a Massively Parallel Computer. In *Second Joint International Conference on Vector and Parallel Processing, Lecture Notes in Computer Science 634*, pages 97–102. Springer-Verlag, September 1992.

[28] C. V. Bullen and J. L. Bennett. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human Collaboration*, chapter Groupware in Practice: Interpretation of Work Practices, pages 69–84. 1993.

[29] E. Carmel, R.D. Whitaker, and J.F. George. Pd and Joint Application Design: A Transatlantic Comparison. *Communications of the ACM*, 36(6):40–48, 1993.

[30] J. Carroll and T. Moran. Introduction to Special Issue on Design Rationale. *Human-Computer Interaction*, 6(3-4):197–200, 1991.

[31] T. Chetham. The Configuration Management of Families of Programs. In *Algorithms, Software, Architecture, Information Processing '92, Volume I*, pages 228–236. Springer-Verlag (North Holland), 1992.

[32] L. J. Clarke, R. A. Fletcher, S. M. Trewin, R. A. A. Bruce, G. Smith, and S. R. Chapple. Reuse, Portability and Parallel Libraries. Report 94-13, Edinburgh Parallel Computing Centre, 1994.

[33] E. J. Conklin. *Groupware '92*, chapter Capturing Organisational Memory, pages 133–137. Morgan Kaufmann, 1992.

[34] J. Conklin and M. L. Begeman. gIBIS: A Tool for All Reasons. *Journal of the American Society for Information Science*, 40(3):200–213, 1989.

[35] B. Curtis. By the Way Did Anyone Study Any Real Programmers? In *Workshop on Empicical Studies of Programmers*, pages 256–262, 1986.

[36] B. Curtis. Empirical Studies of the Software Design Process. In *Human-Computer Interaction INTERACT '90*, pages xxxv–xi, 1990.

[37] S. A. Dart. The Past, Present and Future of Configuration Management. In *Algorithms, Software, Architecture. Information Processing '92 Volume I*, pages 244–251. Elsevier-Science Publishers (North-Holland), 1992.

[38] L. Davies and S. Nielsen. *The Impact of Computer Supported Technologies on Information Systems Development*, chapter An Ethnographic Study of Configuration Management and Documentation Practices, pages 179–192. Elsevier Science, 1992.

[39] J. Dongarra, O. Brewer, J. A. Kohl, and S. Fineberg. A Tool to Aid in the Design, Implementation and Understanding of Matrix Algorithms for Parallel Processors. *Journal of Parallel and Distributed Computing*, 9:185–202, 1990.

[40] V. V. Dongen, C. Bonello, and C. Freehill. High Performance C Language Specification. Technical Report CRIM-EPPP-94/04-12, Centre de Recherche Informatique de Montreal, Quebec, Canada, April 1994.

[41] V. V. Dongen, G. Hurteau, A. Singh, E. Reiher, and H. Hum. A Performance Debugger for a Language for Data Distribution Primitives. In *Workshop on Environments and Tools for Parallel Scientific Computing*, Tennessee, 1994.

[42] P. Ehn and M. Kyng. *Computers and Democracy*, chapter The Collective Resource Approach to Systems Design, pages 17–57. Avebury, 1989.

[43] P. Ehn, B. Molleryd, and D. Sjogren. Playing in Reality. *Scandinavian Journal of Information Systems*, (2):101–120, 1990.

[44] M. Eisenstadt. Tales of Debugging from The Front Lines. In *Empirical Studies of Programmers*, pages 86–112, 1993.

[45] EPCC-KTP-CHIMP-CONC 1.2. *CHIMP Concepts*, June 1991.

[46] S. I. Feldman. Software Configuration Management: Past Uses and Future Challenges. In *3rd European Software Engineering Conference, ESEC '91, LNCS 550*, pages 1–6, Milan Italy, October 1991.

[47] N. Fletton. A Hypertext Approach to Browsing and Documenting Software. In *Hypertext II*, 1991.

[48] N. V. Flor and E. L. Hutchins. *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human Collaboration*, chapter Analyzing Distributed Cognition in Software Teams: A Case Study of Team Programming During Perfective Software Maintenance, pages 272–286. Morgan Kaufmann, 1993.

[49] C. Floyd. A Process-Oriented Approach to Software Development. In *Systems Architecture, Proceedings of the 6th European ACM Regional Conference*, pages 285–294, 1981.

[50] J. Foster and M. Munro. A Documentation Method based upon Cross-referencing. In *Conference on Software Maintenance*, 1987.

[51] J. M. Francioni, J. A. Jackson, and L. Albright. The Sounds of Parallel Programs. In *6th Distributed Memory Conference*, pages 570–77, Portland, OR, April 1991.

[52] J. M. Francioni and D. T. Rover. Visual-Aural Representations of Performance for a Scalable Application Program. In *Proceedings of the Scalable High Performance Computing Conference*, pages 433–44, Williamsburg, VA, April 1992.

[53] P. K. Garg, T. Q. Pham, B. Beach, A. Deshpande, A. Ishizaki, K. Wentzel, and W. Fong. Matisse: A Knowledge-Based Team Programming Environment. *International Journal of Software Engineering and Knowledge Engineering*, 4(1):17–59, 1994.

[54] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley. *PICL: A Portable Instrumented Communication Library-C Reference Manual.* Oak Ridge National Laboratory Oak Ridge,Tn, Technical Report ORNL/TM 11130 edition, 1990.

[55] I. Glendinning, V. S. Getov, A. Hellberg, R. W. Hockney, and D. J. Pritchard. Performance Visualisation in a Portable Parallel Programming Environment. In G. Kotsis et al., editor, *Workshop on Monitoring and Visualization of Parallel Processing Systems*, Moravany, CSFR, October 1992. Amsterdam Elsevier.

[56] S. Grabner and D. Kranzlmueller. *Lecture Notes in Computer Science*, volume 854, chapter Monitoring for Detecting Bugs and Blocking Communication, pages 66–75. Springer-Verlag, 1994.

[57] S. Graham, P. Kessler, and M. McKusick. gprof: A Call Graph Execution Profiler. In *SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126. ACM, 1982.

[58] Graphics, Visualisation, and Usability Center. Visualisation, Animation in Cluster Environments – PVaniM. Available from the World Wide Web at the URL: http://www.cc.gatech.edu/gvu/softviz/parviz/pvanim/pvanim.html, 1995.

[59] J. Grudin. The Computer Reaches out: the Historical Continuities in Interface Design. *ACM*, 1990.

[60] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, and J. Vetter. Falcon: On-line Monitoring and Steering of Large-Scale Paralllel Programs. Technical Report GIT-CC-94-21, College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, 1994. Published in Frontiers '95.

[61] R. Guindon. Designing the Design Process: Exploiting Opportunistic Thoughts. *Human-Computer Interaction*, 5:305–344, 1990.

[62] S.T. Hackstadt and A. Malony. Next-Generation Parallel Performance Visualization: A Prototyping Environment for Visualization Development. In *6th International PARLE Conference Lecture Notes in Computer Science 817*, 1994.

[63] O. Hansen and J. Krammer. A Tool for Optimizing Large Scale Parallel Applications. In *MASCOTS '95: Third International Workshop on Modelling Analysis and Simulation of Computer and Telecommunication Systems*, pages 293–96, Durham, N.Carolina, January 1995.

[64] M. C. Hao, A. H. Karp, A. Waheed, and M. Jazayeri. VIZIR: An Integrated Environment for Distributed Program Visualization. In *MASCOTS '95: Third International Workshop on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 288– 292, Durham N. Carolina, January 1995.

[65] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, 8(5):29–39, September 1991.

[66] B. R. Helm, A. D. Malony, and S. F. Fickas. Capturing and Automating Performance Diagnosis: the Poirot Approach. In *9th International Parallel Processing Symposium*, Santa Barbara California, April 1995.

[67] V. Herrarte and E. Lusk. Studying Parallel Program Behavior with Upshot. Technical Report ANL-91/15, Mathematics and Computer Science Division, Argonne National Laboratory, August 1991.

[68] B. Hill, J. Long, W. Smith, and A. Whitefield. Planning for Multiple Task Work - an Analysis of a Medical Reception Worksystem. In *INTERCHI '93*, 1993.

[69] J. M. Hoc. *Psychology of Computer Use*, chapter Analysis of Beginners' Problem-solving Strategies in Programming, pages 143–158. Academic Press Inc., 1983.

[70] K. Holtzblatt and H. Beyer. Making Customer-Centred Design Work for Teams. *Communications of the ACM*, 36(10):93–103, 1993.

[71] A. Hondroudakis. The Parallel Tools Consortium. Technology Watch Report 1995, Edinburgh Parallel Computing Centre, 1995.

[72] A. Hondroudakis and R. Procter. The Tuner's Workbench: An Environment for Supporting Tuning in the Large. In *ZEUS '95 Workshop on Parallel Programming and Computation*, pages 212–221, Linkooping Sweden, May.

[73] A. Hondroudakis, K. Shanmugam, and R. Procter. The design of a tool for parallel program performance analysis and tuning. In K. M. Decker et al., editor, *Programming Environments for Massively Parallel Distributed Systems*, pages 321–332. Birkhauser, April 1994.

[74] A. Hondroudakis, K. Shanmugam, and R. Procter. Performance Evaluation and Visualization with VISPAT. In *3rd International Conference on Parallel Computing Technologies*, pages 12–15, St. Petersburgh, 1995.

[75] J. A. Hughes, I. Sommerville, R. Bentley, and D. Randall. Designing with ethnography: making work visible. *Interacting with Computers*, 5(2):239–253, 1993.

[76] S. Ichimura, T. Kamita, and Y. Matsushita. A PilotCard-Based Shared Hypermedia System Supporting Shared and Private Databases. In *Conference on Organisational Computing Systems (COOCS '93)*, pages 59 –68, 1993.

[77] IEEE. Threads Extension for Portable Operating Systems (P1003.4a), 1990.

[78] R. B. Irvin and B. P. Miller. Multi-Application Support in a Parallel Program Performance Tool. *IEEE Parallel and Distributed Technology*, 2(1):40 – 50, Spring 1994.

[79] R. B. Irvin and B. P. Miller. A Performance Tool for High-Level Parallel Programming Languages. In K. M. Decker et al., editor, *Programming Environments for Massively Parallel Distributed Systems*, pages 299–313. Birkhauser, April 1994.

[80] J. A. Jackson and J. M. Francioni. Breaking the Silence: Auralization of Parallel Program Behavior. *Journal of Parallel and Distributed Computing*, 18:181–194, 1993.

[81] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modelling.* New York Willey, 1991.

[82] J.Greenbaum and M. Kyng, editors. *Design at Work: Cooperative Design of Computer Systems.* Lawrence Erlbaum, 1991.

[83] B. Johnson. @ATS: Assisted Technical Support. In *Poster Proceedings of the Fifth International World Wide Web Conference*, pages 91–100, 1996.

[84] P. W. Jordan, S. W. Draper, K. K. MacFarlane, and S. McNulty. Guessability, Learnibility and Experienced User Performance. In D. Diaper, editor, *People and Computers VI*, pages 237–245, 1990.

[85] M. Jorgensen. An Empirical Study of Software Maintenance Tasks. *Software Maintenance: Research and Practice*, 7:27–48, 1995.

[86] H. Kahney. *Psychology of Computer Use*, chapter Problem Solving by Novice Programmers. Academic Press Inc., 1983.

[87] S. M. Kaplan. Conversation Builder: An Open Architecture for Collaborative Work. In *Human-Computer Interaction, INTERACT '90*, pages 917–922, 1990.

[88] R. M. Karp and V. Ramachandran. *Handbook of Theoretical Computer Science*, chapter A survey of parallel algorithms for shared-memory machines. North Holland, 1990.

[89] E. Karrels and E. Lusk. Performance Analysis of MPI Programs, 1994. ftp-able from the URL: ftp://info.mcs.anl.gov/pub/mpi/misc/heath.ps.

[90] D. Kimelman and T. Ngo. Program Visualization for RP3: An Overview. Technical Report, IBM Research Division, T. J. Watson Research Center, 1990.

[91] D. Kimelman and G. Sang'udi. Program Visualization by Integration of Advanced Compiler Technology with Configurable Views. In J. Dongarra et al., editor, *Environments and Tools for Parallel Scientific Computing*, pages 73–84. North-Holland, 1993.

[92] R. Klar. Event-Driven Monitoring of Parallel Systems. In G. Kotsis et al., editor, *Proceedings of Monitoring and Visualization of Parallel Processing Systems*, Moravany, CSFR, 1992.

[93] D. Knuth. Literate Programming. *Computer Journal*, 27(2):97–111, 1984.

[94] D. E. Knuth. The Errors of TEX. *Software Practice and Experience*, 19(7):607–685, July 1989.

[95] J. A. Kohl and T. L. Casavant. The IMPROV Meta-Tool Design Methodology for Visualization of Parallel Programs. In *International Workshop on Modelling Analysis and Simulation of Computer and Telecommunication Systems*, 1993.

[96] J. A. Kohl and T. L. Cesavant. A Software Engineering Visualization Methodology for Parallel Processing Systems. In *Sixteenth Annual International Computer Software and Applications Conference (COMPSAC)*, pages 51–56, 1992.

[97] J. Kohn and W. Williams. ATExpert. *Journal of Parallel and Distributed Computing*, 18:205–222, 1993.

[98] V. A. Kryukov, A. V. Maksimov, A. K. Petrenko, and T. A. Polilova. Hierarchical Configuration Management. *Programming and Computer Software*, 20(2):55–68, 1994.

[99] V. Kumar and A. Gupta. Analysis Scalability of Parallel Algorithms and Architectures. TR–91–18, Department of Computer Science, University of Minessotta, 1991.

[100] M. Kyng. Scandinavian Design: Users in Product Development. In *Computer Human Interaction*, 1994.

[101] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–561, 1978.

[102] J. R. Larus. Efficient Program Tracing. *Computer*, 26(5):50–61, May 1993.

[103] E. Leu and A. Schiper. ParaRex: a Programming Environment Integrating Execution Replay and Visualization. In J.J. Dongarra et al., editor, *Environments and Tools for Parallel Scientific Computing*, pages 155–169. Elsevier Science Publications, 1993.

[104] K. G. Lockyer. *Introduction to Critical Path Analysis.* Pitman Publishing Co, 1964.

[105] R. Lougher and T. Rodden. Group Support for the Recording and Sharing of Maintenance Rationale. *Software Engineering Journal*, pages 295–306, November 1993.

[106] R. Lougher and T. Rodden. Supporting Long-term Collaboration in Software Maintenance. In *COOCS 93*, page 228 238, 1993.

[107] P. Luff, C. Heath, and D. Greatbatch. *Requirements Engineering*, chapter Work interaction and technology: The naturalistic analysis of human conduct and requirements analysis, pages 259–288. Academic Press, 1994.

[108] L. Macaulay. *Cooperative Requirements Capture: Control Room 2000*, chapter Requirements Engineering: Social and Technical Issues, pages 67–85. Computers and People. Academic Press, 1994.

[109] R. L. Mack and J. M. Burdett. *The Cognition of Experts: Empirical Approaches to Knowledge Elicitation*, chapter When novices elicit knowledge: Question-asking in designing, evaluating and learning to use software. Springer-Verlag, 1991.

[110] A. Maclean, R. Young, V. Bellotti, and T. Moran. Questions, Options and Criteria: Elements of Design Space Analysis. *Human Computer Interaction*, 6(3 and 4):201–250, 1991.

[111] A. MacLean, R. M. Young, and T. P. Moran. Design Rationale: The Argument behind the Artifact. In *CHI'89*, pages 247–252. ACM, May 1989.

[112] K.H. Madsen and P.H Aiken. Experiences Using Cooperative Interactive Storyboard Prototyping. *Communications of the ACM*, 36(6):57–66, 1993.

[113] J. Malard. MPI: A Message Passing Interface Standard; History, Overview and Current Status. Technology Watch report, Edinburgh Parallel Computing Centre, 1994.

[114] A. D. Malony, D. H. Hammerslag, and D. J. Jablonowski. Traceview: A Trace Visualization Tool. *IEEE Software*, 8(5), 1991.

[115] W. F. McColl. BSP Programming. In G. Bleliocj, editor, *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*, 1994.

[116] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard.* July 1994.

[117] B. P. Miller. What to Draw? When to Draw? An Essay on Parallel Program Visualization. *Journal of Parallel and Distributed Computing*, 18(265–269), 1993.

[118] B. P. Miller, J. M. Cargille, R. B. Irvin, K. Kunchithapada, M. D. Callaghan, J. K. Hollingsworth, K. L. Karavanic, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. available from the World Wide Web at the URL: http://www.cs.wisc.edu/ paradyn/papers.html, 1995.

[119] B. Mohr. SIMPLE: A Performance Evaluation Tool Environment for Parallel and Distributed Systems. In *Proceedings of the 2nd European Distributed Memory Computing Conference, EDMCC2 LNCS 487*, pages 80–89. Springer, April 1991.

[120] B. Mohr. Standardization of Event Traces Considered Harmful or is Implementation of Object-Independent Event Trace Monitoring and Analysis Systems Possible? In *Proceedings of the CNRS-NSF Workshop on Environments and Tools for Parallel Scientific Computing*, pages 103–24, St. Hillaire du Touvet France, 1993. Elsevier.

[121] B. Mohr, D. Brown, and A. Malony. *Lecture Notes in Computer Science v. 854*, chapter TAU: A Portable Parallel Program Performance Analysis Environment for pC++, pages 29–40. Springer-Verlag, 1994.

[122] B. Nardi and J. Miller. Twinkling lights and nested loops: distributed problem solving and spreadsheet development. *Int. Journal of Man Machine Studies*, 34:161–184, 1991.

[123] B. Nardi and J. R. Miller. The Spreadsheet Interface: A Basis for End User Programming. In D. Diaper et al., editor, *Proceedings of the IFIP TC Third International Conference on Human-Computer Interaction*, pages 977–983. North-Holland, 27-31 August 1990.

[124] B. A. Nardi and J. A. Johnson. User Preferences for Task-specific vs Generic Application Software. In *Computer Human Interaction*, pages 392–398, 1994.

[125] P. Naur. *Program Development Studies based on Diaries*, chapter Phsychology of Computer Use, pages 159–170. Academic Press Inc., 1983.

[126] J. Nielsen. *Designing and Using Human-Computer Interfaces and Knowledge Based Systems*, chapter Usability engineering at a discount, pages 394–401. Elsevier Science Publishers, 1989.

[127] J. Nielsen. Paper versus Computer Implementations as Mockup Scenarios for Heuristic Evaluation. In D. Diaper, editor, *INTERACT 90*, pages 315–320, 1990.

[128] J. Nielsen. *Usability Engineering*. Academic Press, 1993.

[129] J. Nielsen and R. Molich. Heuristic Evaluation of User Interfaces. In *Computer Human Interaction*, pages 249–256, 1990.

[130] W. Obeloeer, H. Willeke, and E. Maehle. Performance Measurement and Visualization of Multi-Transputer Systems with DELTA-T. In G. Haring et al., editor, *Proceedings of the Workshop on Visualization of Parallel Systems*, pages 119–144. North-Holland, 1993.

[131] Zave P. An Operational Approach to Requirements Specification for Embedded Systems. *IEEE transactions on Software engineering*, SE-8(3):250–269, 1982. Reprinted in Gehani N and McGettick A.D 1986 Software specification and techniques Addison Wesley.

[132] C. Pancake. Results of user surveys conducted on behalf of Intel Supercomputer Systems Division, two divisions of IBM Corporation and CONVEX Computer Corporation. 1989-1993.

[133] Cherri M. Pancake and Curtis Cook. What Users Need in Parallel Tool Support: Survey Results and Analysis. CSTR 94–80–3, Oregon State University, Department of Computer Science, 1994.

[134] S. Poinson, B. Tourancheau, and X. Vigouroux. Distributed Monitoring for Scalable Massively Parallel Machines. In J. J. Dongarra et al., editor, *Environments and Tools for Parallel Scientific Computing*, pages 85– 101. Elsevier Science Publications, 1993.

[135] N. Pressman. *Software Engineering: A practitioner's approach.* McGraw Hill, 1992.

[136] S. Prestwich and A. Kusalik. Programmer-Oriented Parallel Performance Visualisation. TR 96-01, Department of Computer Science, University of Saskatchewan, Saskatchewan, February 1996.

[137] D. A. Reed. *Computer Performance Evaluation, Lecture Notes in Computer Science v. 794*, chapter Experimental Analysis of Parallel Systems: Techniques and Open Problems, pages 25–51. Springer Verlag, 1994.

[138] D. A. Reed, R. D. Olson, R. A. Aydt, T. M. Madhyasta, T. Beckett, D. W. Jensen, B. A. Nazief, and B. K. Totty. Scalable Performance Environments for Parallel Systems. In *6th Distributed Memory Computing Conference*, pages 562–69. IEEE Computer Society Press, 1991.

[139] C. Reichenberger. Concepts and Techniques for Software Version Control. *Software-Concepts and Tools*, 15:97–104, 1994.

[140] H. Richardson. High Performance Fortran: History, Overview and Current Status. Technology Watch Report 1994, Edinburgh Parallel Computing Centre, 1994.

[141] J. Rieman. The Diary Study: A Workplace-Oriented Research Tool to Guide Laboratory Efforts. In *INTERCHI '93*, pages 321–326, 1993.

[142] B. Ries, R. Anderson, W. Auld, D. Breazeal, K. Callagham, E. Richards, and W. Smith. The Paragon Performance Monitoring Environment. In *Supercomputing '93*, pages 850–59, Portland, November 1993.

[143] R. S. Rist. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Human-Computer Interaction*, 6:1–46, 1991.

[144] D. Rover. Performance Evaluation: Integrating Techniques and Tools into Environments and Frameworks. In *Proceedings of Supercomputing '94*, pages 277– 78, 1994.

[145] D. T. Rover and C. T. Wright. Visualizing the Performance of SPMD and Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:129–146, 1993.

[146] Bodker S., Ehn P., Kammersgaard J., Kyng M., and Sundblad Y. *Computers and Democracy*, chapter A UTOPIAN Experience: On Design of Powerful Computer-Based Tools for Skilled Graphic Workers, pages 251–278. Avebury Pub. England, 1987.

[147] S. R. Sarukkai and D. Gannon. SIEVE: A Performance Debugging Environment for Parallel Programs. *Journal of Parallel and Distributed Computing*, 18:147–68, 1993.

[148] S. Sato and Tatsuro Murakami. Supporting Collaboration with Loose Relationship. In *Conference of Organisational Computing Systems, COOCS '93*, pages 52–58, 1993.

[149] L. T. Schroath. Configuration Management for Software Tests. *Hewlett-Packard Journal*, 44(3):53–59, 1993.

[150] P. Seaton and T. Stewart. Evolving Task Oriented Systems. In *Computer Human Interaction*, pages 463–469, 1992.

[151] H.E. Sengler. *Psychology of Computer Use*, chapter A Model of the Understanding of a Program and its Impact on the Design of the Programming Language Grade. Academic Press Inc., 1983.

[152] K. Shanmugan and K. Tourlas. Application Engineering Tools for MPI and PUL. EPCC-SSP 94-01, EPCC, September 1994.

[153] J. Siddiqi, R. Osborn, C. Roast, and B. Khazaei. The Pitfalls of Changing Programming Paradigm. In *Empirical Studies of Programmers*, pages 219–231, 1996.

[154] A. B. Sinha and L.V. Kale. Projections: A Preliminary Performance Tool for Charm. In *International Parallel Processing Symposium*, 1993.

[155] E. Soloway. What to Do Next: Meeting the Challenge of Programming-in-the-Large. In *Workshop on Empirical Studies of Programmers*, pages 263–267, 1986.

[156] J. C. Spohrer, E. Soloway, and E. Pope. A Goal/Plan Analysis of Buggy Pascal Programs. *Human-Computer Interaction*, 1:163–207, 1985.

[157] J. T. Stasko. Tango: A framework and system for algorithm animation. *Computer*, September 1990.

[158] J. T. Stasko. The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report. Technical Report, GITGVU-95-03, Graphics, Visualization and Usability Center, 1995.

[159] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM Concurrent Computing System. *Parallel Computing*, 20(4):531–45, April 1994.

[160] L. G. Terveen, P. G. Selfridge, and M. D. Long. From Folklore To Living Design Memory. In *INTERCHI '93*, pages 15–21, 1993.

[161] L. G. Terveen, P. G. Shelfridge, and M. D. Long. Living Design Memory – Framework, Implementation, Lessons learned. *Human-Computer Interaction*, 10:1–37, 1995.

[162] H. Thimbleby. Experiences of Literate Programming using cweb (a variant of Knuth's PWEB). *Computer Journal*, 29(2):201–211, 1986.

[163] W. F. Tichy. Design, Implementation and Evaluation of a Revision Control System. In *6th International Conference on Software Engineering*, 1982.

[164] W. F. Tichy. RCS - A System for Version Control. *Software-Practice and Experience*, 15(7), 1985.

[165] S. Toledo. PERFSIM: A Tool for Automatic Performance Analysis of Data-Parallel Fortran Programs. In *5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia, February 1995. IEEE Computer Society Press.

[166] N. Tomov and K-J. Wierenga. Application Engineering Tools for MPI and PUL. EPCC-SSP 93-15, EPCC, September 1993.

[167] B. Topol, J. T. Stasko, and V. Sunderam. The Dual Timestamping Methodology for Visualizing Distributed Applications. GTI-CC-95/21, Georgia Technology Institute, Atlanta GA, May 1995.

[168] X. Vigouroux. Implementation of a Scalable Trace Analysis Tool. In K. M. Decker et al., editor, *Programming Environments for Massively Parallel Distributed Systems*, pages 315–320. Birkhauser, 1994.

[169] P. Wall and A. Mosher. Representations of Work: Bringing Designers and Users together. In *Participatory Design Conference '94*, pages 87–98, 1994.

[170] S. Whittaker and H. Schwarz. Back to the future: pen and paper technology supports complex group coordination. In *Computer-Human Interaction '95*, 1995.

[171] W. Williams, T. Hoel, and D. Pase. The MPP Apprentice Performance Tool: Delivering the Performance of the Cray T3D. In K. M. Decker et al., editor, *Programming Environments for Massively Parallel Distributed Systems*, pages 333–45. Birkhauser Verlag, 1994.

[172] S. Woolgar. *Requirements Engineering, Social and Technical Issues*, chapter Rethinking requirements analysis: Some implications of recent research into producer-consumer relationships in IT development, pages 201–216. Academic Press, 1994.

[173] J. C. Yan. Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers. In *27th Hawaii International Conference on System Sciences*, pages 625–33, Wailea, Hawaii, January 1994.

[174] C. Yang and B. P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *8th International Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[175] E. Zabala and R. Taylor. Maritxu: Generic Visualization of Highly Parallel Processing. In N. Topham et al., editor, *Programming Environments for Parallel Computing*, pages 171–180. Elsevier Science Publishers, 1992.

[176] E. Zabala and R. Taylor. Process and Processor Interaction: Architecture Independent Visualisation Schema. In J. Dongarra et al., editor, *Environments and Tools for Parallel Scientific Computing*, pages 55–71. North Holland, 1993.

[177] Q. A. Zhao and J. T. Stasko. Visualizing the Execution of Threads-based Parallel Programs. GIT-GVU-95-01, Graphics Visualization and Usability Center Georgia Institute of Technology, Atlanta Georgia, 1995.