View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by OPUS - Volltextserver Universität Passau

PolyAPM: Comparative Parallel Programming with Abstract Parallel Machines

Nils Ellmenreich

Mai 2004

Eingereicht an der Fakultät für Mathematik und Informatik
der Universität Passau
als Dissertation zur Erlangung des Grades
eines Doktors der Naturwissenschaften.

Betreuer / Advisor:
Prof. Christian Lengauer, PhD.
Universität Passau

Zweitgutachter / External Examiner:
Dr. John T. O'Donnell
University of Glasgow

Abstract

A parallelising compilation consists of many translation and optimisation stages. The programmer may steer the compiler through these stages by supplying directives with the source code or setting compiler switches. However, for an evaluation of the effects of individual stages, their selection and their best order, this approach is not optimal.

To solve this problem, we propose the following method. The compilation is cast as a sequence of program transformations. Each intermediate program runs on an *Abstract Parallel Machine* (APM), while the program generated by the final transformation runs on the target architecture. Our intermediate programs are all in the same language, Haskell. Thus, each program is executable and still abstract enough to be legible, which enables the evaluation of the transformation that generated it. This evaluation is supported by a cost model, which makes a performance prediction of the abstract program for a real machine.

Our project, PolyAPM, provides an acyclic directed graph – usually a tree – of APMs whose traversal specifies different combinations and orders of transformations. From one source program, several target programs can be constructed. Their run time characteristics can be evaluated and compared.

The goal of PolyAPM is not to support the one-off construction of parallel application programs. For the method's overhead to pay off, the project aims rather at supporting the construction and comparison of many similar variations of a parallel program and a comparative evaluation of parallelisation techniques. With the automation of transformations, PolyAPM can also be used to construct semi-automatic compilation systems.

Zusammenfassung

Eine parallelisierende Compilation besteht aus vielen Übersetzungs- und Optimierungsstufen. Der Programmierer kann den Compiler in diesen Stufen steuern, in dem er im Quellcode Anweisungen einfügt oder Compileroptionen verwendet. Für eine Bewertung der Auswirkungen der einzelnen Stufen, der Auswahl der Stufen und ihrer besten Reihenfolge ist der Ansatz aber nicht geeignet.

Um dieses Problem zu lösen, schlagen wir folgende Methode vor. Eine Compilation wird als Abfolge von Programmtransformationen betrachtet. Jedes Zwischenprogramm gehört jeweils zu einer *Abstrakten Parallelen Maschine* (APM), während das durch die letzte Transformation erzeugte Program für die Zielarchitektur bestimmt ist. Alle Zwischenprogramme sind in der Sprache Haskell geschrieben. Dadurch ist jedes Programm ausführbar und trotzdem abstrakt genug, um gut lesbar zu sein. Durch diese Ausführbarkeit kann die Transformation, durch die das Programm erzeugt wird, bewertet werden. Diese Bewertung wird durch ein Kostenmodell unterstützt, das eine Performance-Vorhersage des abstrakten Programms, bezogen auf eine reale Maschine, ermöglicht.

Unser Projekt PolyAPM liefert einen azyklischen, gerichteten Graphen – in der Regel einen Baum – aus APMs, dessen Traversierungen jeweils bestimmte Kombinationen und Reihenfolgen von Transformationen definieren. Aus einem Quellprogramm können verschiedene Zielprogramme erzeugt werden, deren Laufzeitverhalten bewert- und vergleichbar ist.

Das Ziel von PolyAPM liegt nicht in der Erzeugung eines einzelnen, parallelen Programms. Damit sich der zusätzliche Aufwand der Methode auszahlt, richtet sich das Projekt eher auf die Entwicklung und den Vergleich vieler, ähnlicher Variationen eines parallelen Programms und der vergleichenden Bewertung von Parallelisierungstechniken. Mit der Automatisierung von Transformationen kann PolyAPM dazu benutzt werden, halbautomatische Compilations-Systeme zu bauen.

Contents

Contents vii

1 Introduction **1**
- 1.1 Parallel Programming . . . 1
- 1.2 Software Engineering . . . 2
- 1.3 Abstract Machines . . . 3
- 1.4 PolyAPM . . . 3
- 1.5 Overview . . . 4

2 Preliminaries **5**
- 2.1 Haskell as the Implementation Language . . . 5
  - 2.1.1 Differences between Imperative Languages and Haskell . . . 5
  - 2.1.2 Short Introduction to Haskell . . . 6
- 2.2 Parallel Programming . . . 11
  - 2.2.1 Problem Domains for Parallel Programming . . . 11
  - 2.2.2 General Loop Parallelisation Techniques . . . 11
  - 2.2.3 Loop Parallelisation in the Polytope Model . . . 12
  - 2.2.4 Concepts for Performance Measurement of Parallel Programs . . . 14

3 The PolyAPM Framework for the Development of Parallel Programs **17**
- 3.1 Abstract Machines and Stepwise Refinement . . . 17
- 3.2 Abstract Parallel Machines . . . 18
- 3.3 PolyAPM . . . 18
  - 3.3.1 Machine Model . . . 19
  - 3.3.2 Structuring APMs into Graphs . . . 19
  - 3.3.3 Development Process . . . 20
  - 3.3.4 Abstract Machines and their Programs . . . 22
- 3.4 A Cost Model for PolyAPM . . . 24
  - 3.4.1 Incorporation of the Cost Model into PolyAPM Interpreters . . . 26
- 3.5 Dependence Analysis in Haskell . . . 29

4 Example of a PolyAPM Infrastructure **31**

4.1 Writing PolyAPM Programs . . . 31
4.2 The PolyAPM API . . . 34
4.2.1 Data Types . . . 34
4.2.2 SynAPM Functions . . . 35
4.2.3 SynCommAPM Functions . . . 36
4.2.4 SynDMAPM Functions . . . 37
4.2.5 An APM Example Program . . . 38
4.3 Adapting APM Programs to Target Architectures . . . 39
4.3.1 The mpi_apm Library . . . 39
4.4 Calibrating the Cost Model for send_msgs_a2a . . . 47
4.4.1 Determining the Computation Cost . . . 47
4.4.2 Before Determining the Communication Costs . . . 48
4.4.3 Measuring the Constant Communication Cost . . . 48
4.4.4 Measuring the Variable Communication Cost . . . 49
4.4.5 Confirming the Cost Prediction . . . 53
4.4.6 Determining the Cost Model Parameters . . . 54

5 Case Study I: Finite Differences 57
5.1 Initial Code Generation: The Synchronous Program . . . 58
5.2 The Tiled Program . . . 62
5.3 The Communicating Program . . . 63
5.4 The Distributed Memory Program . . . 65
5.5 The Distributed Memory Program with Relative Coordinates . . . 66
5.6 The C+MPI Program . . . 66
5.7 Benchmarking the C+MPI Program . . . 71

6 Case Study II: LU Decomposition 73
6.1 Problem Specification . . . 73
6.2 Parallelisation . . . 74
6.2.1 Dependence Analysis of LU . . . 74
6.2.2 First Schedule and Allocation (STM1) . . . 74
6.2.3 Second Schedule and Allocation (STM2) . . . 77
6.3 Initial Code Generation . . . 78
6.3.1 Evaluation of Two Alternative Programs . . . 81
6.4 First APM Transformation: Processor Tiling . . . 82
6.5 Second APM Transformation: Generation of Communications . . . 84
6.5.1 From Abstract Memory to Machine State . . . 85
6.5.2 Generating Point-to-Point Messages with genMsg . . . 85
6.5.3 Receiving Messages with updateMem . . . 86
6.5.4 Evaluating the Transformation . . . 87

6.6 Third APM Transformation: Memory Distribution . . . 88
6.7 The Results of the APM Transformations . . . 89
6.8 Generating C+MPI Target Code . . . 90
6.9 Benchmarking the C+MPI Target Code . . . 92

7 Evaluation of PolyAPM Based on the Case Studies 97
7.1 Experiences with the General PolyAPM Approach . . . 97
7.2 Experiences with our Implementation . . . 99
7.3 Evaluation Summary . . . 102

8 Related Work 103
8.1 Manual Parallel Programming . . . 103
8.1.1 Standardised Libraries for Parallel Programming . . . 103
8.1.2 Parallel Functional Programming . . . 104
8.2 Compilation Systems . . . 105
8.2.1 Commercial and Academic Compilers . . . 105
8.2.2 Parallelisation Systems in Research . . . 105
8.3 Abstract Machine Models . . . 105
8.4 Compiler Generation . . . 106

9 Conclusions 107
9.1 Summary . . . 107
9.1.1 Main Contributions . . . 107
9.1.2 Main Features of the PolyAPM Framework . . . 108
9.2 Who Should Use PolyAPM? . . . 109
9.3 Outlook . . . 109

A LooPo Specifications 111
A.1 LooPo Specification for STM1 . . . 111
A.2 LooPo Specification for STM2 . . . 112

Bibliography 113

Chapter 1

Introduction

Despite some pessimistic forecasts, recent years have still seen a growth in the number of transistors per square inch and hence in computing power according to Moore's law [Moo03]. But even with standard workstations matching mainframes from ten years ago, there still exist communities that need even more number crunching resources. Science and engineering projects analyse experimental data or perform simulations with mathematical models. Request for almost unbounded processing power exists in fields where continuous systems are modelled with discrete methods. More recently, large databases, data warehousing, financial statistics and other areas have entered the market of multi-processing. All these application areas are pursued by organisations with the financial resources to purchase dedicated parallel computers. But during the last 10 years, the availability of commodity hardware for multi-processor machines has lowered the entry level price and thus attracted new user groups.

Even though parallel programming has been done for several decades, it remains difficult. Compared to sequential programming, the process contains added complexity: independent parts of algorithms have to be determined and assigned to different processors, intermediate results have to be communicated among the processors and, in the case of distributed memory machines, input data has to be distributed before the computation and the results have to be collected after it. All of these steps have to be done very efficiently as one wants the additional processing power to be used as much as possible on the original algorithm and not on the parallel overhead.

We present a new approach for programming parallel machines: the programming process is decomposed into a sequence of transformations. Input and output of these transformations are programs for abstract parallel machines. A stepwise refinement process is performed to get a parallel program, and the result of each refinement step can be executed on an abstract machine.

1.1 Parallel Programming

To motivate our work, we present a short overview of the current state of affairs in parallel programming. Traditionally, the user base of parallel computers has been small. The few and very expensive machines located at research labs and big corporations were used by specialised scientists. Usually, they were not computer scientists, but physicists, meteorologists, chemists and others. As Fortran is still the predominant sequential programming language in these areas, it is not surprising that its parallel variants are also wide spread.

Fortran was either used with vendor specific libraries or with dedicated compilers for parallel dialects of the language [Ame92, HPF97]. These libraries tie a program to a specific architecture of a specific vendor, so that a new hardware generation or a vendor change results in the need to adapt the code. On the other hand, parallel compilers were just not good in extracting the

parallelism and were often restricted to shared memory machines.

The library situation was improved when in the late 80s vendor-independent message passing libraries were introduced that have a fixed *application programmer's interface* (API) for all implementations on different architectures. PVM [G+94] and MPI [DHHW93] are the most widely spread ones. It is up to the hardware vendors to provide efficient implementations of these standardised APIs on their platforms.

But even with these libraries, parallel programming is tedious and error-prone. Especially getting the communications right is a complex task. Communication is often done in a send/receive-manner, i.e., the sending program issues a send library call to transmit a message while at the same time the receiving program must perform a matching receive call to obtain the message. The success of the receive may be necessary to proceed with the calculations, so that the receiving call *blocks* until the message was received. A small mistake in the program structure can easily lead to a deadlock where two processors mutually wait on each other. Debugging such deadlocks is difficult as debuggers only deal with single programs, but to address communication problems the interaction of a set of programs must be observed.

These difficulties do not exist on shared memory machines, but the common access of all processors to the memory is a bottleneck that prevents unlimited scaling. While programming such machines is easier, the degree of parallelism remains limited.

Hardware vendors tried to combine the ease of shared memory programming with scalability of distributed memory by inventing *Non-Uniform Memory Access* (NUMA) machines. These machines are clusters of tightly connected shared memory machines for which a software layer provides a shared memory programming style for the whole machine. Memory access times differ depending on whether the data is in local or remote memory, thus the term *non-uniform access*.

Today, a customer mainly chooses from three architectures: shared memory, distributed memory and NUMA. Even with shared memory machines without communications, parallel programming remains difficult. If compiler support is required, the choices are either a Fortran or C Compiler with OpenMP [Boa00] support for shared memory, or a High Performance Fortran (HPF) [HPF97] compiler for distributed memory. But in practise most parallel programs today are probably written in Fortran or C with MPI calls.

To sum up, the difficulties when writing a parallel program for a distributed memory machine are: parallelism has to be identified in the algorithm and adopted to the programming model, data has to be distributed and recollected and communication of intermediate values has to be devised. This is all done on top of the normal task of implementing the algorithm itself. In manual programming, all of the above tasks are often done in one huge step, after which a lot of debugging is needed until the program is deemed correct. We will now look at methods helping to ease this process.

1.2 Software Engineering

Structured programming as we know it today originated from several advances made by some of the most distinguished pioneers of computer science. It became necessary by the increasing complexity of software during the 1960s. The concept of "levels of abstraction" was introduced by E.W. Dijkstra to describe a layered system in which upper layers must not access details hidden in lower layers [Dij68]. The idea is to concentrate just on the concepts necessary for each particular layer.

D.L. Parnas then pioneered work on *information hiding* [Par72], which laid the basis for a further structuring of software development. He later used the term *program families* for the fact that software development is rarely linear, but exhibits a tree structure of program variations with slightly different objectives [Par76].

The deconstruction of the programming process into a series of transformations that – step by step – lead to the final program was coined by N. Wirth as *stepwise refinement* [Wir71].

Based on the above work, the discipline of *software engineering* evolved within computer science to help making programming a craft rather than an art form. Among the later inventions during the 1970s were abstract data types, the separation of different program parts into modules and software reuse.

Starting in the 1980s, software engineering focused more on the management of large projects. Design was formalised by several communities, with a particularly active community that paid special attention to the application to object-oriented software development [GHJV95].

1.3 Abstract Machines

A well known tool in the development of language implementations is the *abstract machine*. It serves as a simplified model of a real machine and can be used to simulate the step-by-step execution of a program. Depending on the purpose, very few to almost all technical details of a real machine may be missing. In practise, many compiler implementations use abstract machines as targets for their intermediate code transformations before generating the target code. S. Diehl, P. Hartel and P. Sestoft compiled a comprehensive bibliography of abstract machines in various programming language implementations [DHS00].

The concept of stepwise refinement complements abstract machines in that the refined intermediate programs are still incomplete, yet they need to be written in a form that gives them an operational semantics. This semantics is provided if the refined programs are designed for dedicated abstract machines. The approach gives rise to a sequence of refinements, where the input and output of each refinement step is an abstract machine program. At the end of the sequence, the last abstract machine coincides with a real machine.

1.4 PolyAPM

Our work aims to improve the situation of parallel program development by utilising software engineering concepts for a manual or semi-automatic transformational program generation with profiling support for transformation selection. In an ideal world, a user can implement an algorithm without having to care about hardware details to improve performance. Such details include the specifics of a parallel machine. Unfortunately, as of today, the techniques for automatic parallelisation are not capable to get maximum performance for arbitrary programs. Therefore, we have chosen an approach that organises the transformation of the user's input program to a parallel target program by splitting the process into a sequence of transformations. This process relies on the concept of stepwise refinement to be able to apply many simple transformations with isolated objectives. Out work supports both, manual and possible automatic application of transformations.

Using this approach, in one application scenario an experienced programmer just has to deal with the transformations that cannot yet be automated, thus yielding a semi-automatic compilation. But even if all transformations are done by hand, the entire programming process is simplified by the division into small steps and the ability to immediately evaluate the transformation effect. This helps to narrow down errors and avoids the difficulty of an all-in-one-step approach that is still popular practise.

In case transformations have alternatives, there is the difficult task to determine the best one. To tackle this problem, we provide help by evaluating the effects of transformations. Recall that the result of a transformation, the intermediate program, is designed for an abstract machine. This machine exhibits just as much hardware detail as required by the program. We implement

interpreters for these machines so that the intermediate programs can in fact be executed with real input. The profiling data of such program runs can give insight in the effects of a transformation. Furthermore, we devise a *cost model* with which one can predict the program's performance on a real machine, based on the profiling data of the program's execution on an abstract machine. So, instead of writing final target code in one big effort and just using trial and error for code optimisation, a programmer may use PolyAPM to get a profiling directed transformation sequence which produces well performing final target code.

1.5 Overview

In Chapter 2, functional programming and the state of the art in parallel programming, together with some basic definitions, are introduced. Our PolyAPM framework for the development of parallel programs by stepwise refinement is presented in Chapter 3. It is followed by the detailed description of an example implementation in Chapter 4. This implementation is subject of two case studies for 2D finite differences (Chapter 5) and LU decomposition (Chapter 6). Based on the experiences made in the case studies, we evaluate PolyAPM in Chapter 7. The connection to related work is made in Chapter 8. The work is concluded in Chapter 9.

Chapter 2

Preliminaries

This chapter presents material that is a prerequisite for understanding the content of the following chapters. We start with a short presentation of functional programming with particular focus on the language Haskell (Section 2.1). Then, we present an overview of parallel programming with an emphasis on the areas that are needed for this work (Section 2.2).

2.1 Haskell as the Implementation Language

This section explains the differences between Haskell and imperative languages and provides a motivation for using Haskell in this work. A short introduction to Haskell is given so that a reader unfamiliar with the language is able to follow the code examples in later chapters.

2.1.1 Differences between Imperative Languages and Haskell

The world of programming languages is divided in two main groups: imperative and declarative languages. They have a different computational model as their base. Imperative languages focus on *how* things are computed. They consist of a series of commands that are executed in a certain order. In contrast to this, declarative languages focus on *what* is to be computed. They consist of a set of declarations of no particular order. Inside declarations, references to other declarations may be made. The execution of a program is a request to evaluate a specific declaration; other declarations that are needed for this are automatically evaluated.

The imperative model is machine oriented. Programs comprise a sequence of commands and a collection of data being stored in the computer's memory. A processor retrieves the commands from memory to execute them. The programming model was very much influenced by Alan Turing and his *Turing Machines* [Tur37]. Today, imperative languages are the most widely used. They have evolved into several categories ranging from simple procedural languages (Fortran, C, Pascal, etc.) to object-oriented languages like Smalltalk, C++, Java and C#. Traditionally, these languages provide comparatively fine-grained control over the machine. Memory for data structures needs to be allocated and deallocated, memory content may be destructively overwritten and it is clearly defined how a computation is performed. Modern languages have raised the level of abstraction in that they disallow direct references to memory (pointers) and use automatic garbage collection by a run time system. This evolution trades machine control and often efficiency for ease of programming.

Declarative languages are further divided into logical and functional languages. Haskell is a functional language. A Haskell program consists of a set of functions, that may call each other. The programmer does not specify an order in which calculations take place, but rather the function

in whose result he is interested. A run time system takes care of evaluating all expressions needed to compute this result. The underlying computational model is the *Lambda Calculus* [Bar84].

Functions are evaluated and their results may or may not be bound to names. These names can be used interchangeably with the value they denote. Therefore, the name-value binding

$$name = expression$$

is an equation in the mathematical sense. A prerequisite for the interchangeability is the absence of *side-effects*. The only result of a function is the returned value. There cannot be any assignment to a global variable as it happens in imperative languages.

This property can be used in proofs: either side of an expression can be substituted by the other. This proof technique is called *equational reasoning*.

There is a correspondence between this binding and the variable assignment in imperative languages. Assignments have, however, an inherent right to left direction: evaluate the expression on the right and store the result in the memory location denoted by the name on the left. This means that in imperative languages, the same name (i.e., memory location) can be re-assigned a new value by overwriting the old one. Equational reasoning does not work here, as one name may have several right hand sides and it is not always possible to deduce which one is current. There are some functional languages that incorporate imperative features [OCa, MTHM97]. In the functional programming community, they are called *impure* to contrast them to *pure* languages like Haskell.

The next subsection features some Haskell type system properties that enable a smooth integration of user-defined types. We use these types to represent abstract programs within Haskell. Interpreters for those programs then use these representations.

For our work, we choose Haskell for several reasons: programs for abstract machines can be embedded easily into the language using algebraic data types, corresponding interpreters for user-defined languages are simple to write. We also believe that Haskell is well suited for the implementation of algorithms of our problem domain. Finally, the suitability of a functional language for correctness proofs is of importance for developers of new transformations.

2.1.2 Short Introduction to Haskell

Haskell is a purely functional language that was defined by a committee in 1987 and has since then gone through several revisions until it reached a mature state with Haskell 98 [PJHe99]. Today, most research work which requires a purely functional language is done in Haskell. Several computer science departments use Haskell as one of the first programming languages to teach the fundamentals of programming. Haskell is named after the 20th century mathematician Haskell B. Curry.

All existing implementations are non-commercial and maintained by research groups. The most prominent one is the Glasgow Haskell Compiler (ghc) [GHC], a compiler and interpreter suite originally developed by the functional programming group at the University of Glasgow. Today, the core development takes place at Microsoft Research in Cambridge, UK.

Basic Concepts of Haskell

We now present a short overview of the language concepts of Haskell with a focus on what we need in our work. Programs are collections of functions, possibly separated into modules. The order of the functions within a source file does not matter. A striking feature of the Haskell syntax is its conciseness. Programs tend to be comparatively short with only little syntactic overhead. A "Hello World" program is a one-liner:

```
main = putStr "Hello World\n"
```

The execution of any Haskell program starts at the main function that calls other functions to compute the program's result. String constants are enclosed by quotation marks. The start and end of a block, such as a function body, are usually marked by the indentation level. Alternatively, one could use curly brackets like in C, but this is uncommon.

Parameters to functions are just put after the function name, without any parentheses. Function definitions start with the function name without any special keyword. The following function `solve_eq` returns the two roots of the polynomial $x^2 + px + q$, as defined by

$$x_{1/2} = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$$

without checks whether only one or no roots exist:

```
solve_eq p q = (-p/2 - root, -p/2 + root)
               where root = sqrt(p^2/4 - q)
```

The function takes two formal parameters, `p` and `q`, that are floating point numbers. It returns a pair of the two solutions in parentheses. To enhance readability, the computation of the square root is moved outside the pair and given the local name `root`. The type of this function can be defined as `Float -> Float -> (Float,Float)`, but Haskell 98 derives a more general type (see below). A type with one arrows is called a *function type* and a function whose type has two or more arrows is called *higher-order*. The above function type means the following: the first parameter is of type `Float`, and if such a `Float` is provided, the result is again a function. This result function takes another `Float` and returns a pair of `Float`s. This is different from a simple function that takes and returns pairs of `Float`s. Its type is `(Float,Float) -> (Float,Float)`.

Mentioning types in source files is not mandatory in Haskell. The type checker automatically infers the type of every expression in the program. However, if a function is augmented with a type as the programmer intends it to be, the type checker compares inferred and provided types and generates an error if the two do not match. This way programming errors can be located better. If the type is omitted and, due to a mistake, different from the intended one, but still a legal Haskell type, then possibly the application of the faulty function somewhere else in the program will result in a type error. However, the error will be displayed at the function application, and not the incorrect function definition. Therefore it is good practise to provide types for all functions.

A type expression may contain type variables that are placeholders for other types. The set of types that a particular variable may assume can be restricted by contexts. Functions with type variables in their types are called *polymorphic*. An example is the function `isZero`:

```
isZero a = if a==0 then True else False
```

Its type is

```
isZero :: (Num a) => a -> Bool
```

which means that the type of the parameter can be any type (type variable `a`), as long as the context `Num a` is satisfied. This context restricts the types that `a` can assume to numeric types, such as integers and floating point numbers. It is inferred at the comparison to the numeric constant `0`. With a similar argument, the actual type of the above function `solve_eq` is:

```
solve_eq :: (Floating a) => a -> a -> (a, a)
```

Lists

An important data structure in functional languages is the list, an ordered collection of data items of the same type. In Haskell, lists elements are separated by commas and enclosed by square

brackets. Many standard functions on lists are higher-order. An example is the function `map` that applies a function to each element of a list, yielding a new list:

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x):map f xs
```

The function `f` may change the list elements' type, but the type variables `a` and `b` may also stand for the same type. `map` is polymorphic: it works on lists of any type. The above definition is also an example of *pattern matching*. If a list is provided as a parameter, then empty and non-empty lists often need to be distinguished. This can be done by providing several cases of the function, each with a different pattern on the discriminating parameter. Here, the pattern `[]` in the first definition matches exactly the empty list, the pattern `x:xs` matches only and all non-empty lists. To build patterns, we use type constructors of the corresponding type. The list has two of them, which are `[]` (pronounced *nil*) for the empty list, and `:` (pronounced *cons*) for an operator that adds one new element in front of a given list. This is why `x:xs` matches only non-empty lists: whatever the list `xs` is (it may or may not be empty), the cons-part `x:` means that at least one element has been added to the list, so the matched list cannot be empty.

An example of the use of `map` is the following application to an integer-valued list, yielding a boolean list:

```
> map isZero [2,3,-4,0,2,0]
[False,False,False,True,False,True]
```

Lines starting with a `>` indicate user input in an interactive session, followed by the interpreter's result. A convenient way to construct lists is the *list comprehension*. It looks like the mathematical set notation, starting with a general expression to generate entries and, separated by a vertical bar, one or more generators and/or guards. In a generator `x <- xs`, the operator `<-` *draws* element by element from the list `xs`, naming them `x` each time, and generating one resulting element for each `x`. Boolean guards filter generated elements, and several generators construct cartesian products of their input lists. The following examples demonstrates the conciseness of this notation:

```
> [ x+5 | x <- [1,2,3,4] ]
[6,7,8,9]

> [ x+5 | x <- [1,2,3,4], even x ]
[7,9]

> [ (x,y) | x <- [1,2,3], y <- [9,8,7] ]
[(1,9),(1,8),(1,7),(2,9),(2,8),(2,7),(3,9),(3,8),(3,7)]
```

User-Defined Data Types

An important concept is the definition of new data types. For our work, two kinds are important: type synonyms and algebraic data types. Synonyms are defined with the keyword `type` and are used as a shorthand for more complex types. The definition

```
type Customer = (String,String,String,Int,Int)
```

allows us to use the type name `Customer` instead of the long tuple type. However, no new type is introduced; the two types can be used interchangeably.

If the discrimination of a new type is needed, one can use algebraic data types. The keyword `data` is followed by the new type name and optional type variables. The type is defined by one or

several constructors, that may each have several and possibly different parameters. As a simple example, we define an enumeration type of weekdays:

```
data Day =
   Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
```

There are seven different constructors, each without any parameter. Each constructor is an element of type `Day`, and there are no more, so that together they *construct* the type `Day`.

The constructors generate all elements of a type. Built-in types can have infinitely many constructors, such as the arbitrary-length integer type. Every integer number is a constructor of this type. As an example of constructors with parameters, we can implement a user-defined list type:

```
data Mylist a =  EmptyL
               | Cons a (Mylist a)
```

The type variable `a` is the polymorphic type of the list elements. A `Mylist` of integers has type `Mylist Int`. The difference to the built-in list is just of a syntactical nature. Haskell lists have a two symbol type name `[.]`, and the second constructor `:` is infix in contrast to the prefix `Cons`.

Type Classes

The functions `solve_eq` and `isZero` contain type class restrictions in their types. A type class consists of a set of types that are guaranteed to provide an implementation for certain functions. If a type is a member of that class, we know that we can use a class function on expressions of this type. The predefined Haskell class `Num` is defined in the standard library (*prelude*) like this:

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a
    fromInteger    :: Integer -> a
```

This definition states several things. Any type `a` that is an instance of `Num` must also be an instance of `Eq` and `Show`, which means that equality and conversion to strings must be defined on it. The class definition also provides types for seven functions that have to be defined for a type in `Num`. All predefined numeric Haskell types such as `Int`, `Double`, `Float` and `Complex` are instances of `Num`. If a new type is defined by a user, it can also be made an instance of `Num` and therefore use the overloaded operators `+`, `-`, `*` and so on. There is, however, no mechanism to ensure that the semantics of the new `+` function has anything to do with addition. Instance declarations are made using the keyword `instance`. For some user-defined numeric type `MyNum`, it starts like this:

```
instance Num MyNum where
     a + b = ...
     ...
```

The instance definition provides implementations of the class functions. This is done by overloading the operators and functions. By inspecting the types of the function parameters, the run time system decides which of the overloaded functions it needs to call.

Arrays and `LArrays`

Although lists play a central role in functional languages, scientific algorithms usually use indexed data for which the array is the most suitable representation. Haskell provides arrays of arbitrary

element types. The type of the indices can be any ordered type. This general array implementation is much more flexible than, say, an array in C. Arrays in Haskell are initialised and updated with *association lists*, i.e., lists of index-value pairs. The index domain is always rectangular and its extent is defined by the upper left and the lower right element. The following example creates an integer array with both indices ranging from 0 to 2, but only two elements are defined.

```
a = array ((0,0),(2,2)) [((0,0),3), ((0,1),5)]
```

The exclamation mark is the read operator. A read access of a yet undefined element (e.g., `a!(0,2)`) results in a run time error. A convenient way to define regular arrays is by using list comprehensions:

```
b = array ((0,0),(2,2)) [((x,y),x+y) | x <- [0..2], y <- [0..2]]
```

As arrays are lazy data structures in Haskell, self references in the initial definition are allowed as long as the data dependences between the array elements can be solved (cyclic references lead to a run time error):

```
c = array ((0,5)) ((0,15):[(x, 2*c!((x+1) `mod` 6)) | x <- [1..5]])
```

The value of `c` is:

```
> c
array (0,5) [(0,15),(1,480),(2,240),(3,120),(4,60),(5,30)]
```

Updates of arrays are done by using the `//` operator on an array and an association list. The expression `c // [(0,100)]` yields:

```
> c // [(0,100)]
array (0,5) [(0,100),(1,480),(2,240),(3,120),(4,60),(5,30)]
```

The standard Haskell arrays are multi-threaded, so that after an update the old version of the array can still be used. Consider the following example:

```
let a = array ((0,0),(2,2)) [((0,0),3), ((0,1),5)]
    b = a // [((0,0),5)]
in a!(0,0) + b!(0,0)
```

Two arrays exist here. The updated array `a` gets a new name, while the old, not-updated `a` still exists. A consequence is that every array update requires the creation of a copy that differs from the old array in just this one element. Even if the use is single-threaded (which is the most common case), the Haskell system still creates a copy. This makes Haskell programs with lots of array updates memory and time consuming. In this thesis, we will use Haskell to implement abstract machines and simulate abstract programs on them. The inefficiency of standard array operations reduces the manageable problem size significantly, but there are ways to overcome this.

One solution is to use arrays within a state transformer monad. An expression of a monadic type contains a sequence of sub-expressions that have a guaranteed linear evaluation order [Tho99]. Some applications, e.g., input/output, require such an guaranteed order, so that their corresponding expressions are put in a monad. Array accesses inside a monad are necessarily single-threaded, although each access itself may still be lazy. A compiler optimisation may recognise the single-threadedness and perform the array updates efficiently *in-place*, i.e., by overwriting the old value instead of the standard Haskell copy-update. However, we feel that the monadic programming style would make the APM interpreters and programs difficult to read, as they do not look like Haskell programs anymore.

As an alternative, we use the module `LArray` [Gro03] that implements linear arrays with single-threaded semantics and in-place updates while retaining the standard Haskell syntax. The only drawback of this module is that the compiler cannot check the single-threaded use – if a program using `LArrays` is in fact multi-threaded, the results are wrong. But in our realm of scientific algorithms such a case never happens. Throughout the APM interpreters and programs in this work, we use the `LArray` module.

2.2 Parallel Programming

This section introduces some basic concepts of parallel programming. We first describe the application domains in which parallel computing is used. Then we present standard techniques for loop parallelisation, which play a crucial role in the area of automatic parallelisation, before proceeding to a more advanced technique that we will be using in this thesis. Finally we introduce some concepts for measuring the performance of parallel programs.

2.2.1 Problem Domains for Parallel Programming

The classic domain for parallel programming is scientific computing. This is where very complex computations on large data sets are needed. Be it physics, chemistry or engineering, they all need mathematical algorithms. Most of these algorithms work on matrices that are implemented as arrays. New matrices are computed and matrix elements are updated. In both cases, the algorithm has to iterate over the elements, which is done by a loop nest in an imperative program. Such programs spend the majority of their run time within these loop nests. Thus, loop parallelisation is an important area of research.

Recently, other domains show an interest in parallel computing. Finance companies and other database users have large processing needs. However, database operations are based on a different class of algorithms that we do not deal with in our work.

The advent of parallel clusters made of commodity hardware enables users with less money to use parallel processing power. However, cluster applications usually use coarse-grained parallelism on the process or thread level. These applications often employ the multi-threaded programming paradigm for concurrency reasons so that the performance increase on a cluster is not much more than a welcome side-effect. Therefore, we will not concentrate on this area but on the parallel program generation for scientific algorithms.

2.2.2 General Loop Parallelisation Techniques

Most parallel programs today are still written in Fortran or C, so these languages are in the centre of research in parallel programming.

Compiler optimisations have used loop restructuring transformations for a long time. It turns out that some of these transformations are also useful for turning a sequential loop (nest) into a parallel one. In the context of a loop nest, parallelism means that the iterations of one or several loop levels are independent and can be computed on different processors. The problem is either to identify such loops or, if none are found, to try to transform the loop nest into an equivalent one that does have independent loops. Equivalence is meant semantically – all transformations must preserve the input/output behaviour of the code. We present briefly the basics of loop restructurings.

A central concept is the *data dependence*. Two statements depend on each other if their execution in a reverse order leads to a different effect. However, the concept of a dependence between statement is too coarse-grained. A loop nest with only one statements in the body may have dependences between some of its iterations. Therefore we define dependences on *operations*,

which are statements outside of loops or statement iterations within a loop. Three kinds of data dependences are distinguished. Consider operation O_1, followed by O_2. If they access the same memory location and at least one of them performs a write, there is a dependence. The operations are said to be

flow dependent if O_2 reads what O_1 wrote,

anti dependent if O_2 overwrites what O_1 has previously read, or

output dependent if O_1 and O_2 both write.

In these cases, O_1 and O_2 cannot be interchanged without possibly changing the semantics of the program. There is a simple method to eliminate many dependences. A variable or array is said to be *single assignment* if it is only written once. Every program can be transformed to a single-assignment variant by adding an extra dimension to each variable to hold the different values. If a program consists only of single assignment data structures, only the flow dependences remain as no overwrites occur. Haskell programs are always single-assignment.

Dependences have to be direct. If there is a flow dependence between O_1 and O_2, and another one between O_2 and O_3, we do not consider the dependence between O_1 and O_3. However, all dependences of a program's execution comprise a partial order. If there is more than one strongly connected component within this partial order, then the loop nest can be restructured so that there is at least one parallel loop.

To get an idea of how parallelism can be exposed in loops, we present some of the more frequently used techniques.

Loop Fusion combines two consecutive loop nests. It reduces the loop overhead and may increase the amount of parallelism if at least one resulting loop is parallel.

Loop Coalescing combines two or more loop levels into one, thus reducing the number of loops in a nest. If the newly created loop is parallel, the parallelism is increased.

Loop Interchange changes the order of loops within a loop nest. Some loops may not affect the set of dependences so that they can be freely moved around within the loop nest. It is often favourable to have such loops on the outside and declare them as parallel.

Loop Skewing is a transformation of the loop bounds to establish parallelism when none of the existing loops is parallel. Two or more loops span a multi-dimensional index space. Usually, each loop enumerates one dimension. Instead, a skewed loop facilitates a wavefront of computations through the *index space*. The computations within the wavefront are independent so that they can be enumerated by a parallel loop. This is a special case of the transformation methods based on polytope model that we present in the next subsection.

Other loop transformations include loop shrinking, loop unrolling and loop distribution. A more detailed account of this can be found elsewhere [Wol95].

2.2.3 Loop Parallelisation in the Polytope Model

Each loop iteration can be identified by the values of all surrounding loop variables. If there are n such loops, an n-tuple comprising these values represents a point in $\mathbb{Z}^n$. The entire loop iteration is a subset of $\mathbb{Z}^n$. If the bounds of all loops contain only linear expressions of constants and surrounding loop variables, the iteration set of the loop, the *index space*, is an intersection of $\mathbb{Z}^n$ with a polytope. The left side of Figure 2.1 shows a loop nest with two loops and, below it, the derived iteration space for $n = 3$. Each point denotes one iteration. Each axis relates to a loop level. The arrows between the points denote a data dependence between statements of the

```
for i := 0 to n do
    for j := 0 to i + 2 do
        A(i, j) := A(i − 1, j)
                   +A(i, j − 1)
    end
end
```

```
for t := 0 to 2n + 2 do
    forall p := max(0, t−n) to min(t, ⌊t/2⌋ + 1) do
        A(t−p, p) := A(t−p−1, p)
                     +A(t−p, p−1)
    end
end
```

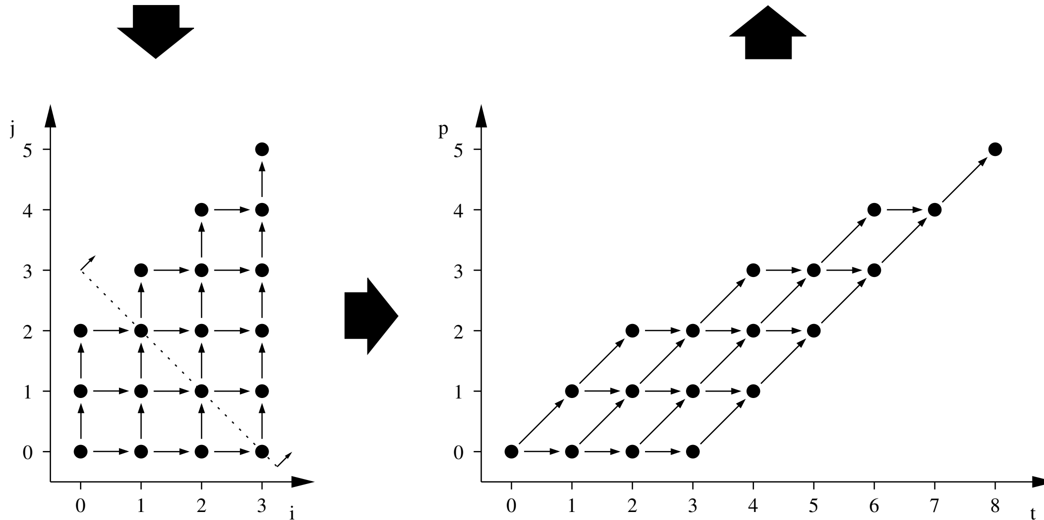


Figure 2.1: Parallelisation in the Polytope Model

corresponding iterations. Note that there are dependences parallel to each axis so that no loop can be made parallel. However, if we think of a wavefront starting in the lower left at $(0, 0)$, going to the upper right as shown by the dotted line, then all iterations on the front are independent.

In order to parallelise the loop nest, the task is to find a coordinate transformation for the index space, resulting in the *target space*, such that the iterations along at least one axis have no dependences between them. Such a loop can be made parallel. This coordinate transformation is called a *space-time mapping* and is applied to all expressions in the loop bounds and in the body that contain the original loop variables. The lower right of Figure 2.1 shows the index space after the coordinate transformation. The transformation yields a time dimensions t and a space dimension p instead of i and j. The time dimension enumerates the advancing front, the space dimension all iterations on the front.

Note that for each time value, the iterations that follow the p dimension vertically have no dependence between them. Thus, a loop enumerating p could be done in parallel. From the target space we can derive the target program on the upper right. The sequential t loop needs only 8 iterations for $n = 3$, while the source program needs 18 sequential iterations. A detailed description of this parallelisation scheme can be found elsewhere [Len93].

The characteristics of a program fragment suitable for treatment with a paralleliser based on the polytope model are:

- The program is a loop nest whose body consists of array assignments.
- Loop bounds and array index expressions are linear and may contain other variables which are constant within the loop nest.
- Other than assignments, only conditionals and procedure calls are analysed in the loop body.

In the parallelisation, the latter are treated as read and write accesses to all parameters.

The determination of the parallelism inherent in the loop nest and the generation of semantically equivalent, parallel code proceed as follows:

- Data dependences are calculated. They associate accesses of different operations on the same array element. If such dependences exist, they impose a partial ordering on the operations.
- The task of "scheduling" is to find a function mapping all operations to points in time. This is also called the *time mapping*. The optimisation aspect is to map every operation to the earliest possible time without violating the data dependences. If there is any parallelism in the program, there will be times with more than one operation scheduled to.
- The dual mapping, the "allocation", assigns the operations to specific processing elements (PE). This is called the *space mapping*. One goal is to optimise data locality on the PEs, thus minimising communication.
- A code generation procedure performs a program transformation of the original source code into parallel target code by using the *space-time mapping*.

Intuitively speaking, the index space of the original loop nest is skewed by means of a coordinate transformation in such a way that there exist some dimensions along which all iterations are independent. These dimensions may be enumerated by a parallel loop, all others are enumerated by a sequential loop. Of course, a skewing must not produce dependences going backwards in time.

2.2.4 Concepts for Performance Measurement of Parallel Programs

Parallel programming is about speed. The question is always how much faster a computation gets if we use a parallel computer. Run time measurements (*benchmarks*) yield performance figures to judge the effect of the parallelisation. For the analysis of the benchmark results, we need some definitions as can be found in any standard text book on parallel programming (e.g., see [Fos95]).

In this section, we write T_s as the run time of the best sequential algorithm on one processor, T_1 for the run time of a parallel algorithm on one processor, and T_p for the run time of a parallel algorithm on p processors.

A *speedup* is a factor by which one program's execution time is faster than another's. If the program is indeed slower, i.e., the speedup is less than 1, people sometimes call it a *negative speedup*. If the speedup on p processors is equal to p, it is called a *linear speedup*. Most often, this is the theoretical limit, as one wouldn't expect that p processors can be more than p times faster than one processor. However, *super-linear* speedups exist in cases where cache and buffer effects play a major role.

Relative Speedup The *relative speedup* on p processors is defined as follows:

$$S_{rel} = \frac{T_1}{T_p}$$

It compares the parallel algorithm on 1 and p processors and, thus, measures the scalability of the parallel algorithm. It is not, however, an absolute measure of the quality of the algorithm. If T_1 and T_p are very big, the relative speedup may still be good as long as the algorithm scales well.

Absolute Speedup This is the real world measure. It compares the parallel algorithm on p processors with the sequential algorithm and is defined as:

$$S_{abs} = \frac{T_s}{T_p}$$

It answers the question: if a sequential algorithm is too slow and a corresponding parallel algorithm is run on p processors, how much faster would it get? As $T_s \leq T_1$ holds, we have: $S_{abs} \leq S_{rel}$.

Efficiency Up to a certain point, the more processors are used, the greater the speedup. However, the speedup increase may slow down, i.e., additional processors have less impact on the speedup. Intuitively, the efficiency goes down. Therefore, we define the *efficiency* as the relation of a speedup to the number of processors:

$$E_{abs} = \frac{T_1}{p * T_p} = \frac{S_{abs}}{p}$$

The efficiency is often presented as a percent value.

Chapter 3

The **PolyAPM** Framework for the Development of Parallel Programs

In this chapter we describe our approach to the structured development of parallel programs. The idea of stepwise refinement of specifications has existed for a long time in computer science. Each step in a refinement process results in a new program version, thus the whole process leads to a sequence of programs. Graphs (usually trees) are used to present the combination of alternative sequences of design decisions. If we look at the various intermediate specifications that exist between all transformations of the parallelisation process, we observe an increasing degree of concreteness while we proceed. Thus, the abstract specification is eventually transformed into a binary for a target machine. Consider the intermediate steps: on our descent along one path down the tree, we pick up more and more properties of the target architecture. But this also means that, most likely, no existing machine matches the level of abstraction of any intermediate specification. If we employ an abstract machine model that is just concrete enough to cover all the details of our intermediate program, we can have it implemented in software and even run our programs on it.

3.1 Abstract Machines and Stepwise Refinement

Abstract machines are being used when a programming model reflects only a subset of a given set of machine characteristics. This reduces the complexity of the machine and thereby also the complexity of the programming process. This way, programs for abstract machines are simpler to write and to reason about, and furthermore the simplified operational behaviour enables simple yet effective cost models.

In our problem domain, the generation of parallel programs, one frequently employs *program transformations* to introduce new program characteristics or optimisations. These transformations use source code in a suitable representation as input and output. With the use of abstract machines and their structurally simple programs, these transformations become easier as they just have to deal with relevant machine characteristics. Lower level details – like code that is required to make the program executable on a real machine – are factored out as long as possible. Eventually a program will be run on a real machine, so that these details have to be filled in. But until that is necessary, the program generation process benefits from the higher abstraction level during most of its phases.

The above observations motivate the use of abstract machines for any program transformation approach. In our work, we employ the *stepwise refinement* method of program generation. Refining a program can be viewed as making it more concrete, less abstract. Therefore, as stepwise refinement is a program transformation approach, it matches ideally with the abstract machine

model. The refined programs are designed for a particular abstract machine and each machine adds only so much more details as the refinement step to it requires.

3.2 Abstract Parallel Machines

Compared to sequential programming, writing a parallel program requires additional work to be done. Not only must one deal with implementing the algorithm in a programming language, additionally things like identifying and placing independent operations, distributing data and improving performance become important. These new concerns are on an operational level that is often more low-level than the actual program. In addition, there have been many approaches on how to separate them from the program in order to improve the readability and ease of programming.

Many abstract machine models for parallel programming have been proposed [DD95]. Our approach is to use the observations from the previous section and use stepwise refinement as a programming model on *Abstract Parallel Machines* (short: APMs) following the ideas of O'Donnell/Rünger [OR97]. They describe APMs as *single program multiple data*-style (SPMD) distributed memory machines. Although the definition itself is purely operational, they have also presented an I/O-specification in a functional language. The programs for these machines consist of functional compositions of *parallel operations* (ParOps). The APM interpreters are implemented and the APM programs are embedded in a functional language, so that APM programs can be interpreted within one language realm.

Several key properties of the APM approach support its feasibility:

- Due to the simplicity with which new data types can be defined and passed around using *higher-order functions*, functional languages are well suited for writing interpreters.
- Together with input, the interpreter for an APM and the APM program are executable, so that the validity of the APM program can be tested.
- APM programs as data types can be handled easily, thus supporting program transformation techniques.
- Semantic properties of pure functional languages like Haskell [PJHe99] enable reasoning techniques that make verification proofs of program transformations comparatively simple (e.g. *equational reasoning*).

The main difference between APMs and most other parallel abstract machines is the ease of APMs to be implemented in software in order to execute their programs, whereas other machines usually serve a theoretical purpose (e.g., cost models).

3.3 PolyAPM

The purpose of PolyAPM is to provide a framework for a stepwise refinement approach to parallel programming. Focus is laid upon dividing the transformation process into small chunks, evaluation and executability of intermediate programs, mixed manual and automatic transformations as well as general support to explore a particular problem domain.

To meet these goals we regard intermediate programs to be executable on an abstract machine. These machines have to match the program's level of abstraction. Thus we need abstract parallel machines whose capabilities may be changed. The APMs as presented by O'Donnell and Rünger may be adapted at two different places: the computation function that effectively represents the APM program and the communication function representing the network wiring. Other than

that, no hardware detail varies at different levels of abstraction. In order to match the needs of stepwise refinement in a program transformation approach, we would like the programs in the transformation process to run on machines that exhibit just enough hardware detail to cover the program's needs, but not more. Therefore we have designed a sequence of APMs that accompany the sequence of transformed programs. In general, there are fewer APMs than programs, since not every transformation introduces new machine requirements.

3.3.1 Machine Model

The PolyAPM abstract machines have a one-dimensional processor field of arbitrary, but finite size n. In a distributed memory setting, each processor has unbounded local memory and is connected to a buffering network with unspecified topology. In a shared memory setting, there is no local memory, but possibly a communication network. As to why, see Section 3.3.4. All processors run the same program in SPMD style. The structure of the programs is a loop nest of at least two outermost, perfectly nested loops. Possible inner loops may not be perfectly nested. Of the two outer loops, exactly one must be tagged sequential and the other parallel. If the outer loop is parallel, we call the program *asynchronous*, otherwise *synchronous*.

In the terminology of the polytope model, these machine properties restrict us to a one-dimensional allocation in both cases. In the synchronous case we need a one-dimensional schedule unless the inner sequential loops do not carry any dependences and may be moved inside the parallel loop. It is important to note that the machine model defines a communication phase at the end of each *time step*, i.e., after each iteration of the outermost sequential loop. There is no communication possible inside the inner loops. Concerning dimensionality, for all problems a one-dimensional allocation can be found. One-dimensional schedules are only guaranteed for programs with linear dependences. But loops derived from multi-dimensional schedules can always be combined into one loop if one accepts computationally complex loop bounds and strides. Therefore, we usually restrict ourselves to one dimension on both cases without losing too much generality.

The parallel loop that enumerates processors is eventually replaced in each program instance by a variable assignment of the corresponding processor number. The sequential time loop provides a global clock enumerating time steps. Within such a time step, first a computation takes place, i.e., everything within the time loop is executed, and second inter-processor communications are performed. Such a time step is similar to BSP's *superstep* [SHM97]. In fact, in the synchronous execution mode we require a barrier synchronisation after the communication phase. The operations of the same logical time step in the asynchronous execution mode may not happen at the same wall clock time, yet they correspond. A limited synchronisation is done by the message exchange, so that no global barrier is necessary.

Note that the communication phase is optional, it only exists if a particular APM is designed for it (e.g., in a distributed memory environment).

3.3.2 Structuring APMs into Graphs

One crucial aspect of parallel programming is the need for design decisions. A transformation process may not be single-threaded, meaning that at some point several alternative transformations are possible. This leads directly to a tree of transformed programs, but in general we observe a directed acyclic graph structure. This is because two or more subsequent transformations may be completely orthogonal and therefore commutative, so that after all have been performed, the resulting programs are identical in all applied orders. Thus, the process of deriving a target program is like traversing the graph of design decisions, the PolyAPM Decision Graph (PDG). Each node in this graph is a transformed program that runs on a dedicated APM. An example of such a graph is depicted in Figure 3.1. It will be discussed in detail in subsequent sections.

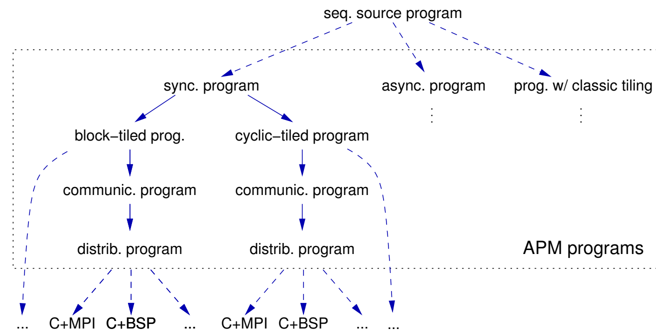


Figure 3.1: PolyAPM Decision Graph (PDG), here only a sub-tree

These accompanying APMs create a graph themselves, the PolyAPM Machine Graph (PMG), in which each node corresponds to an APM and the arcs denote an "is being refined to" relation. As noted above, the PMG has usually fewer nodes than the corresponding PDG, so that there exists a many-to-one mapping of programs to APMs.

To use the PolyAPM framework for program construction it is necessary to identify the possible transformations within the compilation process. These transformations are combined into a PDG. They have then to be examined as to whether they introduce additional machine requirements, so that for each transformation an accompanying abstract machine is defined. As not every transformation introduces new requirements, there is not always the need to define a new machine. The structure of the transformations leads to a hierarchy of APMs that can be implemented in software. It presents the basis for the subsequent programming process of APM programs.

The above framework comprises a software engineering approach to split the process of writing/compiling a program into a sequence of transformations, where each transformed program is designed for an abstract machine. Thus, nothing so far is specific for writing parallel programs as the abstract machine could just as well be sequential. This approach may even have merits for such a general application domain, but we specialise on the development of parallel programs where we benefit greatly from some properties of this approach.

3.3.3 Development Process

In *automatic parallelisation* the view is to tackle the difficulties of writing parallel programs by letting the programmer write a conventional sequential program and leaving the parallelisation effort to a compiler. This approach requires sophisticated program analysis techniques. The PolyAPM approach provides an alternative framework when not all transformations can be automated.

There are some problems to be aware of while considering PolyAPM:

1. the experience of a seasoned programmer may not be turned easily into good program transformations,
2. there may be no systematic way to determine the most suitable transformation among a set of alternatives,
3. some desirable transformations may easily become computationally intractable (exponential or worse) and
4. the formulation of a problem in a sequential imperative way may introduce algorithmically unnecessary data dependences that are difficult to remove and that reduce the degree of

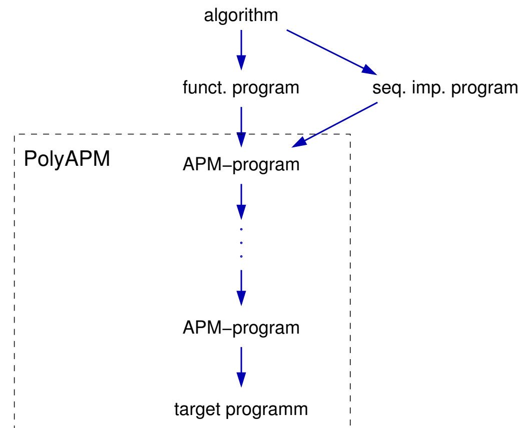


Figure 3.2: PolyAPM Structure

determinable parallelism. See Chapter 6 for an example of this.

While the first and the third problem lie outside the scope of PolyAPM, we strive to provide help for the second and the fourth one. PolyAPM enables a semi-automatic parallel programming approach in which user input will be used where experience cannot be replaced by an algorithm. Since we break down the entire compilation into several phases, a mix of automated and manual transformations and decisions is possible. This may lead to the iterative development of a complete compiler if all transformations are eventually automated. In this case, PolyAPM would not be so much of an alternative, but a construction method for an automatic parallelising compiler. However, because of the four problems which we have just listed, one may not want to automate some transformations. The main goal of PolyAPM is to provide a manual or semi-automatic program generation environment.

As to the fourth problem, the difficulties possibly introduced by an imperative language are avoided by using a *declarative language* for specifying the algorithm. In declarative programs, there is not necessarily a requirement of a sequential order of the operations and no mention of parallelism. The declarations merely state *what* is to be computed, rather than *how*. We argue that this is the easiest of all choices for the programmer, especially as there is a striking similarity between the way mathematical formulae (an important domain in parallel programming) are stated and the way they are being implemented in a functional language. See Section 2.1.2 for an example of such a similarity. We employ the functional language Haskell for this purpose. However, this choice is independent of the languages selected for the abstract machine programs. In fact, we show that the APM programs, although embedded in Haskell, have an imperative control structure. This is no contradiction to the above, as we distinguish between the declarative specification program (before PolyAPM) and the APM programs (part of PolyAPM).

The overall picture describing how parallel programs are developed in PolyAPM is depicted in Figure 3.2. The task is to implement a mathematical algorithm and have it running on a parallel computer. First, the problem is implemented without thoughts of parallelism to relieve the programmer of this additional burden. Then a transformation process is started to parallelise this program. Thus, the declarative program has to be parallelised to match the first and most abstract parallel machine. From there the PolyAPM framework is used to select and perform optimising code transformation until a sufficient level of concreteness is reached. The last APM should be quite similar to the real machine to be used, so that the last transformation from a PolyAPM program to a program for a real machine is as easy as possible. This is important as the last transformation is not formalised.

In the following, we describe this process in more detail. First, recall the PDG of Figure 3.1. The transformations depicted in it have been motivated by our experiences with the *polytope model* for parallelisation [Len93] within the LooPo project [GL96], but PolyAPM is not restricted to this model.

The program development process is divided into several phases as follows:

1. Implementation of a problem specification in standard (sequential) Haskell as a *source program*, which serves as the specification. Note that we avoid talking about sequential Haskell programs, as they have a declarative semantics and do not impose any sequential evaluation order. We have stated before that the two main reasons for using Haskell are its suitability for implementing mathematical algorithms and the smooth integration of APM programs and APM interpreters within one language. However, these reasons make the choice of Haskell only highly suggestive, but not a requirement. We point out above that it is also possible to write the source program in an imperative language.

2. Initial parallelisation of the sequential program. This requires the analysis of the problem to identify independent computations that can be computed in parallel – a process which might be done manually, or with the help of a parallelisation tool (as is the case in our example in Section 4). We have used LooPo for this purpose. In any case, the result of the parallelisation should map each computation to a virtual processor and to a logical point in time (see Section 2.2.3). The granularity of the computation is the choice of the programmer, as the PolyAPM framework will maintain this granularity throughout the process. As the source program will most likely contain a repetitive construct, e.g., recursion or a comprehension, it is often sensible to perform the parallelisation on these and keep the inner computations of the recursion/comprehension *atomic*.

   Without loss of generality, we assume a one-dimensional processor field so that we have the basic computations, their allocation in *space* (i.e., the processor) and their scheduled computation time. With these components, the problem has a natural expression as a loop program with two loops: one processor loop which is parallel, one time loop which is sequential, and the loop body which is our atomic computation. Thus, we have either a synchronous or an asynchronous program.

   This motivates the corresponding branches of the PDG in Figure 3.1. The right branch for classical tiling is a special case in which parallelisation is not the first step.

3. Based on the parallelisation, the source program is transformed into an APM program, which resembles an imperative loop nest with at least two loops levels (there may be additional loops in the source program's core computation). The program is subject to several transformations to adapt it to other APMs. This is a central aspect of PolyAPM and is discussed in more detail in Section 3.3.4.

4. The final result of the compilation, the *target program*, has to be executable on a parallel machine. Therefore, the last APM program is transformed into a *target language* for the parallel machine. It is important that the target language exhibits at least as much control as the last APM, so that no optimisation of any APM program transformation is lost. Suitable target languages, among others, are C+MPI and C+BSP.

3.3.4 Abstract Machines and their Programs

The APMs form a tree, as shown in Figure 3.3. There is a many-to-one mapping from the programs to APMs. An APM program must reflect the design characteristics of the corresponding APM, e.g., in case of a synchronous program, a loop nest with an outer sequential and an inner parallel loop and a loop body, which may contain more loops. This separates the loops which represent the parallel execution from the inner sequential loops to be executed on the processors. Here,

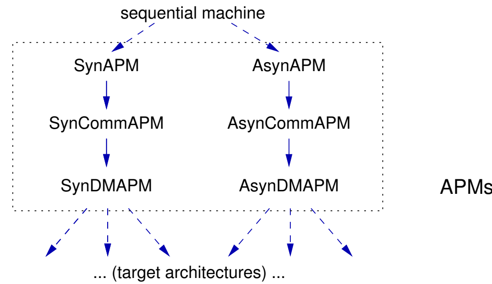


Figure 3.3: PolyAPM Machine Tree

we deal only with a one-dimensional allocation. The model could be extended to incorporate multi-dimensional allocations.

The synchronous program is subject to a sequence of source-to-source program transformations. Each adds another machine characteristic or optimises a feature needed for execution on a real parallel machine. Assuming that the original parallelisation was done for a number p of processors whose value depends on the input, the p processors' workload has to be distributed on rp real processors of the target machine. This transformation is called *processor tiling*, in contrast to tiling techniques with other purposes. In this work, the term *workload* denotes the amount of computational work that is scheduled on a processor. Depending on the context, workload is measured in number of operations or in computing time.

The next two transformations complete the transition to a distributed memory architecture with communications. This has been deliberately divided into two transformations. First, while still maintaining a shared memory, we generate communication directives. As a second step, the memory is distributed, introducing the necessity of communication. The reason for this unusual separation is twofold. One of the aims of PolyAPM is to make each transformation as simple as possible, and both communication generation and memory distribution can get complicated. Furthermore, if we applied both transformations in one step and the resulting APM program had an error, it would be more difficult than necessary to isolate the reason for this error. When interpreting an APM program that communicates even in the presence of shared memory, the communications perform identity operations on the shared memory cells. The APM interpreter checks for this identity and issues a warning in case of a mismatch. This way, wrong communications are detected already after the first step, while the effect of missing communications shows up only after the distribution of memory.

The transformed program runs on an APM capable of communications, the SynCommAPM, which provides a message queue and a message delivery system. We assume that each processor stores data in local memory by the *owner computes rule*. This placement strategy distributes the global data completely into the local memories of all processors, and each data item is stored only on the processor where it was computed (i.e., the *owner*). Therefore, data items computed elsewhere have to be communicated, either by point-to-point communication or by collective operations. If we were to employ a different storage management rule, this transformation would have to be adapted accordingly.

The "unnecessary" communications of the SynCommAPM program become crucial when the memory is being distributed for the SynDMAPM program. This branch of the tree uses the owner computes rule, making it easy to determine which parts of the global data space are actually necessary to keep in local memory. That completes the minimal set of transformations needed for

a synchronous loop program on a distributed-memory machine. The last transformation generates so-called *target code*, i.e., it transforms the SynDMAPM program into non-APM source code that is compilable on the target machine. Possible alternatives include C+MPI and C+BSP.

As outlined in Figure 3.1, transformation sequences other than the one for synchronous parallelism are possible. In addition to the corresponding sequence for asynchronous parallelism, Figure 3.1 contains a branch for a typical sequence as employed by the tiling community [Wol89].

3.4 A Cost Model for PolyAPM

In the previous sections we have presented how the compilation process is broken up into a sequence of transformations that are viewed as source-to-source transformations of abstract machine programs. If there exist alternatives for a particular transformation, a choice has to be made. Help for such a decision may be given by a cost model, which is usually a function of the number of processors and the input size, and whose values correspond to the expected execution times of the program.

For PolyAPM we have devised a cost model that is applicable to any abstract program, so that the anticipated performance can be used to judge the last transformation. As the programs become more concrete during the transformation process, their cost should correspond more closely to the target program's execution time. Since we do not impose severe regularity restrictions on PolyAPM programs (such as linear dependences), the communication structure may not be regular or may even be dynamically changing. This presents a problem with some cost models that require communication properties to be inferred from the program code. Therefore, we utilise the APM interpreters which can run the abstract program and simultaneously collect profiling data of the run. This data is used to derive a cost value.

There is one inherent problem with such a cost model for an abstract architecture: it requires some values that can only have enough credibility if measured on a real machine, or at least if motivated by experiences made with real machines. Yet, the cost model is for an abstract machine with the real machine possibly not yet decided upon.

It can be part of the PolyAPM programming process to find the best suitable architecture for a given problem. So which machine parameters does one use in the cost formula? The answer is to vary machine parameters in the cost formula so that transformation impacts on different machines can be observed alternatively. This introduces even more choices into the PDG. Of course, if the target architecture is already fixed, then the machine parameters remain constant and the model benefits from better accuracy of the prediction.

If different machine parameters favour different transformations and one of them is selected, then further down the derivation tree the selected machine has to remain fixed.

The general idea of obtaining the cost model formula is quite similar to BSP. Like supersteps, we have a sequence of time steps, each consisting of a computation phase and a possible communication phase. The existence of the latter depends on whether the particular APM is performing communications. This time step model applies to the synchronous and the asynchronous execution.

Also like BSP, in each step we compute the computation and communication times for each processor and determine the maximum for both over all processors. The reason is that in the synchronous model, the communication synchronises all processors. In the case of PolyAPM, all processors synchronise even before and after the communication. The time when all processors enter the communication phase is determined by the processor with the longest computation, and likewise the longest communicating processor determines the length of the global communication phase. As the processor with the longest computation may be different from the one with the longest communication, we determine the maximum values independently and sum them up to get the time (i.e., cost) for a particular time step. The different time steps are independent so

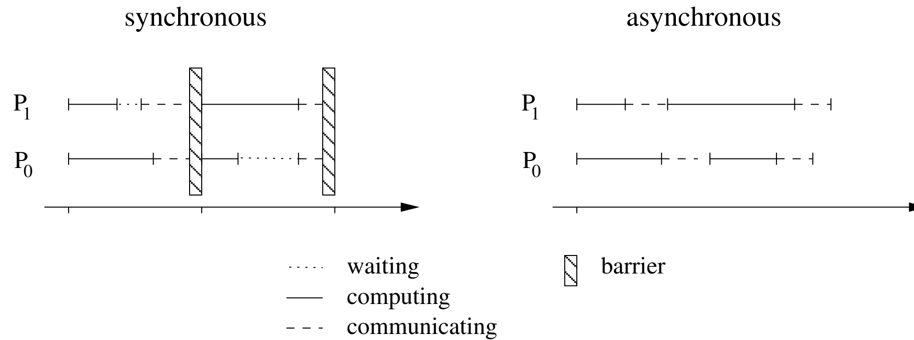


Figure 3.4: Synchronous and Asynchronous Communications

that their execution times are added.

This process is adequate for the synchronous model, but it only provides an upper bound for asynchronous programs, and even then only under the proviso that messages are coalesced. In the asynchronous case there is no waiting for the slowest. Processors may be synchronised pair-wise by message exchange, but there is no global synchronisation. The slowest processor of a time step doesn't necessarily have to synchronise at all. Thus, the other processors may already proceed with the next logical time step. If the slowest processor is not always the same, compensation can occur and the total execution time may be smaller than the sum of all maximum values of all slowest processors at each time step. See Figure 3.4 for an illustration of this effect.

However, an implementation may choose not to coalesce messages, i.e., at the end of a time step one processor may send several physical messages to the same communication partner. With coalescing they would be integrated into a single physical message. The problem is that without coalescing the message startup cost occurs more than once per communication partner. In general, asynchronous execution may lead to less waiting and thus a shorter execution time. But in a case where it may therefore be favourable and message coalescing is used, without coalescing a total execution time higher than in the synchronous model is possible.

These problems somewhat reduce the suitability of our cost model for asynchronous programs. This is unfortunate, but we have decided to keep it this way for the following reasons:

- This cost model is only a decision aid for the development process, it is no prerequisite. Therefore the restricted usability of the cost model in the asynchronous case does not preclude the usefulness of the entire approach.
- For asynchronous programs with message coalescing we do have an upper bound for the costs. This is just a loss of accuracy compared to the synchronous case.
- Cost models for asynchronous programs are rare and complicated [MRRV99]. There doesn't appear to be a simple extension of our model to cope with asynchrony.

The necessary data to compute the cost function is gathered during runs of the PolyAPM programs on their abstract machine interpreters, thus making this an empirical rather than analytical approach. The cost values (computation and communication costs) have to be gathered separately for each time step and the cost for all steps are added up to get the cost for the program.

In contrast, the BSP model deals with communication cost quite differently. In order to prevent hot spots in unbalanced distributions, it uses a randomised distribution of computations to achieve a randomised and, thus, evenly balanced communication pattern. This approach destroys any locality. In addition, BSP treats communication *en masse*, but its creators argue that their model still provides a good cost estimate. However, they admit that this approach does not work

well for regular programs that benefit from locality, but they claim that randomisation is worth it in most cases ([SHM97], page 5). We are not in the position to reject this claim, but we argue that our problem domain, the scientific algorithms, has a comparatively regular communication structure and therefore benefits from careful placement. Thus, we decided to be more specific than BSP in that we do a complete bookkeeping of all messages and their sizes in order to get a more detailed cost estimate.

The cost values for each computation and each message being sent are provided by the APM programmer within the APM program and indicate an approximate execution time to perform the operation. The values have to be determined by manually analysing the computations and setting them in relation to the unit cost (one floating point multiplication). The communication cost is relative to the sending of one floating point value.

The unit of the cost model value is called a *tick*. All times within the cost model formula are expressed in ticks. While the computation cost by definition is expressed in ticks, the communication costs have to be normalised to be comparable. This normalisation factor has to be determined by experiments.

In the APM programs, the programmer has to provide an estimate of how many ticks the computations within the loop body take. This is done by putting those values into a data structure from which the cost model calculation of the APM interpreters can read. For the communication cost, the length of the message is included in the APM message structure. The interpreters record this cost during message sending and delivery.

The calculation of the cost of a program run is defined by the PolyAPM cost formula in Figure 3.5. It defines for each time step and for each processor independently the communication cost as the sum of sending cost, receiving cost and a possible barrier cost. These costs have to be normalised suitably so that they can be added to the computation cost. For the sending and receiving cost we have the normalisation factors $\overline{s}$ and $\overline{r}$, that transform message lengths (denoting the cost of a message) into the number of ticks that are required to send or receive messages of that length. The message startup cost m is assumed to be normalised before being multiplied with the number messages. And finally the communication phase may be finished by a barrier synchronisation whose normalised cost is denoted by b.

In case that APM communications are to be neglected (e.g., on shared memory systems or with very abstract programs), only the computation cost remains and c_{it} is set to 0. For asynchronous programs we assume any synchronisation to be done by message passing so that the barrier cost b is set to 0.

3.4.1 Incorporation of the Cost Model into PolyAPM Interpreters

The cost model presented above is almost too complex for a manual calculation of a cost index, especially as the input's size also has to be taken into account. But in the PolyAPM framework we are able to execute the intermediate programs on abstract machine interpreters. And these interpreters are being used to collect the data necessary to calculate the cost formula's result. However, the relative costs of a computation and a message transfer are difficult to gauge automatically, so we rely on user input. Within the APM program, as part of a computation's result state, a floating point value has to be given that states a relative cost. Similarly, a component of the message data type tuple is the relative cost value associated with sending the message. The PolyAPM interpreters collect and count all these cost values in the course of a program execution. Together with the machine constants mentioned in the cost model the interpreters are able to calculate the cost value. It is a relative estimate of the expected run time.

We do not normalise the cost model's result to wall clock time for two reasons: the APM programs are not yet the target programs, they are unfinished and require further transformations before they are fit to run on a real machine. Therefore we think it unrealistic to expect a forecast of a future target program's run time based on an unfinished predecessor. The second reason is

Measured and predetermined values:

- w_{it} the workload of processor i at time t
- s_{ijt} the cost (read: size) of **sending** message j from processor i at t
- r_{ijt} the cost (read: size) of **receiving** message j from processor i at t
- $\overline{s}$ the normalisation factor to relate a *sending cost unit* to a workload unit such that 1 *workload unit* $\sim \overline{s} * 1$ *sending cost unit* (*obtained by experiments*)
- $\overline{r}$ like $\overline{s}$ but for the *receiving costs* (*obtained by experiments*)
- b the cost for a *barrier synchronisation* (*obtained by experiments*)
- m_s the startup cost to initiate a message (*obtained by experiments*)
- m_r the startup cost to receive a message (*obtained by experiments*)

Calculated values:

- $\#s_{it}$ number of messages sent by processor i at time t (the number of s_{ijt} for a given i and t)
- $\#r_{it}$ likewise for received messages
- c_{it} we calculate c_{it}, the communication cost of processor i at time t, as the sum of all sending costs plus the sending startup costs plus the receiving costs plus the barrier cost:

$$c_{it} = \left(\sum_{j=1}^{\#s_{it}} \overline{s} * s_{ijt} + m_s \right) + \left(\sum_{j=1}^{\#r_{it}} \overline{r} * r_{ijt} + m_r \right) + b$$

- $cost_t$ the cost of time step t is defined as

$$cost_t = \max_{i \in procs} (w_{it} + c_{it})$$

Final Cost Function:

for a fixed input size, number of processors and machine parameters:

$$cost = \sum_t cost_t$$

Figure 3.5: PolyAPM Cost Model

```
lcost t = (fromIntegral (t`div`2)*lfact +1 + 2)
ucost t = (fromIntegral ((t-1)`div`2)*ufact +1 +1 + 5)

body_s_sc:: (GlobalStateShM LUmem b c d, [Idx]) -> [Int]
             -> ((GlobalStateShM LUmem b c d, [Idx]), [Int])
body_s_sc (GStateShM (a,l,u) msgs
           ((Comps crange (ca:compsal)):clist),[n,maxp]) (idxlist@[t,rp,p]) =
  ((GStateShM mem msgs
    ((Comps crange ((ca//[(rp,cost+ca!rp)]):compsal)):clist), [n,maxp]), idxlist)
   where  (mem,cost) =
             if (t `mod` 2 == 0)
             then ((a, stmnt1 a l u (t,p,n), u), (lcost t)+2.0)
             else if (t+2)`div`2 == p
                  then ((a,l,u), 5.0)
                  else ((a,l,stmnt2 a l u (t,p,n)), (ucost t)+5.0)
          stmnt1 a l u (t,p,n) = ... omitted ...
          stmnt2 a l u (t,p,n) = ... omitted ...
```

Figure 3.6: LU SynCommAPM Body

that we don't need to. The whole point of this cost model is to be able to decide between several alternative code transformations based on the assumed performance impact. For this purpose a relative comparison suffices.

The following example shows how the user defined cost values are to be provided. The code is taken from the LU decomposition example for SynCommAPM of Section 6. Figure 3.6 shows two different computations, namely new values for L and U, that are performed at even resp. odd times. The result type is of `GlobalStateShM`, denoting a global state for a shared memory machine. It is a machine state that is aware of the cost mechanism. It contains the current memory, a list of pending messages and a list of `CostItem`s that hold the profiling data needed to compute the cost model prediction. Three types of cost items are used: the computation cost, the message receive cost and the message sending cost. Within the loop body, only the computation cost is updated. The one-dimensional cost array `ca` contains one floating point cost value per real processor. The computation cost of the current iteration is therefore added to the `rp`th element of `ca`. The computation cost is determined by the cost of the statement (`lcost` or `ucost`) plus the cost of evaluation the surrounding `if` guards. For the latter, we count the number of operations (2 for the outer `if` and 3 for the inner). This way, the empty statement in the first `then` branch of the second `if` gets a cost of 0 plus the 5 ticks of its `if` guards assigned. The values of `lcost` and `ucost` are determined by analysing the structure of the two statements and counting the number of operations in them.

The message generation deals with two types of cost: the cost of generating messages, which is added to the computation cost, and the cost of transmitting the message. Each message is accompanied by its *cost value* that corresponds to the amount of data contained in the message. This value is retrieved twice: first during message creation for the sending cost, and the second time during delivery for the receiving cost. This way, the computation of sent and receive costs are independent and if they don't match at the end of a program's execution, the interpreter may flag an error that message delivery is at fault.

```
generateMsg LU_blocked_SynCommAPM [t,rp,p] (n:maxp:_) (a,l,u)  =
    if(t`mod`2==0)
    then ([Msg (rp, to_p, t, t+1, L, (p,t`div`2+1), l!(p,t`div`2+1),1)
         | to_p <- [rp+1 .. maxp-1]], (rp, gen_cost_B1))
    else if (2*p-2>t)
         then ([Msg (rp, to_p, t, t+1, U, ((t+1)`div`2, p), u!((t+1)`div`2, p),1)
               | to_p <- [rp+1 .. maxp-1]],(rp, gen_cost_B1+3.0))
         else ([], (rp, 5.0))
   where gen_cost_B1 = fromIntegral ((maxp-1)-(rp+1))  -- msgs cost
                       + 2.0                           -- 1st if guard
```

Figure 3.7: LU SynCommAPM Message Generation

PolyAPM views a message as a tuple within the `Msg` type. Its last component is the message cost. An element of this type has the following form:

```
Msg (from, to, source time, destination time, domain, index, value, cost)
```

The message generation function of LU in Figure 3.7 sets the last component of the tuple accordingly. As the message's payload is only a single float, the cost value is set to 1. Note that the function `generateMsg` returns a pair of the generated message list and the generation cost. This cost is determined like the computation cost in the body and is added to the computation cost array `ca`.

The collection and calculation of the communication cost is performed by the SynCommAPM and SynDMAPM interpreters without further user interaction.

3.5 Dependence Analysis in Haskell

A prerequisite to writing the first APM program is to have a suitable parallelisation of the algorithm or input program. In case this parallelisation is not blatantly obvious, one often needs to determine the data dependences between computed and read array elements. These dependences can be analysed to determine the independent computations that may be executed in parallel.

PolyAPM was designed to be applicable to loop program transformations done with the polytope model (see Section 2.2.3). The polytope model will be used in the second case study in Chapter 6. Its scheduling and allocation methods need a formal definition of the program's dependences as their input.

There are many publications on dependence analysis for imperative programs ([Fea91, Ban93] and more), but as we suggest the use of Haskell as a specification language, we need to describe how dependences can be retrieved in this setting.

Data dependences between computations impose a partial order which any evaluation order must follow. The partial order is used to identify independent computations which can be executed in parallel. The aim of the dependence analysis is to obtain this partial order by an analysis of the source code. In the following, we present a simple algorithm to determine the flow dependences between array elements in array computations of a Haskell program. As Haskell is single-assignment, output- and anti-dependences do not exist.

Array computations in Haskell are often specified with the `array` construct. The array elements are usually defined by an array comprehension. Due to the fact that data type definitions can be mutually recursive, a set of array definitions may depend on each other. We define two arrays a and b to be in relation R, i.e., aRb, if and only if an element of b is being referenced

in the definition of a. Then, the weakly connected components with respect to R are the sets of arrays in which each array is either the source or the destination of a dependence. The array computations within a component are parallelised together.

Given a Haskell program, the first task is to determine these array component sets. This is done by the pseudo-algorithm in Figure 3.8.

Input: – Haskell-program with array definitions
Output: – Array component sets S_i, $i \in \mathbb{N}$

1. Construct a list L of all arrays in the program.
2. `while` there exists an unmarked array A in L `do`
3. Construct the weakly connected component S_A of arrays which includes A.
4. Mark all arrays of set S_A in list L
5. `output` all S_i

Figure 3.8: Array Component Set Determination

The second task is to determine all dependences within a set. They will be used later for the parallelisation of all computations within the set. The pseudo-algorithm for the dependence analysis is presented in Figure 3.9.

Input: – Haskell-program with array definitions
– Component set S
Output: – Set of all dependences with index spaces of set S

1. create empty set D
2. `foreach` array A in S
3. `foreach` array A' of S used in the definition of A
4. construct a dependence $A' \rightarrow A$; add it to set D, together with appropriate index space.
5. `output` D

Figure 3.9: Dependence Analysis Algorithm

The determination of the index space in Step 3 has to consider possible boolean restrictions of generators in Haskell's list comprehension. The original polytope model [Len93] is restricted to index spaces with defined by linear expressions in the loop bounds. For simplicity we use this model, i.e., the boolean predicates may only be linear relations of index variables. Extensions of the polytope model can deal with general `if` statements [Col95], which correspond to arbitrary boolean predicates in our context.

The above dependence analysis will be applied in Chapter 6 to the LU decomposition algorithm. We did not implement the analysis, but rather performed it manually according to the above pseudo-code. The result is a set of dependences. If they are to be used as input for a polytope based parallelisation with LooPo, the obtained dependences, together with index space and statement descriptions, have to be brought into a form that LooPo can process. Such a form is called a *LooPo specification*. Examples are shown in Appendix A and discussed in Chapter 6.

Chapter 4

Example of a PolyAPM Infrastructure

While the previous chapter explained the general PolyAPM development model and a set of synchronous example machines, we present in the present chapter an exemplary implementation of these machines as interpreters for their programs. We have implemented interpreters for SynAPM, SynCommAPM and SynDMAPM as Haskell modules. These interpreters are used in the development process to run the APM programs with real input data and determine program characteristics such as the cost value.

In Section 4.1 we describe in general terms how to write an APM program for our interpreters. Section 4.2 contains a detailed description of the application programmer's interface of our example APM implementations. The final APM program will have to be transformed to a program on a target architecture. How to do that we present by way of example using a Scali Linux cluster in Section 4.3. Finally, properties of the target architecture are captured in the set of cost model parameters. Section 4.4 illustrates the process of devising and performing benchmarks to determine the cost model constants for the Scali cluster.

4.1 Writing PolyAPM Programs

A PolyAPM program is a loop program with a body of array assignments. Depending on the APM the program is designed for, additional functions for communications may be needed. Of course, anyone using PolyAPM can extend PDG and PMT which, in turn, may impose more requirements for the program.

In this section we describe the program components and their requirements for the synchronous APM line as described in Section 3.3.4. That is, we present our example implementation of the PolyAPM machines SynAPM, SynCommAPM and SynDMAPM. It is conceivable to define completely different machines based on the PolyAPM ideas which would lead to different program properties. The motivation to present our implementation is to augment the general description of APMs with concrete examples and experiences.

Common to all presented programs is the loop structure comprising loop bounds and a loop body. They have to be of certain Haskell types so that the provided APM machine interpreters can use them. Figure 4.1 shows the structure of a PolyAPM program using all mandatory and optional functions.

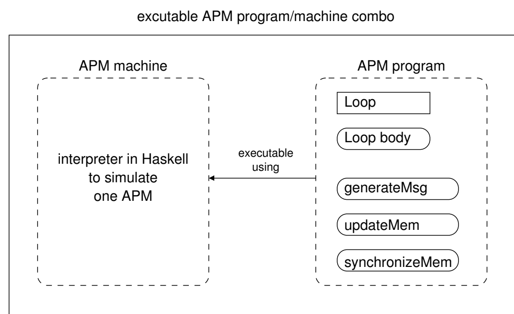


Figure 4.1: APM Program Structure

The following types are all defined in the `APM` module. Loops are defined by

```
-- loop: finitely many loop levels and one inner body
data Loop e b =  LP [LoopGen] (Body e b)
```

and consist of a list of loop generators (one for each loop level) and the loop body. A loop generator defines lower and upper bounds, the stride and whether the loop is parallel or sequential. The bounds are integer valued so that the loop is like a `for` loop (lower bound is inclusive, upper bound is exclusive). Bounds and strides may depend on two sets of integer values: loop variables of surrounding loops and other program values (i.e. structural parameters). They are therefore functions of these values. The predefined Haskell types are

```
data Par_annot = Par | Seq
type Idx   =  Int
type Lbd   =  Idx
type Ubd   =  Idx
type Step  =  Idx

type LoopGen = (Par_annot,
                ([Idx], [Idx]) -> Lbd,
                ([Idx], [Idx]) -> Ubd,
                ([Idx], [Idx]) -> Step)
```

The annotation is for the complete description of the loop nest but has no effect on the interpreters. Whether a loop is treated as parallel depends on the machine type. Of course, other APM implementations might behave differently.

The loop body has the following type:

```
data Body e b  =  BD (e -> b -> e)
```

A body is viewed as a function that takes the current machine state (of type `e`) and the loop variable values (of type `b`) to compute a new machine state as the result. The kind of state that has to be supplied differs from APM to APM. Some need distributed, some shared memory. Some have communications and need message queues, others do not. The specifics of the different machines are presented in detail in Section 4.2. While the state type is defined by the APM, the memory type as part of the state is not. The programmer has full control over the organisation of the abstract memory. This is possible because all functions that access the memory are user defined and the APM interpreters do not directly modify the memory. However, the user defined memory needs to reflect the APM's architecture: if it is shared memory, then the memory type has to comprise the entire program's data, otherwise just the local data.

In the collection of APMs presented here, we have up to three different auxiliary functions to modify and read the user defined memory. They are called from within the interpreters. To ensure that their types fit, three Haskell type classes have been defined. The user program needs to provide the corresponding instance declarations. The three classes are `Synchronizing`, `Sendable` and `Updatable`.

All of the above APMs require a function called `synchronizeMem` as defined by the class `Synchronizing`:

```
class Synchronizing machine mem where
    synchronizeMem:: machine -> mem -> mem
    synchronizeMem a = id
```

Its purpose is to reorganise global or local memory in a synchronised manner at the end of each time step. The class default is the identity which can be overloaded in case a program needs reorganisation. When this function is called the APM interpreter ensures that all body computations on all processors have completed and all messages have been sent. Some methods (e.g., multi-grid algorithms) require a global memory reorganisation that is independent of the computations as performed in the loop body. In a lot of cases, nothing has to be done here, which is why the default instance is the identity function.

One thing to note is that all instance declarations of the three APM functions are parameterised with a machine parameter. This is a discriminating data type for each APM program whose only constructor relates the instance declarations to their APM programs. The reason is that different APM programs might want to provide their own instances (in this case `synchronizeMem`) operating on the same memory data type. In the case of `synchronizeMem`, without the machine parameter we would have several instance declarations of the class `Synchronizing` with type `synchronizeMem:: mem -> mem`. If the `mem` type is identical in two or more cases, a compile time error prohibits the redeclaration of these instance declarations. The machine parameter make the otherwise identical instance types artificially different, so that several instance declarations using the same memory type become possible. It would not help to put the different instance declarations into separate modules as instance declarations are always globally im- and exported. The separate name spaces of modules do not refer to instance declarations.

Starting with `SynCommAPM`, SPMD-style communication is introduced. The APM programs need to exchange values among the processors they run on. While the message delivery is done by the APMs, message generation and the memory update are part of the APM program. Conceptually, the communication takes place at the end of each time step and is separate from the computation. Because of this, the communication code is not part of the loop program. Instead, along the lines of `synchronizeMem`, we have the functions `generateMsg` and `updateMem` to be defined by the APM program.

```
generateMsg:: machine -> [Idx] -> [Idx] -> virtprocmem -> ([Msg a b c],(Idx,Float))
updateMem:: machine -> Msg dom idx val -> realprocmem -> realprocmem
```

The function `generateMsg` takes the machine label, the list of current loop indices, the list of structural parameters and the memory. Since the function is part of the APM program, the memory structure is known so that the message data can be retrieved from memory. The result is a list of messages that are placed in the message queue. These messages will be delivered during the delivery phase just before the time step with the specified delivery time starts. Until then, the message is assumed to be buffered by the network.

The message structure is given in the `Msg` data type:

```
  data (Ix b, Show a, Show b, Show c) =>
--                      From To   SrcT DestT Dom Idx Val Cost
     Msg a b c = Msg (Proc,Proc,Idx, Idx,  a,  b,  c,  Float)
```

A message contains origin and destination of its delivery, both of which are defined by a processor id and a logical time. The data item is defined by its domain (a user defined data type used by `updateMem` to determine the part of the memory where the data item is to be stored), an index into the destination array (is left unused if the destination is a scalar variable) and finally the value. Associated with each message is the cost to send this message. During delivery the message costs are collected and integrated into the cost formula (see Section 3.4 for details).

On the receiving end, the function `updateMem` takes a message and the memory in order to make the message's data persistent in the memory.

The only functions that have to know about the structure of the memory are the loop body and the three auxiliary functions `synchronizeMem`, `updateMem` and `generateMsg`. In addition, there need to be functions that construct the input memory and preload the necessary input data as well as retrieve the computation result after the program execution. But none of the APM interpreter functions will ever access the memory directly as they have no knowledge of its structure.

4.2 The PolyAPM API

The PolyAPM example implementation of synchronous APMs provides a list of functions to be used by the APM program. In the previous section we have already described the functions `generateMsg`, `updateMem` and `synchronizeMem`. In the following we present the API of the APM Haskell module for use by APM programs.

4.2.1 Data Types

An APM program is encoded as a Haskell data type. The topmost structure is a loop, consisting of a list of loop generators and a loop body. A generator comprises three functions to compute the lower and upper bound of a loop and its stride. The arguments to these functions are of two kinds: one is a list of structure parameters, which are external values from the loop's point of view, and the other is a list of the loop variable values of the surrounding loops. These types have been described in Section 4.1.

All synchronous APM interpreters use at least the computation cost of the PolyAPM cost model. To record the cost, a one-dimensional floating point valued array is used. The index denotes the processor id. Every body computation has to update the corresponding cost value.

All arrays containing cost information are stored in a list of type `CostItem`. Programs for SynAPM and SynCommAPM update the computation cost directly. `SynDMAPM` programs with their independent local states first put the cost values into a list of type `CompCostList`, from where the APM interpreter collects them at the end of each time step and stores them in the global `CostItem` structure. While the `SynAPM` interpreter only uses the `Comps` component, `SynCommAPM` and `SynDMAPM` use all three of them.

```
type CompStat = LArray Int Float
type MsgStat  = LArray (Int,Int) [Float]

data CostItem =
       Comps      (Int,Int) [CompStat]
    |  RcvdMsgSzs ((Int,Int),(Int,Int)) [MsgStat]
    |  SentMsgSzs ((Int,Int),(Int,Int)) [MsgStat]
    | Void
```

All cost items have a list of one- or two-dimensional arrays. For each time step a new array is added at the head of the list. The cost array has exactly one entry per processor, while the other

two contain the message cost index by sending and receiving processor. The range for new arrays is provided as the first parameter within the data type.

The computations at each space/time point are simply accumulated in a single floating point number, so that we get the total computation cost for each processor at a time step. The number of computations is lost as the cost model does not require it. In contrast, the message costs are stored in lists with one entry per message. The reason is that we need more information than just the total cost. The length of this list equals the number of messages, which is used by the cost model. Furthermore, target machines may coalesce PolyAPM messages sent to the same target processor into one physical message, so that only one startup cost per target processor and not per PolyAPM message incurs. In this case, the cost model calculation needs to analyse the sent and received messages to infer the number of communication partners. Optional coalescing on the target machine is acknowledged by the PolyAPM cost model.

4.2.2 `SynAPM` Functions

To run a SynAPM program, we pass it to the SynAPM interpreter function which is called `runloop_s`. Its type is

```
runloop_s:: (Synchronizing m realprocmem) =>
            m -> Loop (realprocmem, [Idx], CompStat) [Idx]
              -> [Idx] -> (realprocmem, CostItem) -> (realprocmem, CostItem)
```

The only type class requirement is that the program's memory type (together with the program/machine identifier) is made an instance of class `Synchronizing`.

The function `runloop_s` takes the following parameters:

- a program identifier of any type `m`,
- a loop whose body function takes and returns a triple consisting of memory, a list of indices and the computation statistics,
- a list of indices containing the structure parameters (for each particular program, the order of the parameters in this list is constant), and
- a pair of the memory itself and the cost item for the computation statistics.

It returns the memory after the execution of the program and the new filled cost item. Both can then be inspected for the result of the program run. In other words, input and output are performed by writing the memory before and reading it after the computation.

Any function that accesses the memory has to be written as part of the APM program, since the APM interpreters have no knowledge of the structure and the semantics of the memory. This obviously includes a function that retrieves the computational results after the computation. However, the computation statistics are of a predefined structure. The APM module provides a function to calculate the PolyAPM cost as described in Section 3.4:

```
calc_costmodel:: CostModel -> [CostItem] -> String
```

This function takes as parameters a particular model for a specific machine and the list of cost items. In this context a `CostModel` is a Haskell record comprising a set of machine parameters. For a discussion of cost models and how to obtain them see Section 4.4. The result will be a string that contains a message about the computational and overall PolyAPM cost of the program run, as described by the second parameter.

4.2.3 SynCommAPM Functions

The introduction of communication affects the machine interpreter in several ways. The machine state is defined not only by the memory, but also by the content of message queues. Message generation and delivery have to be implemented and the cost model needs to take the communication cost into account.

First of all, a new interpreter function runloop_sc is introduced:

```
runloop_sc::
    (Updatable m b c d a, Synchronizing m a, Show d, Show c, Show b, Ix c) =>
    m -> Loop (GlobalStateShM a b c d,[Idx]) [Idx] -> [Idx]
      -> GlobalStateShM a b c d -> GlobalStateShM a b c d
```

This function differs in two main ways from runloop_s: for one, the memory and computation statistics have been combined with a global message queue into the type GlobalStateShM, and secondly there is the requirement of making the memory and message types an instance of Updatable. The latter is done by providing an implementation of updateMem that takes memory and a message and updates the memory with the message's content. The definition of GlobalStateShM is as follows:

```
data GlobalStateShM a b c d =
    GStateShM a           -- arbitrary memory
              [Msg b c d] -- global msg queue
              [CostItem]  -- cost model item list
```

In order for the communication mechanism to work, functions of the APM program and the APM interpreter have to be interleaved in a certain way.

The PolyAPM view is that, after each body execution, we might need to send newly computed values to other processors. This is done by the sender generating one message for each destination processor. Broadcasts and multicasts are not supported.

Messages are generated after each iteration of the body and immediately appended to the global message queue. The communication phase takes place at the end of each time step. During it, the global message queue is searched for messages whose destination time will be the next time step. These messages are removed from the queue and delivered. In case there are sequential loops within the processor (p) loop, there are possibly several body iterations per time step.

The reason for generating messages after each iteration is simplicity: the loop indices of the last generation are still there, right along with the values to be communicated. So it is feasible to communicate right at this point. If a coarser communication granularity were selected (such as time step-wise), any program transformation that changes the time loop later on would also require a change of message generation. Thus there is an advantage of doing it at the smallest possible granularity, which is right after every body iteration.

The body function of type BD (e -> b -> e) as part of the loop program consists of a composition of the real body function and a predefined message generation function genMsg_sc. The idea is to generate a message containing a new value as soon as it is computed. In the above body type, the type variable e is the state and b is the list of loop indices. The genMsg_sc function also needs the loop indices, so that the user defined body function has the type e -> b -> (e,b), thus passing on state and loop variables. The function genMsg_sc consumes the loop variables and returns just the state, thus its type (e,b) -> e. If combined together, they yield a body function of type e -> b -> e. In all these types, the machine parameter is omitted.

With a genMsg_sc function of the following type

```
genMsg_sc::(Sendable m a1 b c a, Show a1, Show b, Show c) =>
             m -> ((GlobalStateShM a a1 b c, [Idx]), [Idx]) ->
             (GlobalStateShM a a1 b c, [Idx])
```

a typical loop body definition looks like this:

```
-- like function composition, but combines a second
-- function with two parameters
comb :: (c -> d) -> (a -> b -> c) -> (a -> b -> d)
(f ‘comb‘ g) x y  =  f (g x y)

loop_s_sc = LP loop_bounds_s_sc
               (BD ((genMsg_sc LU_blocked_SynCommAPM ) ‘comb‘ body_s_sc))
```

The genMsg_sc function calls the user defined function generateMsg to generate new messages based on state and indices, and places them in the outgoing message queue. To make sure that generateMsg fits into the APM function framework, we introduce the type class Sendable:

```
class (Ix b) => Sendable machine a b c mem where
    generateMsg:: machine -> [Idx] -> [Idx] -> mem -> ([Msg a b c],(Idx,Float))
```

An instance uses the system's memory and all types involved in a message. The first list of Idx contains the loop indices, the second all structural parameters. As part of an APM program, the function knows about the memory's structure and can access it to retrieve the computed values for a generated message. The function generates one message for each receiver and places them in the result list. The messages are appended to the processor's local message queue from where they are collected during the next communication phase. In addition to the messages, the function returns processor id and message generation costs. The interpreter adds this cost to the computation cost of the designated processor.

The counterpart of generateMsg is defined in the type class Updatable:

```
  class Updatable machine dom idx val realprocmem where
    updateMem:: machine -> Msg dom idx val -> realprocmem -> realprocmem
```

The updateMem function is called from the communication delivery system to make the payload of a message persistent in memory. This function also has to be part of the APM program as it needs knowledge about the memory structure. Its parameters are a message and the memory and it returns the updated memory.

4.2.4 SynDMAPM Functions

The interpreter of SynDMAPM employs the distributed memory model, where each processor has its own local memory. Remote access to it can only be done by means of communication. To implement this memory type, SynDMAPM needs a new global state type, the GlobalStateDM:

```
data CompCostList = CCL [Float]

data LocalState a b c d = LState a [Msg b c d] CompCostList

data GlobalStateDM a b c d = GStateDM (LArray Int a)
                                      [Msg b c d]
                                      [CostItem]
```

The difference from SynCommAPM's GlobalStateShM is the memory type: we now have an array of memories of type a, while SynCommAPM's memory type was free-style. However, the current implementations are more restrictive in that they require the machine state to be (GlobalStateDM (LocalState a b c d) b c d). This way, all generated messages will be placed in the local message queues, which are a part of the LocalState type, before they are moved to the global message queue in the immediately following communication phase. In addition, the LocalState contains a CompCostList in which the body functions place the computation cost. At the end of each time step the APM interpreter removes the values from all computation cost lists and stores them in the global computation CostItem. This is necessary as the body computations get only their own local state and have no access to the global state, so that they cannot access the CostItem list.

4.2.5 An APM Example Program

The following is a small example program for SynAPM. It consists of some memory type, a program label type (necessary for instance declarations), the loops, a loop body and an instance declaration for synchronizeMem. It iterates five times over a one-dimensional array of floats, each time computing the square root of every array element.

```
type Mem = LArray Int Float

data Example_SynAPM = Example_SynAPM

loop_s = LP [(Seq, \([],(n:_))->0,
                   \([],(n:_))->5,
                   \([],(n:_))->1),
              (Par, \((t:_),(n:_))->0,
                    \((t:_),(n:_))->n,
                    \((t:_),(n:_))->1)]
              (BD body_s)

body_s::(Mem,[Idx],CompStat) -> [Idx] -> (Mem,[Idx],CompStat)
body_s (a, splist@(n:_), ca) (t:p:_) =
    (a//[(p,sqrt (a!p))], splist, ca//[(p,1.0+ca!p)])

instance Synchronizing Example_SynAPM Mem where
    synchronizeMem Example_SynAPM a = a
```

The loop construct comprises the generator list and the loop body function. The first loop generator defines a sequential outer loop ranging from 0 to 4, stride 1 (the upper bound is exclusive). The bounds and the stride are defined as functions, taking a list of outer loop variables and a list of structure parameters. As the sequential loop is the outermost one, its loop variable list is empty. Its structure parameter list matches the first element as n. Possible further structure parameters are ignored at this point by the underscore pattern _.

The second loop is tagged parallel and ranges from 0 to $n-1$, stride 1. It is the inner one, so it may use the outer loop variable. Therefore, the first list is (t:_), referring to the sequential loop variable as t.

The body takes a triple comprising the memory, the structure parameter list and the cost array, as well as a list of the loop variables. It returns the updated triple: the pth array element of a is replaced by its square root, and the pth element of the cost array is increased by one, the cost for the square root operation.

Finally, each SynAPM program needs to define an instance of `Synchronizing`. Most APM programs, like the one in this example, don't need it and define it as the identity function on the memory.

4.3 Adapting APM Programs to Target Architectures

APMs can be adapted to any parallel target architecture. The most abstract APMs we present have shared memory. With the default being one shared memory, we view memory distribution as an additional property that an architecture may have. Thus it will be introduced in a more specific APM. PolyAPM itself has no bias towards either shared memory or distributed memory machines. In the past, massive parallelism was only possible on distributed memory machines where some installations have several thousands of processors. Shared memory machines have a severe performance bottleneck in the interconnection network, because all processors need to have fast access to the memory. Because of this limitation, until a few years ago there existed no shared memory machines with more than 32 processors. However, more recently machines with up to 100 processor appeared on the market. Programs for shared memory machines do not need communication, thus easing the programming. A compromise between both worlds are *Non-Uniform Memory Access* machines (NUMA), that consist of a cluster of shared memory machines. A software layer provides shared memory programming without explicit communication, but the performance difference between a local and a remote memory access is measurable, hence the name.

All performance measurements in this work have been conducted on our distributed memory machine, a 32 node dual Pentium III 1 GHz Linux cluster with a high-performance SCI communication network by Scali [Sca]. The system software for the SCI network provides a low level communication library. However, code for this library is difficult to write and not portable. Therefore, Scali provides its own implementation of MPI in the ScaMPI library on top of SCI. The alternative MPI implementation MPICH was not available for these experiments. The LAM MPI library [LAM] does not support the SCI network and can therefore only be used on our cluster with the 100 MBit Ethernet network. Since Ethernet has a comparatively high latency we do not use LAM.

4.3.1 The `mpi_apm` Library

The whole point of the PolyAPM project is to ease the development of parallel programs while striving for code quality by choosing the best program among several alternatives. One particularly difficult aspect of writing a parallel program is to specify the communications. In order to ease the effort in this regard, the abstract programs perform one-sided communications. This means that only the sending side knows which values are communicated to which destination. The receiver just enters the communication phase and is ready to receive messages, if any. It has no knowledge beforehand about incoming messages. The communication partners are determined by following the data dependences. For many algorithms, it is possible to program a static communication scheme where also the receivers know how many messages to expect during each communication phase. Such a scheme would incur less overhead as but we trade this overhead for easier programming (just the sending routine has to be implemented) and the possibility to deal with algorithms that have a dynamic communication structure. Furthermore, the absence of explicit blocking receives eliminates a common source for deadlocks in parallel programs.

The difficulty lies in supporting this programming style with MPI. Version 2 of the MPI standard defines remote memory access as a way to implement one-sided calls, but only few MPI libraries implement this. In particular, ScaMPI does not. To solve this problem we have implemented the `mpi_apm` library (see Section 4.3.1 for details) that uses collective operations to simulate one-sided calls. All processors independently call the communication function at the

same logical time step. Only the senders know of the messages about to be sent. Messages are exchanged and during this process the processors are synchronised.

A Communication Mechanism for One-Sided Calls and MPI 1

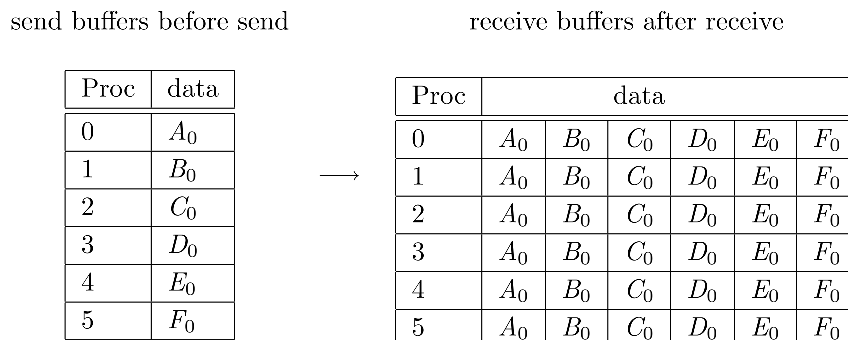
To improve efficiency we accumulate the possibly large number of messages in message buffers, thus trading the number of messages for the size of messages. This trade-off is favourable with most current communication networks. The *data* to be sent from processor i to processor j is just the MPI message buffer containing all PolyAPM messages that are to be sent between the two at this time. This way, instead of a large number of messages that might be sent between two processors in each time step, we have only two at most, i.e., one in each direction.

A common case in the parallelisation of loop programs is to use the owner computes rule. If other processors need the new value, then it has to be communicated. Each processor just sends its newly computed values so that the outbound message buffer is identical for all destinations. The only exception is if the values from processor i are not needed at all by processor j. In this case, no communication takes place between the two.

In the following we assume one send buffer of a fixed and suitably large size s per processor. The maximal number of physically available processors is denoted by $maxp$, so that the processors are numbered from 0 to $maxp-1$. The communication mechanism described above requires $(maxp-1) * s$ receive buffers on each processor. There are several ways to implement this mechanism using MPI collective operations.

Implementing the Mechanism with `MPI_Allgather`

If we disregard the fact that not all pairwise communications have to take place, the mechanism fits exactly the MPI call `MPI_Allgather` (see Figure 4.2).



send buffers before send

Proc	data
0	A_0
1	B_0
2	C_0
3	D_0
4	E_0
5	F_0

$\longrightarrow$

receive buffers after receive

Proc	data					
0	A_0	B_0	C_0	D_0	E_0	F_0
1	A_0	B_0	C_0	D_0	E_0	F_0
2	A_0	B_0	C_0	D_0	E_0	F_0
3	A_0	B_0	C_0	D_0	E_0	F_0
4	A_0	B_0	C_0	D_0	E_0	F_0
5	A_0	B_0	C_0	D_0	E_0	F_0

Figure 4.2: The MPI Call `MPI_Allgather`

The contents of all send buffers are distributed across all processors. The MPI call takes as parameters the send buffer as well as a send data type and the number of data items to be sent, so that the buffer is not sent as a whole if it is not full. The small PolyAPM messages that are stored consecutively within the send buffer are defined as a new MPI data type. The number of items to be sent is in terms of this data type. There is only one such number per sender, but it may be different for different senders. It is not possible for one sender to send different amounts of data to different receivers.

The problem is that, in the one-sided call scheme, other processors do not know how many messages they will receive from their communication partners. Therefore we use two allgathers. The first distributes the buffer fill size of each sender across all processors, so that all of them have an integer array containing the number of messages they are about to receive from each partner.

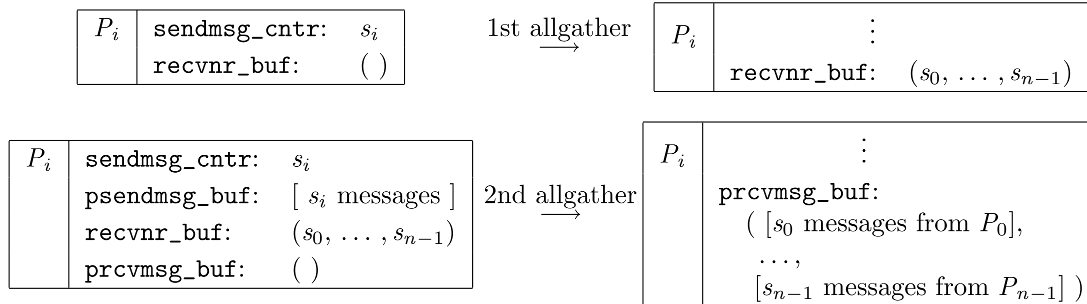


Figure 4.3: Allgather Data Exchange

The amount of data received from each partner is constant: one integer. Then a second allgather that communicates the contents of all send buffers can take place. It uses the just acquired information about how many messages each partner will send. The overhead for the first allgather is constant, but if we omit it and always communicate the entire send buffer, the additional volume is proportional to the difference between the average send buffer usage and the send buffer size. In general, this difference grows with the problem size, so the price for the additional allgather is worth it.

The data exchange of the two allgathers as it is implemented by the C+MPI code in Figure 4.4 is depicted in Figure 4.3. A box contains the buffers and variables that are used for the allgather call. Counters are integer valued and buffers may be empty, have a fixed number of elements (enclosed by parentheses) or a variable number of elements (enclosed by square brackets). The expression "s_i messages" denotes a sequence of s_i messages. On the right hand side in the boxes after the call, only data items that were changed by the call are displayed.

The first allgather just distributes the `sendmsg_cntr` of all processors. They comprise the array that says how many messages are to be received by any processor in the next allgather. The second allgather distributes the `psendmsg_buf`, and all processors place all received messages in `prcvmsg_buf`. The `prcvmsg_buf` has exactly n elements, but each element is itself an imaginary buffer of variable size. Note that, due to the nature of the allgather, the content of `prcvmsg_buf` is identical on all processors.

The C+MPI code in Figure 4.4 implements the quasi one-sided PolyAPM communication using `MPI_Allgather`. Both, send and receive buffers are just single arrays. We use displacements to separate the data for different partners. As the number of PolyAPM messages (read: the length of the one coalesced MPI message) differs across the processors and invocations of the function, the second call is an `MPI_Allgatherv` (for messages with a variable message length). The actual lengths of the messages to be received are stored in the `rcvnr_buf` array that is filled in the first allgather.

Implementing the Mechanism with `MPI_Alltoall`

The problem with allgather has already been mentioned: if a processor sends data, it has to send the same data to all partners. In unbalanced communication patterns many unnecessary messages are created (see Section 6 for an example of unbalanced structures).

The next more general MPI collective call is an `MPI_Alltoall`. It provides different send buffers for each communication partner. Its communication structure is depicted in Figure 4.5. Each processor has as many send buffers as there are processors in the system. Buffer i ($0 \leq i < maxp$) is sent to processor i. The receiving part is analogous to an allgather. This communication pattern is more general than we need: if a processor sends messages at some time, all receivers get the same data from it. We only want to be able to exclude some partners from receiving the message for efficiency concerns. Similarly to the receive buffer organisation in allgather, the

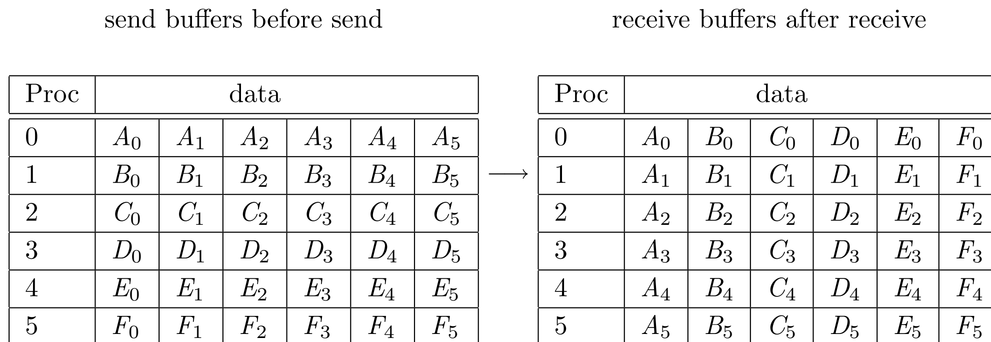
```
void send_msgs_gather(int n,int maxp, int sendmsg_cntr,
               int sendnr_buf[],int displ[],
               int sendbuf_size,int rcvnr_buf[],
               struct Msg psendmsg_buf[],
               struct Msg prcvmsg_buf[],
               MPI_Comm  mycomm){
  int i,j;

  for(i=0; i<maxp; i++){
    displ[i] = i*sendbuf_size;
  }

  MPI_Allgather(&sendmsg_cntr,1,MPI_INT,rcvnr_buf,1,MPI_INT,mycomm);
  MPI_Allgatherv(psendmsg_buf,sendmsg_cntr,PMsgType,
                 prcvmsg_buf,rcvnr_buf,displ,PMsgType,mycomm);

  /* Collect new messages */
  for(i=0; i<maxp; i++){
    for(j=0; j<rcvnr_buf[i];j++){
      updateMem_with_msg(prcvmsg_buf[displ[i]+j]);
    }
  }
}
```

Figure 4.4: C+MPI Code for Allgather Message Exchange



send buffers before send

Proc	data					
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	B_3	B_4	B_5
2	C_0	C_1	C_2	C_3	C_4	C_5
3	D_0	D_1	D_2	D_3	D_4	D_5
4	E_0	E_1	E_2	E_3	E_4	E_5
5	F_0	F_1	F_2	F_3	F_4	F_5

$\longrightarrow$

receive buffers after receive

Proc	data					
0	A_0	B_0	C_0	D_0	E_0	F_0
1	A_1	B_1	C_1	D_1	E_1	F_1
2	A_2	B_2	C_2	D_2	E_2	F_2
3	A_3	B_3	C_3	D_3	E_3	F_3
4	A_4	B_4	C_4	D_4	E_4	F_4
5	A_5	B_5	C_5	D_5	E_5	F_5

Figure 4.5: The MPI Call `MPI_alltoall`

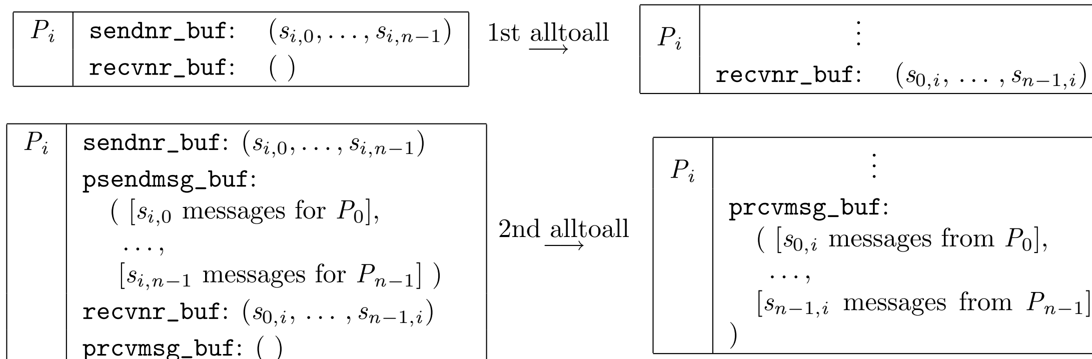


Figure 4.6: Alltoall Data Exchange

different send buffers in alltoall are actually combined to one array with a displacement array pointing at the starts of the different send points. However, since a sender sends the identical data to all its receivers, there is only one such send point for all receivers. The send displacements always point to the beginning of the send, so that no space is wasted by replicating the send buffer for each receiver.

But the alltoall call does not check this special case and considers the send buffers distinct, each with a different sent counter (here in the array `sendnr_buf`). This way we can mask out some processors by setting their send counter to 0. On the other hand, if one processor sends to several partners, the MPI run time system does not know that the messages have the same content, thus they need to be sent separately over the network. In this regard the alltoall is less efficient than the allgather. But experiments have shown that, especially with unbalanced problems like LU decomposition, the run time behaviour of `send_msgs_a2a` is significantly better.

Figure 4.6 shows the data exchange of processor i before and after the alltoall calls. The difference to the allgather exchange is that the sender has potentially different data to send to different destinations, so that the `prcvmsg_buf` after the second alltoall is not identical on all processors. This exchange is implemented by the C+MPI code in Figure 4.7.

Other Implementation Options

The communication mechanisms based on alltoall and allgather only simulate one-sided communications. The receiver still has to enable potential message exchange during the communication phase by calling either of `send_msgs_gather` or `send_msgs_a2a`. Communication without any participation at the receiving application's end is not possible.

The MPI 2 standard defines remote memory access (RMA) functions that allow just that: local memory segments of distributed machines are made available to other processors for direct writing. Functions called `MPI_Put` and `MPI_Get` may be used to access and update remote memory. The call structure is asymmetric, i.e., the MPI application on the corresponding end does not have to issue any MPI call. The underlying MPI infrastructure takes care of this. However, the mechanism is not completely unlike the two presented above, only the abstraction layer is different. The communication partner also has to be ready to receive communication requests that it doesn't know of beforehand. The difference is only that the MPI application does not need to participate. Thus, the programming of RMA is simpler, but most likely not more efficient.

Unfortunately, the ScaMPI libraries on our cluster do not implement the RMA functions of MPI 2.

```
void send_msgs_a2a(int n,int maxp, int sendmsg_cntr,
               int sendnr_buf[],int sdispl[],
               int sendbuf_size,int rcvnr_buf[],
               struct Msg psendmsg_buf[],
               struct Msg prcvmsg_buf[],
               MPI_Comm mycomm){
  int i,j;
  int rdispl[maxp];

  for(i=0; i<maxp; i++){
    sdispl[i] = 0; /* if we send, then always the same data */
    rdispl[i] = i*sendbuf_size;
  }

  MPI_Alltoall(sendnr_buf,1,MPI_INT,
               rcvnr_buf,1,MPI_INT,
               mycomm);
  MPI_Alltoallv(psendmsg_buf,sendnr_buf,sdispl,PMsgType,
                prcvmsg_buf, rcvnr_buf,  rdispl,PMsgType,
                mycomm);

  /* Collect new messages */
  for(i=0; i<maxp; i++){
    for(j=0; j<rcvnr_buf[i];j++){
      updateMem_with_msg(prcvmsg_buf[rdispl[i]+j]);
    }
  }
}
```

Figure 4.7: C+MPI Code for Alltoall Message Exchange

```
 1 initialisation
 2
 3 FOR t1 = ...
 4    FOR p1 = ...
 5       FOR t2 = ...
 6         ...
 7           FOR tn = ...
 8             loop body body_s_dm
 9             message generation generateMsg
10           ENDFOR tn
11         ...
12       ENDFOR t2
13    ENDFOR p1
14    sending and receiving messages (APM interpreter)
15    updating memory with received values updateMem
16    memory synchronisation synchronizeMem
17 ENDFOR t1
18
19 gathering results
```

Figure 4.8: Operational Structure of a SynDMAPM Program

Adapting a SynDMAPM Program to the mpi_apm Library

The SynDMAPM machine was designed to be quite similar to a real parallel machine with distributed memory in order to ease the transformation of programs between them. Let's consider again the main parts of an APM program: type definitions, initial value declarations, the loop program and the synchronizeMem, updateMem and generateMsg functions.

Most Haskell types cannot be translated directly to C. However, the program domain that we deal with chiefly uses integers and floating point numbers, usually organised as scalars or arrays, and these types have a direct correspondence in C. Haskell tuples can be represented in C either as records or arrays.

The structure of a SynDMAPM program is displayed in Figure 4.8. The initialisation and result gathering in lines 1 and 19 are very application specific. It is up to the programmer to care for a suitable data layout so that the process of distributing and gathering data is simple and easily translatable into the target language. However, the definition of the PolyAPM message type and the creation of communication buffers for the `mpi_apm` library is performed by the `mpi_apm` function `define_PMsgType`.

The program pattern from lines 3 through 17 can be easily translated into any imperative language. The function calls within the pattern (here: emphasised by italics) are translated to C and inserted into the pattern. Figure 4.8 contains five such function calls of which only the first one may require significant translation work.

In the PolyAPM framework we deliberately do not place restrictions on the structure of loop bodies for the APM programs in order to provide a very general approach. However, if the programmer uses that freedom to write a loop body in a very functional style, a high price for a translation to a corresponding imperative loop body has to be paid. On the other hand, restrictions on programs (e.g., as the polytope model imposes [Len93]) may well result in an easy translation. We had no difficulties to translate any loop body to C that stemmed from a numerical algorithm expressed in Haskell.

The next function is the message generation in line 9. The Haskell APM functions generate lists of messages that the APM interpreters place in the global message queue. In the C program

the messages have to be placed in an mpi_apm message buffer. The difference between the two is that the APM program generates one message per data item and destination, whereas the mpi_apm library stores each data item once in the message buffer and stores the destinations in the sendnr_buf array (the destination times are neglected by the mpi_apm library). Thus, a processor in an APM program is able to send different messages to different destinations. Using the mpi_apm library, a processor will always send the same messages to all its communication partners. The following example illustrates the transformation process.

```
instance Sendable Foo Foo_Dom (Int,Int) Float Foo_mem where
  generateMsg FooC [t,rp,p] splist (a,b)  =
     [Msg (rp, to_p, t, to_tm, dom, xidx, yidx, val, 1)
      | (to_p,to_tm) <- [(x,x+3)| x<- [rp..maxp-1]]
     ]
     where dom             = if t `mod` 2 == 0 then A else B
           (xidx,yidx,val) = if dom == A
                             then (p,    2*t, a!(p,2*t))
                             else (2*t+1,p  , b!(2*t+1, p))
```

Figure 4.9: An Exemplary generateMsg APM Function

The exemplary APM function generateMsg in Figure 4.9 belongs to an APM program named Foo (this is the machine parameter, see Section 4.1). It generates messages containing floating point values from two-dimensional, integer indexed data structures. These data structures are enumerated by the type Foo_Dom and values of structures A and B are actually sent. In the example the message content depends on the value of t. A list comprehension ranging over the destination processors creates one such message for every destination.

The C code in Figure 4.10 is a translation of the above APM function. First, depending on t, the message domain is selected. Then the corresponding indices and the value are determined. This information is filled into one message slot of the send buffer and the respective counter is increased. A for loop that corresponds to the above list comprehension tags the entries of destination processors in the sendnr_buf so that the message is being sent there during the next communication phase.

```
dom = t%2==0 ? A : B ;
if(dom==A){
  xidx = p;     yidx = 2*t; val = a!(p,2*t);
} else {
  xidx = 2*t+1; yidx =  p;  val = b!(2*t+1, p);
}
psendmsg_buf[sendmsg_cntr].msgdomain = dom ;
psendmsg_buf[sendmsg_cntr].xidx      = xidx ;
psendmsg_buf[sendmsg_cntr].yidx      = yidx ;
psendmsg_buf[sendmsg_cntr].value     = val ;
sendmsg_cntr++;

for(x=rp; x<maxp; x++){
  to_p            = x;
  to_tm           = x+3; /* not needed */
  sendnr_buf[to_p] = 1;
}
```

Figure 4.10: An Exemplary mpi_apm C Function

The remaining functions are easy to transform. In line 14 of the program structure the message exchange takes place. This is done by the APM interpreter and initiated by a call of `sendmsgs_a2a`.

The memory update in line 15 refers to the APM function `updateMem`. The corresponding C function is just a `for` loop ranging over the receive buffer and updates the memory according to the message content.

Finally line 16 refers to the memory synchronisation function. Its content is application specific so that no general advice for a transformation can be given. In most cases, this function will be the identity so that it may be ignored for the transformation to C. An example for a non-identity synchronisation is presented in Chapter 5.

The above example shows that there is a syntactical difference between the APM functions and the corresponding `mpi_apm` C functions, but that on a semantic level the difference is small enough to make a transformation between the two easy. The presented PolyAPM implementation is a proof of concept so that we did not invest more work to ease the transformation on a syntactical level.

4.4 Calibrating the Cost Model for `send_msgs_a2a`

The PolyAPM cost model, as presented in Section 3.4, is intended to give a rough cost estimate of a program's execution time for a given input on a specific, real machine. The aim is to provide some guidance when deciding between different program transformations. As the ultimate goal is to produce a fast program on a real machine, the cost model tries to predict the future run time behaviour even for an abstract program.

Section 3.4 motivates our empirical approach. The interpreters execute abstract programs with real input data and collect run time statistics about the computation and the communications. In order to compute the cost value for a real machine from these statistics, we need to know about some machine properties that relate different statistical data.

Remember that a tick is defined as the time that is necessary to multiply two floating point numbers. The calibration process is needed to determine the values of the normalisation factors $\overline{s}$, $\overline{r}$, the message startup costs m_s and m_r and the barrier cost b for a given machine and communication library. Since the cost model should provide a performance prediction, these values have to be adapted whenever a change in the library or hardware layer may affect the run time behaviour. Furthermore, if programming models do not need the full generality of the PolyAPM cost model, they may set unneeded values to 0. As an example, if messages are sent in a blocking style, the processors may be implicitly synchronised so that no explicit barrier is needed. In that case the barrier cost b is set to 0.

We will now describe the process of obtaining the PolyAPM cost model parameters for the Scali Linux cluster that has been described in detail in Section 4.3. The `send_msgs_a2a` function from the `mpi_apm` library is used for message exchange.

4.4.1 Determining the Computation Cost

To get the cost for a computation we have performed 10^8 array multiplications in a loop on one of the cluster nodes. This took 8.1973160 seconds, so that the time for one computation is $c = 8.197316 * 10^{-8}$ seconds. The factor c will later be used to normalise run times of communication benchmarks to cost model *ticks*.

We have compared operation times of addition, maximum and *ceildiv* (i.e., $\lceil \frac{a}{b} \rceil$) to the multiplication time. All the different run times were surprisingly similar. However it did make a difference whether one operand was recently used so that one can expect it to reside in a processor

register or at least in the processor cache. Therefore the memory access is the governing factor. Experimental results that we conducted show that on the average, one of two operands has to be fetched from memory, while the other is usually in the cache. Because of this, we assign all of the above operations the cost of one tick when determining the computation cost within an APM program.

4.4.2 Before Determining the Communication Costs

The PolyAPM cost model deals with send, receive and barrier costs independently and combines their cost. It is unfortunate that the function `send_msgs_a2a` does all the communications on both the sending and the receiving end, within a global alltoall call. This makes it difficult to find the calibration parameters for this software environment.

The first distinction has to be made between constant costs and variable costs, where the variation is with respect to the number of messages. Consider an MPI program running on n processors in parallel. Some fraction of the communication cost will be constant in every communication phase. Any synchronisation cost is an example of this. The rest of the communication cost increases if number and lengths of the messages increases. However, both the constant and the variable cost grow with an increasing number of processors.

A typical time step in a PolyAPM program consists of three parts:

1. computations of values
2. filling of the send buffers
3. `send_msgs_a2a`

Parts 2 and 3 comprise the communication. Part 2 certainly imposes a variable cost and part 3's cost is a combination of variable and fixed costs. The function `send_msgs_a2a` in turn can be separated into four phases:

1. initialisation of displacement arrays (these are arrays of offsets (or displacements) that MPI uses to access multiple send buffers within one C array)
2. first alltoall to exchange the number of messages
3. second alltoall to exchange the messages themselves
4. unmarshalling the messages and storing their content in memory

Phases 1 and 2 impose constant cost, phases 3 and 4 a cost which varies with the number and lengths of the messages. Next we present the measurement of the constant cost, then the measurement of the variable cost, which is done by benchmarking the complete communications and subtracting the constant cost.

4.4.3 Measuring the Constant Communication Cost

The benchmarking setup is simple: only the program parts that contribute a constant to the communication cost – phases 1 and 2 – remain in the test program. All other parts and steps are commented out. In our benchmarks we measure the time of 1000 `send_msgs_a2a` calls, each being the constant cost for different values of n processors. The results are depicted in Figure 4.11. The analysis of the data is performed with statistical methods, but the graph already suggests a linear relationship between the constant cost imposed by the communications between varying number of processors.

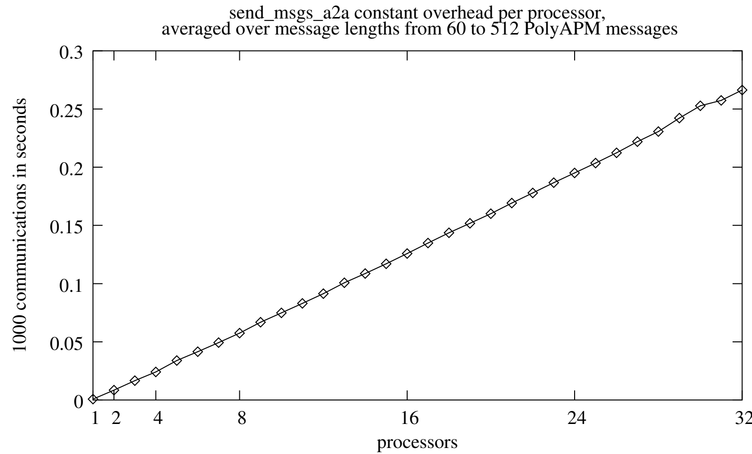


Figure 4.11: `send_msgs_a2a` Constant Overhead per Processor

A detailed analysis shows that there is a fixed cost associated with the communication of two processors and, for every additional processor participating in the communication, the same cost has to be added. This cost is 0.0085554 seconds for 1000 communications, which leads to the relation:

$$\text{constant cost with } n \text{ processors} = (n-1) * 0.0085554s * 10^{-3} \tag{4.1}$$

For all further calibration benchmarks with `send_msgs_a2a` we subtract the constant cost as defined by Equation 4.1.

4.4.4 Measuring the Variable Communication Cost

As the alltoall send does all sends, receives and implicit synchronisation in one MPI call, we have to benchmark the run times of specific communication patterns in order to separate their costs. The idea is to have one primary processor perform only one kind of communication (either send or receive), but many of them, whereas all other processors do the opposite operation just once. As the number of processors grows, the primary processor will dominate the run time. Suppose we want to measure receive times. On n processors, the primary processor will perform $n-1$ receives. We are interested in the linear relation between the number of PolyAPM message receives to run time. Therefore we perform the timings for 1 to 512 PolyAPM messages, each accounting for roughly 20 bytes. This series of results is expected to yield a linear relation, but most likely the "line" will be different for different numbers of processors. In other words, just to measure the cost for receiving a message we need to perform many benchmarks to get a family of linear relations, from which we should be able to the deduce the cost for receiving a single message. The same applies to the inverse, the cost for sending a message.

Figure 4.12 illustrates the setup to measure the receive costs for $n = 5$. Processor 4 is the primary one. In the communication phase between time 0 and time 1, we have $n - 1$ sends to the primary processor. As n increases, the number of receives by the primary processor increases. The corresponding situation for send costs is presented in Figure 4.13.

We will first deal with determining the receive costs. Figure 4.14 shows the experimental results we obtained by performing the receive benchmark on 8 nodes, so that 7 nodes send their messages of varying length to the 8th. The x-axis ranges over the number of PolyAPM messages, from 1 to 512. The y-axis denotes seconds for 1000 such sends. One can see that from about 60 messages onward the results lie on an almost straight line, while for fewer messages the times

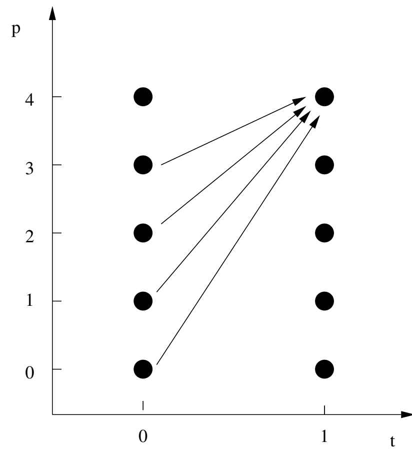


Figure 4.12: Measuring Receive Costs

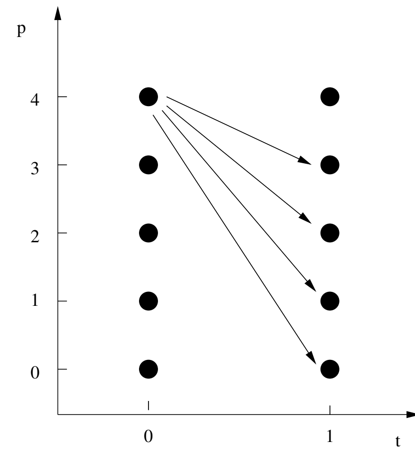


Figure 4.13: Measuring Send Costs

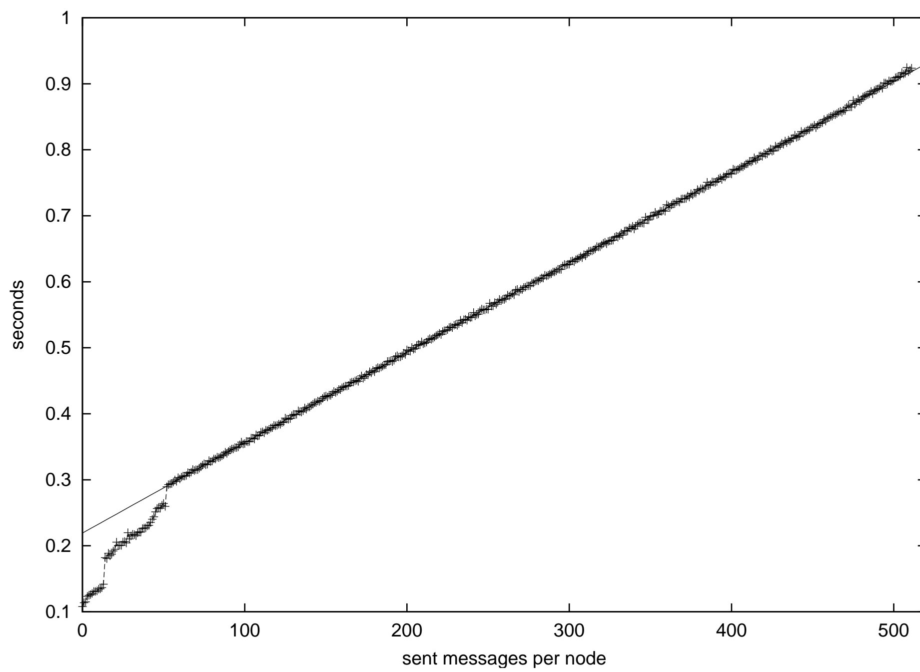


Figure 4.14: Linear Approximation of Receive Costs on 8 Nodes

are well below that line. As the reason for this we assume that small message numbers do not completely fill buffers in the MPI or network layer, so that overhead for buffer management does not yet occur. Since we expect in practise rather more than fewer messages and do not want to over-complicate things, we decide to deal only with cases where there are more than 60 messages. The displayed pattern is almost identical for other numbers of processors, and the straight linear behaviour goes on beyond 512 messages.

In order to analyse the data we need a linear approximation of the data in the upper, linear part. This approximation is already included in Figure 4.14. It was obtained by the statistical *least squares* method. For our experiments, we have written scripts that feed the benchmark data into Maple [Map] to obtain the linear equations. The data points for message lengths between 0 and 60 are neglected.

Another problem is a property of the least squares method: data points that are way off and could be considered erroneous have a disproportional effect on the result. We cannot explain such erroneous data points, but the lack of real time behaviour of the Linux operating system on the nodes may be the reason for some of them. It is always possible for some system daemon to consume suddenly CPU time and mess up the benchmark result. In theory, a single wrong data point can alter the result arbitrarily. Our experience is that we have only very few erroneous points per benchmark, but we prefer to remove them to get a more accurate approximation.

All our benchmarks use message lengths from 0 to 512 PolyAPM messages. After removing the first 60 points usually we haven't observed more than 3 erroneous data points. To be on the safe side, we remove the 10 worst fitting points before determining the final linear approximation.

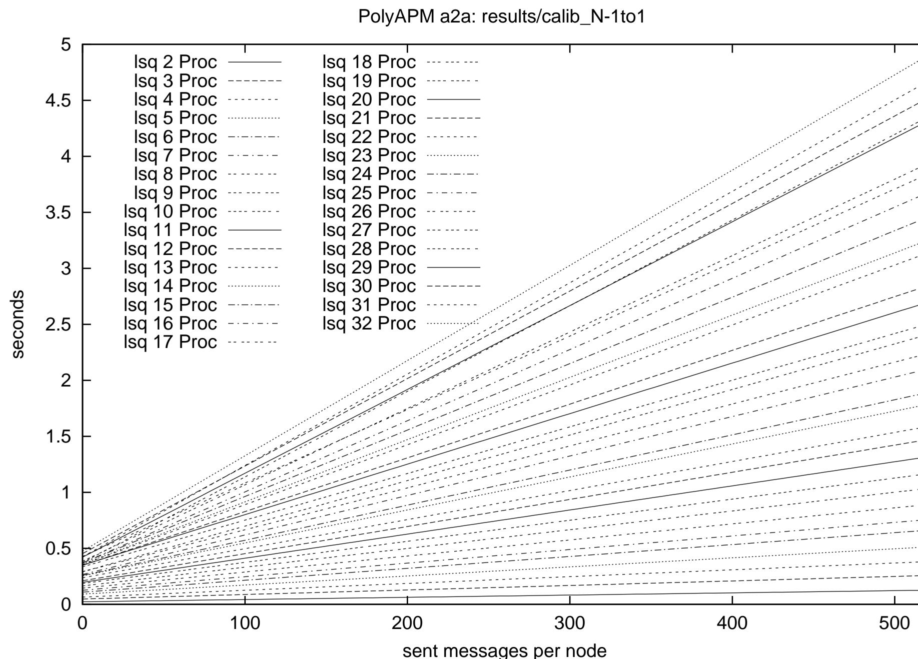


Figure 4.15: Receive Costs over Message Lengths for 2 to 32 Procs

From here on we will only deal with the linear functions that approximate the benchmark results. Benchmarks for receive cost (as depicted in Figure 4.14) are performed for sets of processors ranging from 2 to 32 on the Scali cluster. The resulting linear times, normalised to variable cost by subtracting the constant cost, are displayed in Figure 4.15. While there are too many lines to identify a particular one, one can observe the regularity in the relation of the lines. The lowest line is for the 2-processor benchmark, and upwards the lines for increasing numbers of processors are drawn.

This regularity suggests that the variable cost for a given message length is proportional to the number of processors. The ratio of the slopes of the lines and the number of processors appears to be constant. A simple analysis of the proportions within the data set leads to the cost for receive on one processor, the receive cost.

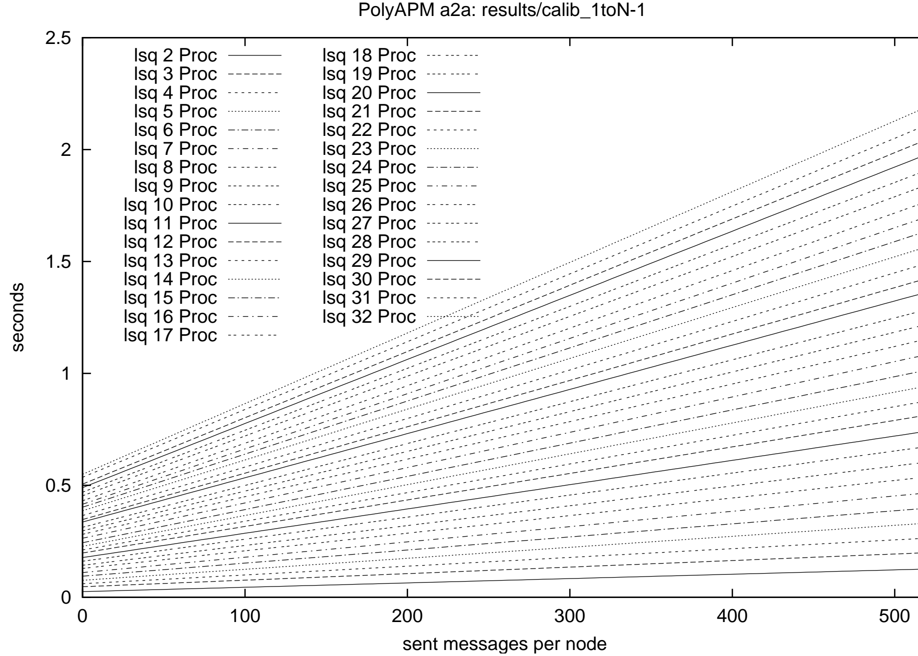


Figure 4.16: Send Costs over Message Lengths for 2 to 32 Procs

All the evaluation detail that has been mentioned in the above discussion for the receive cost also applies to the send costs as depicted in Figure 4.16. Before we analyse the results in detail, a few things can be observed by just comparing Figure 4.15 to Figure 4.16. The dynamic startup costs appear to be quite similar (around 0.5 seconds), but the slope of the receiving cost lines is about twice as high. It appears that the receiving cost per data item is about twice as high as the sending cost.

A more accurate comparison has been performed by analysing the slopes and axis intercepts. Of particular importance are the slope and intercept differences between i and $i+1$ processors. The average of the slope differences denotes the average cost of sending or receiving data to or from one additional processor. Consider the sending costs: we have sequences of slopes $a_0, a_1, \ldots, a_n$ and intercepts $b_0, b_1, \ldots, b_n$ that pairwise form linear equations $l_i := y = a_i * x + b_i$. The average slope $\overline{a}$ and intercept $\overline{b}$ are similar to a_0 and b_0. This leads to the approximation $l_i^{\text{approx}} := y = (i-1) * \overline{a} * x + (i-1) * \overline{b}$. If the errors $a_i - (i-1) * \overline{a}$ and $b_i - (i-1) * \overline{b}$ are small, the hope is that for a given i and sensible values of x the total error of $l_i^{\text{approx}}(x) - l_i(x)$ is also small.

For 1000 communications our experiments determined the following costs:

	send costs		receive costs	
	$\overline{a}_s$	$\overline{b}_s$	$\overline{a}_r$	$\overline{b}_r$
seconds	$9.875275 * 10^{-05}$	0.0174725	$2.773241 * 10^{-4}$	0.0144195

Figure 4.17: Calibration Results for 1000 Communications

The two average constants, $\overline{b}_s$ and $\overline{b}_r$, are expected to be similar: they denote the startup cost for initiating a communication with an additional partner. This cost is defined by the hardware and

communication library software layers that should be similar in both cases. We therefore define the general constant cost per additional communication partner b_{av} as the average of the two: 0.0159460 seconds. As the constant cost of one processor communicating with itself is negligible, and b_{av} is defined for 1000 communications, we define the constant part of the variable cost as:

$$\text{constant part of variable cost with } n \text{ processors} = (n-1) * 0.0159460s * 10^{-3} \tag{4.2}$$

4.4.5 Confirming the Cost Prediction

So far the costs for sends and receives have been determined by just one kind of benchmark. A double-check should confirm the predictions of the cost model. To do this we have performed a third benchmark which is a combination of the send and the receive benchmark. The communication structure is depicted in Figure 4.18.

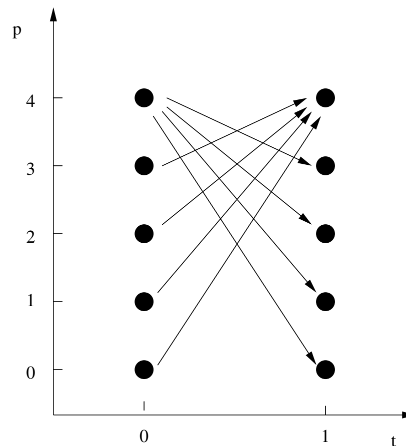


Figure 4.18: Confirming Send and Receive Costs

The idea is to accumulate send and receive costs at one processor which then dominates the benchmark's run time. From the number of sends and receives we will predict the run times of the benchmark and compare the results with the timings. This prediction is not done by way of the PolyAPM cost model, it is just a sanity check for the run times in Figure 4.17.

The prediction is $(p-1) * (\overline{a}_s + \overline{b}_r) * x + b_{\text{av}}$ with x being the number of PolyAPM messages and p the number of processors. The benchmark results are shown in Figure 4.19. For each number of processors we have a linear function that approximates the run time depending on the message length, i.e., the number of PolyAPM messages. On the other hand, the prediction above also yields such a linear function. To compare the two, we consider the functions to be of the form $f(x) = a * x + b$. The comparison is done on the parameters a and b and presented in Figure 4.20.

The a column shows an error margin of almost 60% with lower processors, which later goes down to about 10% at 32 processors. The error in the b column is smaller and also shrinks with an increasing number of processors. The decreasing error was to be expected, as the prediction is based on the averages of a and b values over 32 processors. The fact that the decrease in the b column is not monotonic is not really a problem. The values of b denote the difference in the startup cost of prediction and benchmarks for messages of zero length. There is a considerable error in the benchmark to be expected for such messages. Furthermore, the value of b is only of small impact on the performance prediction in Figure 4.21.

However, more important than just comparing a's and b's is to measure the difference of the function's values. The function $f(x)$ predicts the cost of x messages sent per node. For realistic values of x, the accuracy of this prediction is crucial. Note that the column of b in Figure 4.20

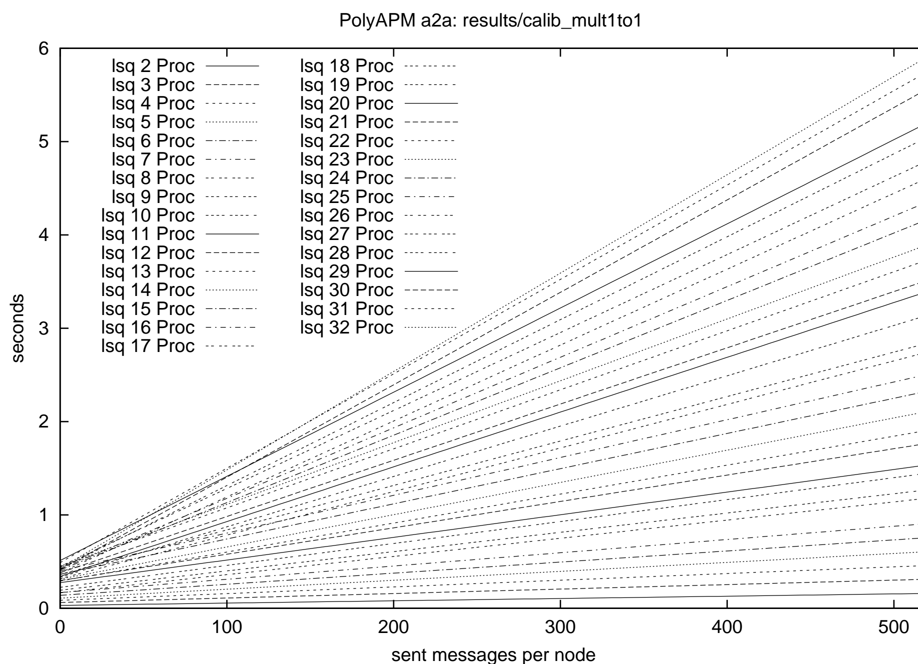


Figure 4.19: Send and Receives on 1 Processor out of 2 to 32 Procs

is $f(0)$. For 100 and 500 PolyAPM messages, we have compared the measured and predicted communication cost for all numbers of processors and determined the difference in percent (see Figure 4.21). The value of b has only limited impact on the values of $f(100)$ and $f(500)$.

In total, the error margin is acceptable. The performance of the entire system depends on the bandwidths of several buses and networks and the sizes of several buffers. We have employed a simplistic linear prediction model and the error margin is well below a factor of 2 in all cases, and usually much better than that. In addition, the cost model prediction is higher than the actual value, so that it is a conservative approximation. The reason is most likely that we scaled the cost for a single message, ignoring the fact that due to pipelining in the hardware the actual cost might be lower than this linear scaling.

The PolyAPM cost model does not strive for accurate run time prediction of the final program but is rather a tool for the relative cost comparison. Highest accuracy is neither feasible nor necessary for our purposes.

4.4.6 Determining the Cost Model Parameters

So far we have determined the computation cost c, the constant communication startup cost (given in Equation 4.1) and the send and receive costs (given in Figure 4.17). All communication values are normalised with respect to c.

The cost model parameters, as mentioned in the PolyAPM cost model in Figure 3.5, have to be defined for the cost model computation in the APM interpreters. They are stored in a record of type *CostModel*, and we can have several of them. One such record is given as a parameter to the cost model computation of the APM interpreters.

```
data CostModel = CostM {mstartup_s::Float, mstartup_r::Float,
                        normsent::Float, normrcvd::Float,
                        barrier::Int->Float, coalesce::Bool}
```

procs	diff a	diff b
2	52.67%	-28.18%
3	56.76%	-39.90%
4	57.36%	-37.56%
5	57.13%	-36.01%
6	58.17%	-38.52%
7	58.89%	-40.18%
8	35.14%	-28.92%
9	45.55%	-30.98%
10	44.02%	-33.40%
11	55.03%	-39.98%
12	46.39%	-39.11%
13	45.42%	-32.47%
14	41.47%	-31.52%
15	39.66%	-37.35%
16	39.35%	-38.98%
17	28.93%	-17.31%
18	34.23%	-23.54%
19	25.51%	-11.62%
20	21.78%	-9.06%
21	24.84%	-14.32%
22	24.88%	-22.97%
23	24.91%	-20.51%
24	19.78%	-8.54%
25	19.35%	-7.14%
26	15.61%	7.39%
27	17.03%	-5.55%
28	14.27%	3.09%
29	16.76%	-11.18%
30	10.15%	14.02%
31	11.57%	-0.49%
32	10.59%	17.82%

Figure 4.20: Differences of Measured and Predicted Communication Cost Functions $f(x) = a * x + b$ in `calib_mult1to1`

procs	100 msgs	500 msgs
2	14.34%	40.32%
3	6.00%	39.25%
4	8.33%	40.65%
5	9.62%	41.08%
6	7.72%	40.85%
7	6.42%	40.68%
8	6.55%	26.25%
9	9.43%	33.95%
10	6.99%	32.02%
11	5.37%	37.96%
12	3.23%	31.91%
13	8.25%	33.39%
14	7.39%	30.59%
15	2.24%	27.42%
16	0.82%	26.64%
17	10.53%	23.53%
18	9.58%	26.75%
19	11.55%	21.52%
20	10.61%	18.64%
21	9.89%	20.54%
22	5.39%	19.10%
23	6.75%	19.57%
24	9.67%	16.95%
25	10.01%	16.75%
26	13.03%	14.92%
27	9.25%	14.88%
28	10.70%	13.31%
29	6.76%	13.96%
30	11.27%	10.44%
31	7.68%	10.52%
32	12.65%	11.12%

Figure 4.21: Differences of Measured and Predicted Communication Costs for Message Lengths of 100 and 500 in `calib_mult1to1`

The barrier cost is the sum of the constant communication startup cost (see Equation 4.1) and the fixed cost associated with each message exchange (see Equation 4.2), divided by c. This cost is imposed once for every communication phase. The constant part of the variable cost is an unusual component of the barrier cost. One would rather expect for b_s and b_r to be the major components of m_s and m_r, resp. However, we found that, in the case of `send_msgs_a2a` with message coalescing, this variable startup cost is independent of the number of sent and received messages. The ruling parameter is the number of communication partners. This is also the case with the constant startup cost itself. Because of this, we have included both in the barrier cost and set the message startup costs m_s and m_r to 0.

The variable send and receive costs are the average slopes per processor of the corresponding benchmark functions, i.e., $\frac{\overline{a}_s}{c}$ $\frac{\overline{a}_r}{c}$.

The above arguments lead to the following set of PolyAPM cost model parameters:

```
CostM {mstartup_s=0.0, mstartup_r=0.0,
       normsent=1.2046961, normrcvd=3.3831080 ,
       barrier= (\n->(fromIntegral (n-1))*298.8954664),
       coalesce = True}
```

These numbers provide us with insight into the performance behaviour of our Scali cluster with the `send_msgs_a2a` function within the `mpi_apm` library. Receiving is twice as expensive as sending, and both factors are unusually small when compared to the cost of one computation, which is by definition one tick. So receiving a floating point number is about 3.4 times as expensive as computing it. In other architectures, we can expect communication to be more expensive. However, the barrier cost for each communication phase is comparatively high. The parameter n within the barrier cost denotes the number of processors participating in the communication.

Chapter 5

Case Study I: Finite Differences

As a first illustrating example, we choose the two-dimensional finite difference method. We start with an abstract problem specification, and by going through a process of program transformations – each yields a new, interpretable specification with equal input/output behaviour – we eventually obtain an executable program for a specific target platform.

First we need to implement the specification in Haskell and identify the parallelism. The APM program expresses the parallelism as a loop nest with one of the two outermost loops being tagged as "parallel". Then we derive subsequent APM programs until a final transformation to the target language is feasible.

The abstract specification of the two-dimensional finite difference problem (as presented by [Fos95]) describes an iterative process of computing new array elements as a combination of the neighbour values and the previous value at the same location. In one such iteration an input array is used to produce an output array of the same size. An application of the finite difference problem typically consists of many such iterations. Many application domains, such as image processing, operate on two-dimensional data.

In order to get a picture of what this algorithm does see Figure 5.1. The left part shows the computation of a new element. Its old value is multiplied by four, then the values of the right, left, upper and lower neighbours are added, and finally the result is divided by eight. This operation can be performed only if all four neighbours exist, which is not the case at the borders of the array. The right part of the picture shows that in one finite difference iteration only the inner elements are being computed, while the border elements are just copied from the old array. No matter in which order the elements of the new iteration are computed, every computation needs old values from the previous iteration that are being overwritten by current ones. Thus two copies of the array are needed: one with the old content that is read-only, and one to hold the new values that is write-only. Subsequent iterations may alternate the role of the two copies.

Formally, the new array a_t at iteration t with $(xh - xl) * (yh - yl)$ elements is defined as:

$$\begin{aligned}
&\Big(\forall x \in \{xl+1 \ldots xh-1\}, \forall y \in \{yl+1 \ldots yh-1\} :: \\
&\quad a_t[x,y] := \frac{a_{t-1}[x-1,y] + 4 * a_{t-1}[x,y] + a_{t-1}[x+1,y] + a_{t-1}[x,y-1] + a_{t-1}[x,y+1]}{8}\Big), \\
&(\forall y \in \{yl \ldots yh\} :: \\
&\quad a_t[xl,y] := a_{t-1}[xl,y], \\
&\quad a_t[xh,y] := a_{t-1}[xh,y]\ ), \\
&(\forall x \in \{xl+1 \ldots xh-1\} :: \\
&\quad a_t[x,yl] := a_{t-1}[x,yl], \\
&\quad a_t[x,yh] := a_{t-1}[x,yh]\ )
\end{aligned} \tag{5.1}$$

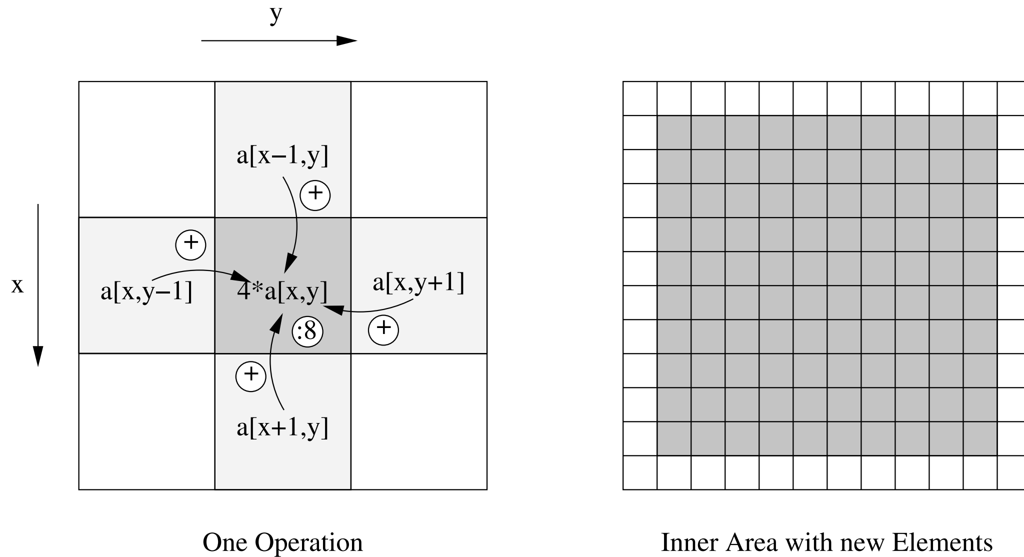


Figure 5.1: 2D Finite Differences

The Haskell specification of this problem is given in Figure 5.2. Note how closely the Haskell program of Figure 5.2 corresponds to Equations 5.1. `findiff2D` represents one iterative step in which a new array is computed from the previous array a, just as the Equations 5.1 require. Function `fd` calls `findiff2D` as often as the parameter n prescribes. It does so by a repeated function composition using `ncombine`. In the current example, the number of iterations is arbitrary, but fixed. We set $n = 20$. Each call to `findiff` generates a new Haskell array with the updated values. This corresponds to a destructive update of an array using an imperative programming model. In this example, all references to a refer to the previous iteration, so that, as explained above, we need two copies of the array.

We continue with the presentation of transformed `findiff` programs, emphasising the difference to the corresponding predecessor programs.

5.1 Initial Code Generation: The Synchronous Program

The Haskell program specification (Figure 5.2) has to be transformed into an imperative loop nest. We call this process the *initial code generation* as it is the first of a sequence of program transformations. In parallel programming the term *code generation* is usually reserved for the construction of the final program version, to be executed on a parallel machine.

The parallelisation of the finite difference source program is done manually. It is obvious that the calculations of the array elements inside one particular `findiff2D` call are independent, but they all depend on the previous values. Thus, `fd` corresponds to an outer, sequential loop with 20 iterations, whereas the list comprehensions inside `findiff2D` yield a parallel loop.

To write a SynAPM program for `findiff`, we proceed as follows:

1. We define the memory contents, here: two arrays `a` and `a1` of the same kind as the input array.
2. We set the read-only structure parameter list to $[n]$, where n describes the size of `a`.
3. We define the loops: one outer sequential loop, arbitrarily set to 20 iterations, and one inner parallel loop, ranging from 0 to $n^2 - 1$.

```
findiff2D:: LArray (Int,Int) Float -> LArray (Int,Int) Float
findiff2D a = array ((xl,yl),(xh,yh))
              ([((xl,y), a!(xl,y)) | y <- [yl..yh]] ++
               [((xh,y), a!(xh,y)) | y <- [yl..yh]] ++
               [((x,yl), a!(x,yl)) | x <- [xl+1..xh-1]] ++
               [((x,yh), a!(x,yh)) | x <- [xl+1..xh-1]] ++
               [((x,y), (a!(x-1,y)+a!(x+1,y)+4*a!(x,y)+a!(x,y-1)+a!(x,y+1))/8 )
                | x<-[xl+1..xh-1], y<-[yl+1..yh-1]])
    where ((xl,yl),(xh,yh)) = bounds a

ncombine:: Int -> (a->a) -> (a->a)
ncombine 0 f = f
ncombine n f = f . (ncombine (n-1) f)

fd :: (LArray (Int,Int) Float) -> (LArray (Int,Int) Float)
fd a  = ncombine 20 findiff2D a
```

Figure 5.2: Sequential Haskell Specification of Finite Differences

4. We write a loop body function.
5. We define an instance of the `Synchronizing` class.

This defines the synchronous loop nest `loop_s` in Figure 5.3. The body function of a SynAPM program has the following type: `BD (e -> b -> e)`, i.e., it takes some *state*, consisting of memory and structure parameters, and a list of current values of all surrounding loops, to return an updated state. Figure 5.3 shows the state to be of type `(Mem,[Idx],CompStat)`. The memory type `Mem` comprises the two arrays. The first array `a` is the one computed by the previous iteration, and `a1` is computed by the current iteration. Note that the structure parameter list `splist` consists of only one item: n, the size of the arrays. As the bordering array elements are just copied and not computed, a case analysis is needed.

The x and y coordinates addressing the array elements are defined in terms of the parallel loop variable p. The two separate generators of x and y from the specification are replaced by one p loop.

The computation statistics are part of the PolyAPM cost model. The basic cost unit is a floating point operation. The cost type is a one-dimensional array of processor numbers and their costs. In each loop iteration the current computation cost is added to the pth element of the cost array `ca`.

The programmer determines the sum of all operations' costs that are executed within the loop body. In the `body_s` function in Figure 5.3 we determine the cost of the `if` guard and add the computation costs of the branches. The guard contains four comparisons and three disjunctions, so that we estimate the cost with seven units. The first branch does no computation at all, so the total cost is just seven units. The second branch executes `stmnt1` that we gauge with additional six units, thus yielding a total of 11 units. These two cost values are added to the cost array as part of the return state.

Between the time steps we need to reorganise the memory. With each time step, we need a new array that holds the new values. It is placed in the right component of the memory pair `(a,a1)`. The reorganisation shifts the arrays: the old `a` array is disregarded, the old `a1` array becomes the new `a` array, and an empty array is created and named `a1`. This task has to be performed after all loop iterations of the time step are completed. The function `synchronizeMem` is the right place for this. Subsequent `synchronizeMem` instances will be identical until the memory type changes in `SynDMAPM`.

```
type Mem = (LArray (Int,Int) Float,LArray (Int,Int) Float)

if_cost  = 7  --  cost for the if-guards inside the loop body
elt_cost = 6  --  cost to compute one new element in the body

loop_s = LP [(Seq, \([],(n:_))->0,
                   \([],(n:_))->21,
                   \([],(n:_))->1),
                (Par, \((t:_),(n:_))->0,
                      \((t:_),(n:_))->n*n,
                      \((t:_),(n:_))->1)]
                (BD body_s)

body_s::(Mem,[Idx],CompStat) -> [Idx] -> (Mem,[Idx],CompStat)
body_s ((a,a1),splist,ca) (t:p:_) =
         if (x == xl || x == xh || y == yl || y == yh)
         then ((a, a1//[((x,y),a!(x,y))]),
               splist, ca//[(p, if_cost+ca!p)])
         else ((a, stmnt1 a a1 (t,x,y,n)),
               splist, ca//[(p, if_cost+elt_cost+ca!p)])
    where stmnt1 a a1 (t,x,y,n) =
              a1//[((x,y),(a!(x-1,y)+a!(x+1,y)
                           +4*a!(x,y)+a!(x,y-1)+a!(x,y+1))/8)]
          x                 = p `mod` n
          y                 = p `div` n
          (low,up)          = bounds a
          ((xl,yl),(xh,yh)) = (low,up)
          (n:_)             = splist

instance Synchronizing Findiff2D_SynAPM Mem where
    synchronizeMem Findiff2D_SynAPM (a,a1) =
        (a1,listArray (bounds a1) [ error ("fd "++(show i))|i<-(indices a)])
```

Figure 5.3: SynAPM Version of the Finite Differences

If we interpret the APM program with four processors and a problem size of $n = 256$, we obtain the following results in our cost model:

```
Cost model, total (no comms): 4.113507e8, agg. work: 1.7762808e7
```

These results tell us that the cost of the parallel execution of this program is by an order of magnitude greater than the aggregated cost of all computations. Since the latter can be viewed as an estimate of the sequential run time, we obtain a speedup of less than 1. This bad performance is typical for initial SynAPM programs. The reason is that after each time step a barrier synchronisation takes place and the cost model profile for the Scali cluster contains a barrier cost function that is linear in the number of participating processors. The initial APM program uses the maximum number of (virtual) processors as given by the problem size, with each processor computing only one element per time step. Therefore, the barrier cost per processor is much higher than the small computation cost, yielding the bad speedup.

```
data Findiff2D_til_SynAPM = Findiff2D_til_SynAPM

tilesize2D n maxp = min (n `div` (intSqrt maxp)) n

loop_til = LP loop_bounds_til (BD body_til)

loop_bounds_til = [(Seq, \([],(n:maxp:_))->0,
                         \([],(n:maxp:_))->21,
                         \([],(n:maxp:_))->1),
                    (Par, \((t:_),(n:maxp:_))->0,
                          \((t:_),(n:maxp:_))->maxp,
                          \((t:_),(n:maxp:_))->1),
                     (Seq, \((t:rp:_),(n:maxp:_))->
                             max ((rp `mod` intSqrt maxp) * tilesize2D n maxp) 0,
                           \((t:rp:_),(n:maxp:_))->
                             min ((rp `mod` intSqrt maxp) *
                                   tilesize2D n maxp + (tilesize2D n maxp)) n,
                           \((t:rp:_),(n:maxp:_))-> 1),
                      (Seq, \((t:rp:x:_),(n:maxp:_))->
                             max ((rp `div` intSqrt maxp) * tilesize2D n maxp) 0,
                            \((t:rp:x:_),(n:maxp:_))->
                              min ((rp `div` intSqrt maxp) *
                                    tilesize2D n maxp + (tilesize2D n maxp)) n,
                            \((t:rp:x:_),(n:maxp:_))-> 1)]

body_til::(Mem,[Idx],CompStat) -> [Idx]
        -> (Mem,[Idx],CompStat)
body_til ((a,a1),splist@(n:_),ca) [t,rp,x,y] =
      if (x == xl || x == xh || y == yl || y == yh)
      then ((a,a1//[((x,y),a!(x,y))]),splist,ca//[(rp,if_cost+ca!rp)])
      else ((a, stmnt1),splist,ca//[(rp,if_cost+elt_cost+ca!rp)])
    where stmnt1 =
              a1 //[((x,y), (a!(x-1,y)+a!(x+1,y)+4*a!(x,y)+a!(x,y-1)+a!(x,y+1))/8)]
          (low,up)         = bounds a
          ((xl,yl),(xh,yh)) = (low,up)

instance Synchronizing Findiff2D_til_SynAPM Mem where
    synchronizeMem Findiff2D_til_SynAPM = synchronizeMem Findiff2D_SynAPM
```

Figure 5.4: Tiled SynAPM Version of the Finite Differences

5.2 The Tiled Program

The first transformation within the APM framework according to the PDG (Figure 3.1) is the *processor tiling* to reduce the number of parallel processors with the following effects: matching an existing machine with limited resources, improving locality and increasing the computational work load per processor in order to increase efficiency.

Figure 5.4 shows the synchronous `findiff` program after a processor tiling transformation. The parallel loop of `loop_s` has been partitioned into tiles such that the number of remaining parallel iterations matches the number of physically available processors (as defined by `physprocs`). The old parallel loop ranges over all n^2 elements of the result array, so that the parallelism is reduced from n^2 to `maxp`. Each tile comprises a number of independent loop iterations that are now sequentially executed on the same physical processor. Therefore, one or more loops are necessary to enumerate these sequential iterations. In our case, we choose a two-dimensional tiling, so that two inner sequential loops are added.

A two-dimensional tiling with square tiles will exhibit the smallest number of communications later on. This is because the computations of the finite difference problem need only neighbour values, so that the number of communications needed for all iterations within a tile is proportional to the length of the circumference of the tile. The index space is rectangular, so we use rectangular tiles to partition the index space. The rectangles with the smallest circumference are squares, which is why we use square tiles to minimise the communications. This is called the *surface-to-volume effect* (see [Fos95], page 40). Costing an alternative APM program would easily reveal that a different rectangular tiling causes a higher number of communications.

We choose to enumerate the iterations within a tile with two sequential loops, one for each dimension. If we used just one loop, ranging over p, we could re-use the `loop_s` body. But in that case the loop bounds for the p loop are quite complicated. We pay for simplicity in the loop bounds with a slight modification of the body function.

The two inner sequential loops enumerate the absolute x and y coordinates of each computation within a tile. While `body_s` has to determine the values for x and y from the value of p, `body_til` gets them directly as loop variables. Except for the adaptation of these variables, the body function does not change. As the APMs work with an arbitrary but fixed number of processors, the tiled program can still run on SynAPM.

An execution of the tiled program yields a dramatic decrease of the parallel cost (again four processors and input size of 256):

```
Cost model, total (no comms): 4459533.0, agg. work: 1.7762808e7
```

The main reason for this decrease is that, instead of previously 256 processors, now only four processors have to synchronise. We show in Section 4.4.3 that the barrier cost is a linear function of the number of involved processors. Since no communication is involved, the speedup improved from significantly less than one to 3.98. This is expected as the load should be distributed equally over the four processors.

Code Changes to Obtain the *tiled* Program

- The `tilesize2D` function to determine the length of the square tiles is added.
- The former p loop is replaced by the rp loop that enumerates real processors. Two additional loops for x and y are added to enumerate operations within a tile.
- The body function takes four loop variables, t, rp, x and y. The computation of x and y has been removed.

5.3 The Communicating Program

Loop and body functions are the same as in the tiled program except for the memory type. Whereas in the previous APM programs the memory type could be defined freely, we now require the parameterised type `GlobalStateShM` that couples memory and message queue. Functions that alter the memory now have to access it as part of the global state. The message queue is not manipulated directly. The `SynCommAPM` program is presented in Figure 5.5. Emphasising font is used for pseudo code that replaces some longer Haskell code. Function `synchronizeMem` has been omitted as it is identical to the SynAPM version.

```
gen_cost = 8  --  cost for the message generation right after the body

body_sc = similar to body_til
loop_sc = similar to loop_til, msg generation after each body

instance Sendable Findiff2D_SynCommAPM SC_Dom (Int,Int) Float Mem where
  generateMsg Findiff2D_SynCommAPM [t,rp,x,y] (n:maxp:_) (a,a1) =
             ([Msg (rp, to_p, t, to_tm, A, (x,y), a1!(x,y), 1.0)|
                to_p <- (
                         (if is_left_border  then [rp-1] else [])++
                         (if is_right_border then [rp+1] else [])++
                         (if is_upper_border then [rp-(intSqrt maxp)] else [])++
                         (if is_lower_border then [rp+(intSqrt maxp)] else [])
                        ),
                to_tm <- [t+1]
              ],(rp,gen_cost))
      where ts             = tilesize2D n maxp
            is_left_border  = xpos == 0      && is_inner_element
            is_right_border = xpos == (ts-1) && is_inner_element
            is_upper_border = ypos == 0      && is_inner_element
            is_lower_border = ypos == (ts-1) && is_inner_element
            is_inner_element= ((x > 0) && (x < n-1) && (y > 0) && (y < n-1))
            xpos           = x `mod` ts
            ypos           = y `mod` ts

instance Updatable Findiff2D_SynCommAPM SC_Dom (Int,Int) Float Mem where
  updateMem Findiff2D_SynCommAPM
    (Msg (from_p, to_p, from_tm, to_tm, dom, idx, val, cost)) (a,a1) =
         if (a1!idx) == val
         then (a,a1 //[(idx,val)])
         else error "Wrong update of cell"
```

Figure 5.5: SynCommAPM version of the Finite Differences

New are two additional functions that have to be implemented by the programmer and which are called from inside the interpreter:

- `generateMsg` generates new messages originating from each processor at each time step;
- `updateMem` updates the state's memory with values sent in a message.

A remark for the Haskell expert: these two functions have to be introduced by class instance declarations because the APM interpreter needs some type information to use the – from the interpreter's point of view – undefined functions as stubs. This is because the APM interpreters

reside in a separate Haskell module that is being used by different APM program modules. So, for every APM program the specific instances of these three functions are different, yet they need to fit into the APM, and making them instances of multi-parameter type classes guarantees the integration into the APM interpreter.

The function `generateMsg` first checks whether the element just computed is at the border of a tile, but not at the border of the index space. If those conditions are satisfied, then messages to the neighbouring tiles are generated. Together with the list of generated messages this function returns the cost for the generation, paired with the processor id.

The function `updateMem` checks before an update whether the memory's and the message's values are identical. If they differ, the interpreter issues a run time error message because a wrong communication message was generated. This is by no means a method to prove correctness of communications, but testing the SynCommAPM program with a variety of inputs without errors can provide some confidence in the message generation, which belongs to the more error-prone parts of parallel programming. The SynDMAPM provides further communication checks.

To evaluate the transformation we execute the transformed SynCommAPM program again on four processors with an input size of 256:

```
Cost model, total: 7236516.0
 work totals: [7193214.0,7193214.0,7193214.0,7193214.0]
 send totals: [42,42,42,42] = 168
 recv totals: [42,42,42,42] = 168
 comm totals: [43301.758,43301.758,43301.758,43301.758]
```

Several things are worth noticing: the total cost has gone up again, and when compared to the aggregated (i.e., sequential) work of the SynAPM programs, it exhibits an expected speedup of only 2.45. The additional data provides insight into the reason for this loss of efficiency. Each list of totals contains four elements, one for each processor. The values are the sums of the respective cost for each processor over the entire program execution. As finite difference is a completely symmetric problem, the cost is identical for all processors.

The computational work (work totals) is the major component of the total cost and by two orders of magnitude greater than the communication cost (comm totals). The latter is small since we do not have many communications going on – just 42 sends and receives per processor. However, note that the employed cost model profile is for coalesced messages (as implemented by the `mpi_apm` library), so that the number of PolyAPM messages is much higher than 42.

The increase in the total cost is mainly due to the increase of computation cost. However, the loop bodies do the same amount of work. The additional cost stems from the effort for message generation, which is in this example relatively high compared to the body computations.

Note that, in this completely symmetrical problem, the total cost is just the sum of computation and communication costs. For asymmetric problems, this is usually not the case. The total cost is the sum of the maximal processors costs of each time step. If – over time – different processors assume the maximum, then the total cost may be higher than each individual processor's sum.

Code Changes to Obtain the *Communicating* Program

- A `GlobalStateShM` type combining memory and message queue replaces the memory; types in body and `LP` are adapted accordingly.
- Each call of the body is followed by a call of the message generation function of SynCommAPM, which in turn calls the provided `generateMsg` (see Section 4.2.3).
- Instance declarations for `generateMsg` and `updateMem` are added.

```
type DM_local_state = LocalState Mem SC_Dom (Int,Int) Float

data Findiff2D_SynDMAPM = Findiff2D_SynDMAPM

body_dm::(DM_local_state ,[Idx]) -> [Idx] ->  ((DM_local_state,[Idx]),[Idx])
body_dm (LState mem msgs (CCL statl), splist) (idxlist@[t,rp,x,y]) =
    if (x == xl || x == xh || y == yl || y == yh)
    then ((LState (a,a1//[((x,y),a!(x,y))]) msgs
           (CCL ((if_cost):statl)),splist), idxlist)
    else ((LState (a,a1//[((x,y),stmnt1)]) msgs
           (CCL ((if_cost+elt_cost):statl)),splist), idxlist)
    where stmnt1 = (a!(x-1,y)+a!(x+1,y)+4*a!(x,y)+a!(x,y-1)+a!(x,y+1))/8
          (a,a1)            = mem
          ((xl,yl),(xh,yh)) = bounds a
          (n:_)             = splist
```

The functions `loop_dm`*,* `generateMsg`*,* `updateMem` *and* `synchronizeMem` *are similar to before and have only been adjusted to the new memory data type.*

Figure 5.6: SynDMAPM version of the Finite Differences

5.4 The Distributed Memory Program

With the paradigm shift from shared to distributed memory, the memory representation in the APM programs has to be adapted. Each processor gets its own chunk of the memory, which in this case is defined by the type `DM_local_state` in the program in Figure 5.6. The distributed memory comprises all the data which is computed on this processor and a copy of the remotely owned data that is required for the computation. The values of the latter are communicated before the computation.

The two-dimensional array is divided into square sub-arrays according to the tile size, with an additional rim of width 1 at the edges to accommodate copies of neighbour elements needed for the computation. With a tile size of ts, the arrays in the local memory of each processor contain $(ts + 2)^2$ elements. The tiles overlap because of the rims.

Since Haskell (and therefore PolyAPM) programs allow for array indices of any ordered type, the new local memory arrays that contain the local tiles can use the identical indices of the corresponding SynAPM global memory sub-array. Unlike, e.g., in C, Haskell arrays do not need to be indexed from 0 to $size - 1$. This leads to only few adaptations in the `body_sc` function to result in Figure 5.6.

The previous APM programs could directly change the cost information in the globally available `[CostItem]` list. The loop bodies of a `SynDMAPM` program have only access to the local state. In Section 4.2.4 we explain how the body function stores its computation cost in a local computation cost list `CCL` and the how interpreter collects its content at the end of each time step to store it in the global cost structure.

Also, the instance declaration functions `generateMsg`, `updateMem` and `synchronizeMem` have to be adapted to the new memory type, but all of these changes are straightforward.

As the new memory type affects neither the computation cost nor the communication overhead, the total PolyAPM cost remains the same:

```
Cost model, total: 7236516.0  (9989028.0)
...
```

Code Changes to Obtain the *Distributed Memory* Program

- The memory type within the global state changes to an array of local state types, one state for each processor. Body function, `generateMsg`, `updateMem` and `synchronizeMem` are adapted accordingly.
- In the `LP` structure, just the names of the body/`generateMsg` functions change.
- Any non-APM functions have to be adapted which create the initial state to be fed into the PolyAPM program and also functions that access the state after the APM interpreter finishes.

5.5 The Distributed Memory Program with Relative Coordinates

So far all APM programs have used absolute coordinates. However, target languages like C require their arrays of size n to be indexed from 0 to $n-1$. One aim of PolyAPM is to make each transformation simple, with special emphasis on the last transformation to the target program. Since a change in the way the memory is indexed affects most APM program parts, we perform another PolyAPM transformation. The result is again a `SynDMAPM` program, but with relative indexing, so that the subsequent transformation to C is simpler.

In the code in Figure 5.7 the sub-arrays that comprise the memory of a specific tile are indexed with relative coordinates. As the loop bounds remain unchanged, the body function converts the absolute index values x and y to their respective relative counterparts $xRel$ and $yRel$ that start from 0. All memory accesses are adapted accordingly.

The index change also affects message generation. Each message contains the array index of the transmitted value. The previous APM programs use absolute coordinates in which each index is globally unique. This is not true for the relative coordinates, in which the local tile memory of each processor is indexed from (0,0) to $(ts-1, ts-1)$. The coordinates of the sender's elements have to be converted to the coordinate system of the receiving end. This is done by the function `dst_relCoords` that computes the relative coordinates of the destination processor.

The functions `updateMem` and `synchronizeMem` as well as the cost model results are not affected by the index change so that they are not repeated here.

This program resembles very much an imperative SPMD program with loops as the control structure so that the transition to C+MPI can be performed easily.

Code Changes to Obtain the *Distributed Memory* Program with Relative Coordinates

- Replace all array indices x and y by $xRel$ and $yRel$.
- Adapt `generateMsg` with a `dst_relCoords` to send the relative coordinates as indices in a message.

5.6 The C+MPI Program

This last transformation leaves the APM realm. Conceptually, nothing interesting happens, but a language barrier has to be crossed. The simpler the body function is, the easier its transformation into a C function gets. The premier area of parallel programming, scientific computation, usually deals with arithmetic operations on arrays. The array as the most frequently used data structure

```
body_rel_dm::(DM_local_state ,[Idx]) -> [Idx] ->  ((DM_local_state,[Idx]),[Idx])
body_rel_dm (LState mem msgs (CCL statl), splist@(n:maxp:_))
            (idxlist@[t,rp,xAbs,yAbs]) =
    if (xAbs == 0 || xAbs == n-1 || yAbs == 0 || yAbs == n-1)
    then ((LState (a, a1//[((xRel,yRel),a!(xRel,yRel))]) msgs
           (CCL ((if_cost):statl)),splist), idxlist)
    else ((LState (a,a1//[((xRel,yRel),stmnt1)]) msgs
           (CCL ((if_cost+elt_cost):statl)),splist),idxlist)
    where stmnt1      = (a!(xRel-1,yRel)+ a!(xRel+1,yRel)+ 4*a!(xRel,yRel)+
                         a!(xRel,yRel-1)+ a!(xRel,yRel+1)                  )/8
          (xRel,yRel) = abs2rel (xAbs,yAbs) rp n maxp
          (a,a1)      = mem


instance Sendable Findiff2D_rel_SynDMAPM SC_Dom (Int,Int) Float Mem where
  generateMsg Findiff2D_rel_SynDMAPM [t,rp,xAbs,yAbs] splist@(n:maxp:_) (a,a1) =
     ([Msg (rp, to_p, t, to_tm, A, dest_coords, a1!(xRel,yRel), 1.0)
       | (to_p, dest_coords) <-
         ((if is_left_border  then [(rp-1,      dst_relCoords (rp-1))] else [])++
          (if is_right_border then [(rp+1,      dst_relCoords (rp+1))] else [])++
          (if is_upper_border then [(rp+sqMaxp, dst_relCoords (rp+sqMaxp))] else [])++
          (if is_lower_border then [(rp-sqMaxp, dst_relCoords (rp-sqMaxp))] else [])
         ),
         to_tm <- [t+1]
      ], (rp, gen_cost))
      where sqMaxp          = intSqrt maxp
            dst_relCoords p = abs2rel (xAbs, yAbs) p n maxp
            (xRel, yRel)    = dst_relCoords rp
            ts              = tilesize2D n maxp
            is_left_border  = xRel == 1  && is_abs_inner_element
            is_right_border = xRel == ts && is_abs_inner_element
            is_upper_border = yRel == ts && is_abs_inner_element
            is_lower_border = yRel == 1  && is_abs_inner_element
            is_abs_inner_element
                            = ((xAbs > 0) && (xAbs < n-1) &&
                               (yAbs > 0) && (yAbs < n-1))
```

Figure 5.7: SynDMAPM version of the Finite Differences with Relative Indexing

exists in both languages. This is not to say that more general problem domains cannot be handled, but then the target code transformation can get more complicated.

To generate target code, abstract APM communications have to be transformed into MPI calls by using the `mpi_apm` library. The memory data type and its distribution/aggregation function need imperative equivalents. But all these changes are isolated, and in most cases not difficult, especially if this transformation was taken into account while choosing the appropriate Haskell types.

Procedure to Obtain the *C+MPI* Program

- The template in Figure 4.8 for an SPMD program is used. The body, message generation and memory update functions have to be filled in, i.e., the functions `body_rel_dm`, `generateMsg`, `updateMem` and `synchronizeMem` are rewritten in C.
- The memory type has to be adapted, the MPI message handling is performed by the `mpi_apm` library.
- Any functions creating and reading the finite difference arrays have to be reimplemented in C.

```
void apm_main_loop(int n){
  int t,x, y, xAbs,yAbs, status, i;
  const int rp = rank;
  const int lbX = MAX((rp % sqrtMaxp) * ts,      0);
  const int ubX = MIN((rp % sqrtMaxp) * ts + ts, n);
  const int lbY = MAX((rp / sqrtMaxp) * ts,      0);
  const int ubY = MIN((rp / sqrtMaxp) * ts + ts, n);

  for (t = 0; t < NUM_RUNS; t++) {
    clear_senddata();

    for(xAbs = lbX; xAbs < ubX; xAbs++) {
      for(yAbs = lbY; yAbs < ubY; yAbs++) {
        body2d(t,rp,xAbs,yAbs,n);
        genmsgs(xAbs, yAbs, n);
      }
    }
    send_msgs(n);
    synchronizeMem();
  }
}
```

Figure 5.8: Finite Differences in C+MPI: Function `apm_main_loop`

We will now present some selected C functions of the target program to illustrate the transformation. Figure 5.9 shows the complete set of correspondences between the APM and the C program.

The filled-in template of Figure 4.8 is the function `apm_main_loop` as depicted in Figure 5.8. Initially, the extent of the tile is determined and used for the inner loop bounds. At the beginning of each time step the send buffers are cleared. The two loops start the enumeration of all absolute indices of a tile and call `body2d` for the computation and `genmsgs` for message generation. After all computations of a time step are completed, the messages are sent and the arrays are swapped.

APM program parts	C+MPI program parts
general loop structure, template in Figure 4.8	apm_main_loop (Fig. 5.8)
body_s_relDM (Fig. 5.7)	body2d (Fig. 5.10)
generateMsg (Fig. 5.7)	genmsgs (Fig. 5.11)
synchronizeMem (SynAPM version Fig. 5.3)	synchronizeMem (Fig. 5.12)
updateMem (SynCommAPM version Fig. 5.5)	updateMem_with_msg
APM interpreter message exchange	send_msgs (Fig. 5.12)

Figure 5.9: Correspondences of APM and C Programs

```
void body2d(int t, int rp, int xAbs, int yAbs, int n){
  int relX, relY;
  abs2rel(xAbs,yAbs, &relX, &relY, rank, n);
  if (xAbs == 0 || xAbs == n-1 || yAbs == 0 || yAbs == n-1) {
    fd_a1[relX][relY] = fd_a[relX][relY];
  } else {
    fd_a1[relX][relY] =  (fd_a[relX-1][relY] + fd_a[relX+1][relY] +
                          4*fd_a[relX][relY] + fd_a[relX][relY-1] +
                          fd_a[relX][relY+1])/8;
  }
}
```

Figure 5.10: Finite Differences in C+MPI: Function body2d

Figure 5.10 displays the C implementation of the body function. The loop variables are passed on as parameters, while the arrays, represented by fd_a and fd_a1 are globally defined. The structure of the function follows the corresponding APM function (compare with function body_relDM in Figure 5.7).

The message generation of the C program in Figure 5.11 corresponds to the PolyAPM function generateMsg in Figure 5.7. The Haskell list comprehension does not have a direct correspondence in C. Messages are created and placed in the send buffer by the function genMsgTo. This is the equivalent of creating one message as a list element in the APM function. Message creation is guarded by several predicates. The C function has a nested if structure to accommodate the cases defined by the conjunction of these predicates, while the Haskell function contains a combination of guarded list generators. But the underlying principle is identical: boolean guards steering the creation of messages. We argue that this principle can be observed in the APM program and translated without much effort to an equivalent implementation in the target language.

The communication phase at the end of each time step is performed by the C function send_msgs as displayed in Figure 5.12. This function sets the number of messages to send for all current communication partners and consequently calls the mpi_apm function send_msgs_a2a (see Section 4.3.1).

The APM function synchronizeMem shifts the arrays by creating a new one to hold the new elements. The advantage is that a new array is devoid of any defined elements so that program errors can be detected easily. However, this frequent creation is not very efficient. While the emphasis with APM programs is on correctness, the programs in the target language strive for efficiency while maintaining correctness. Therefore, we avoid creating a new array in synchronizeMem by reusing the old one. The arrays simply swap their purpose. We do this in the C function synchronizeMem in Figure 5.12 by swapping the array pointers of a and a1.

```
void genMsgTo(int to_p, int x, int y, double sendval) {
  const int offset = sendnr_buf[to_p];
  psendmsg_buf[to_p*ts+offset].xidx = x;
  psendmsg_buf[to_p*ts+offset].yidx = y;
  psendmsg_buf[to_p*ts+offset].value = sendval;
  sendnr_buf[to_p] += 1;
}

void genmsgs(int xAbs, int yAbs, int n) {
  int xRel, yRel, isRelBorder;
  double sendval;
  const int is_abs_inner_element =
    ((xAbs > 0) && (xAbs < n-1) && (yAbs > 0) && (yAbs < n-1));
  abs2rel(xAbs, yAbs, &xRel, &yRel, rank, n);
  isRelBorder = ((xRel == 1) || (xRel == ts) ||
                 (yRel == 1) || (yRel == ts));

  /* if we are at the local border of a tile... */
  if(isRelBorder) {
    /* and we are not at the border of the global array */
    if(is_abs_inner_element) {
      sendval = fd_a1[xRel][yRel];

      if (xRel == 1) {  /* send left */
        genMsgTo(rank-1, ts+1, yRel, sendval);
      }
      if (xRel == ts) { /* send right */
        genMsgTo(rank+1, 0, yRel, sendval);
      }
      if (yRel == ts) { /* send up */
        genMsgTo(rank+sqrtMaxp, xRel, 0, sendval);
      }
      if (yRel == 1) {  /* send down */
        genMsgTo(rank-sqrtMaxp, xRel, ts+1, sendval);
      }
    }
  }
}
```

Figure 5.11: Finite Differences in C+MPI: Message Generation

```
void send_msgs(int n){
  int i;
  for(i=0; i<size; i++) {
    if(sendnr_buf[i] > 0) sendnr_buf[i] = sendmsg_cntr;
  }

  send_msgs_a2a(n, size,
                sendmsg_cntr, sendnr_buf,
                displ, sendbuf_size, rcvnr_buf,
                psendmsg_buf, prcvmsg_buf,
                MYCOMM);
}

void synchronizeMem(){
  double** tmpptr;
  tmpptr = fd_a;
  fd_a = fd_a1;
  fd_a1 = tmpptr;
}
```

Figure 5.12: Finite Differences in C+MPI: functions `send_msgs` and `synchronizeMem`

5.7 Benchmarking the C+MPI Program

		1 processor		4 processors			16 processors		
size	seq	time	abs su	time	rel su	abs su	time	rel su	abs su
64	0.003	0.127	0.02	0.135	0.94	0.02	0.792	0.16	0.00
128	0.018	0.178	0.10	0.145	1.22	0.13	0.730	0.24	0.02
256	0.145	0.413	0.35	0.202	2.04	0.72	0.710	0.58	0.20
512	0.582	1.327	0.44	0.439	3.02	1.33	0.841	1.58	0.69
1024	2.407	4.974	0.48	1.356	3.67	1.77	1.012	4.92	2.38
2048	9.690	19.431	0.50	5.008	3.88	1.93	2.040	9.52	4.75
4096	40.091	77.383	0.52	19.623	3.94	2.04	5.525	14.01	7.26

Figure 5.13: Benchmarking 2D Finite Differences in C+MPI

We have benchmarked the C+MPI version of the 2D finite difference problem on a 64-processor Scali Linux cluster (see Figure 5.13). The problem sizes ranging from 64 to 4096 represent the number of elements of each dimension of the 2D input array. We have also implemented a sequential version of the same algorithm. Its run times on the same machine are displayed in column *seq*. All times are measured in seconds and given for 20 repetitions of the loop nest to get measurable times for small problem sizes. We can observe increasing speedups with increasing problem sizes. The parallel program scales well as the excellent relative speedups show (columns *rel su*. With large problem sizes the absolute speedups become satisfactory and peak at an efficiency around 50% (columns *abs su*).

We compare now the benchmark results with the cost model predictions. Figure 5.14 is an overview of cost model results for a relative finite difference SynDMAPM program.

The first observation is that the cost model predictions are too optimistic concerning the speedup. However, in Section 6.9 we explain in more detail a particular property of the PolyAPM cost model: its calibration is geared towards big problem sizes. In particular, even the predictions

		4 processor		16 processors	
size	seq cost	cost	su	cost	su
64	1086456	468449	2.32	213214	2.19
96	2468088	1031980	2.39	357417	2.88
128	4408824	1821303	2.42	558068	3.26
256	17762808	7236516	2.45	1925151	9.22

Figure 5.14: Cost Model Predictions for 2D Finite Differences

for small problem sizes match sufficiently well the behaviour of target code for large problem sizes. If, as a consequence, we compare the predicted speedup of 2.45 of the APM program with $n = 256$ with the benchmark results of the target program with $n = 4096$ (speedup 2.04), then the prediction is only off by 20%. Similarly, on 16 processors, we have a predicted speedup of 9.22 compared to the benchmarked result of 7.26. Again, the error is about 21%. If we consider the coarseness of the cost model, and the fact that it is meant to be only a decision support system and no hardware simulation for performance prediction, then the results are quite good.

Chapter 6

Case Study II: LU Decomposition

The following case study presents a PolyAPM-based program generation for the LU decomposition problem, including the exploration of design decisions with support from the cost model.

6.1 Problem Specification

We have chosen the non-pivotal LU decomposition of a non-singular square matrix $A = (a_{ij})$, $(i, j = 1, \ldots, n)$.

The result consists of one lower triangular matrix $L = (l_{ij})$, with unit diagonal, and one upper triangular matrix $U = (u_{ij})$, such that $A = LU$. L and U are defined recursively as follows [Ger78]:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} * u_{kj}, \quad j \leq i, \quad i = 1, 2, \ldots, n \tag{6.1}$$

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} * u_{kj}}{l_{ii}}, \quad j > i, \quad j = 2, \ldots, n \tag{6.2}$$

The Haskell implementation used for this example is shown in Figure 6.1. As in the previous chapter, note the close relationship between the problem specification and the code. In particular,

```
lu_decomp:: Array (Int,Int) Float ->
            (Array (Int,Int) Float, Array (Int,Int) Float)
lu_decomp a = (l,u)
  where
   l = array ((1,1), (n,n))
     [ ((i,j), a!(i,j) - sum [ l!(i,k)*u!(k,j) | k <- [1..(j-1)]])
      | i <- [1..n], j <- [1..n] , j<=i ]

   u = array ((1,2), (n,n))
     [ ((i,j), (a!(i,j) - sum [ l!(i,k)*u!(k,j) | k <- [1..(i-1)]])/ l!(i,i))
      | j <- [2..n], i <- [1..n], i<j ]

   (_ , (n, _)) = bounds a
```

Figure 6.1: Haskell Code for LU Decomposition

as the computations of L and U are mutually recursive, in most (i.e., strict) programming languages the programmer has to think about the data dependences between L and U in order to find a sequential schedule. This additional "serialisation" would have to be undone in a subsequent parallelisation. However, this is not necessary with the given Haskell code. The lazy semantics of Haskell ensures a flow of computation as the data dependences require, thus relieving the programmer of the burden to think about it.

In the following, we will present a sample derivation of the LU decomposition for a loop nest with an SPMD message passing interface. We treat the above program as two statements within a two-dimensional index space spanned by i and j. The scalar product with index k is considered an atomic part of each computation.

6.2 Parallelisation

The parallelisation is done by first determining the data dependences in the above program and then feeding these to the space-time mapping tools (i.e., the scheduler and allocator) of LooPo [GL96]. In particular, we compare two different space-time mappings returned by these tools.

6.2.1 Dependence Analysis of LU

By applying the dependence algorithm of Section 3.5 to the LU example, we end up with one component set S, comprising A, and the two mutually recursive arrays L and U, both referring to A. The set of dependences is presented in Figure 6.2.

Arr	No	Source	Dest.	Restr.	Restricted Index Space
L	1	$l(i,k)$	$l(i,j)$	$j \leq i$	$(i,j,k) \in \{1,\ldots,n\} \times \{1,\ldots,i\} \times \{1,\ldots,(j-1)\}$
	2	$u(k,j)$	$l(i,j)$		
U	3	$l(i,k)$	$u(i,j)$	$i < j$	$(i,j,k) \in \{1,\ldots,(j-1)\} \times \{2,\ldots,n\} \times \{1,\ldots,(i-1)\}$
	4	$u(k,j)$	$u(i,j)$		
	5	$l(i,i)$	$u(i,j)$		

Figure 6.2: Dependences in the LU Example

This set of dependences describes a partial order on the index space of L and U. The figure contains, for each array, the referenced array elements (referenced by itself or others), the restriction imposed on the original array's domain definition (an affine relation) and the restricted index space of the dependence. To get an idea of the structure of the dependences, Figure 6.3 contains a graphical presentation of the index space with L's dependences on the left and U's dependences on the right. The numbers next to the different types of arrow styles in the legend correspond to the dependence numbers in Figure 6.2. The reason for the different dimensionality of the two index spaces is the structure of the body computations: an intermediate list is generated by k and subsequently summed up. This index k provides an additional dimension for the index space of the dependences.

Dependences and index spaces are the input for the scheduling and allocation algorithms as is described in the next section.

6.2.2 First Schedule and Allocation (STM1)

This section describes the identification of a schedule and allocation, which comprise the first space-time mapping (STM1). Index space, dependences and variables constitute a *program speci-*

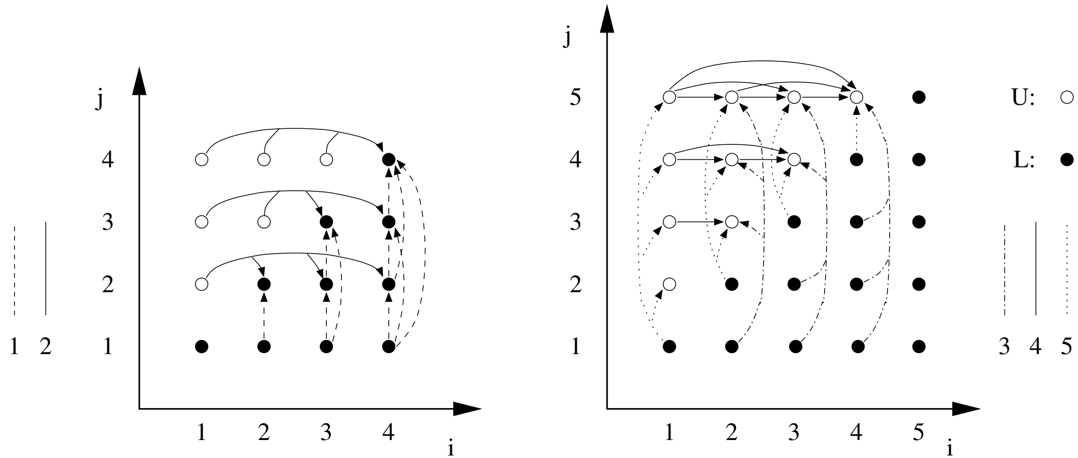


Figure 6.3: Dependences between Array Elements

fication that serves as an input for LooPo. If the input were an imperative loop program, LooPo could perform the dependence analysis and generation of a program specification by itself. As we use Haskell and perform our own, albeit simpler dependence analysis, we have to provide the specification ourselves. The corresponding information about the LU decomposition algorithm is depicted in the specification in Appendix A.1.

We use LooPo interactively to analyse the specification. Standard LooPo input is an imperative loop program in Fortran or C. These programs are parsed and subjected to a dependence analysis. At this point, LooPo's internal data structures contain the index space and the dependences, ready for a parallelisation. We already have the index space and dependences. We can use the `spectoloopo` module that parses the specification and transforms it into LooPo's internal representations. LooPo's main window after loading the `.spec` file is presented in Figure 6.4. Next we use the Feautrier scheduler [Wie95, Fea92] to get a schedule for the specification and obtain:

$$\theta_l(i,j) = 2*(j-1) \quad (6.3)$$
$$\theta_u(i,j) = 2*(i-1)+1 \quad (6.4)$$

This schedule honours the fact that the definitions of L and U are mutually recursive, so that their overall computation is interleaved. Note that, at each point in time, several computations can be performed, e.g., at logical time $2*(j-1)$ we can compute $l(i,j)$ for all i.

The mapping of computations, which are performed at the same time, to virtual processors is defined by the allocation function. Finding a valid function is not difficult, since every mapping from the set of parallel computations to the natural numbers will do. The difficulty is finding a sensible function which minimises the number of communications. This is done by placing dependences on single processors, i.e., allocating a computation on the same processor as a previous computation it depends on. Then, the data item can simply stay in the local memory of the processor. Up to now, no provably optimal algorithm for generating an allocation has been found, but some heuristic algorithms have been proposed [Fea94, DR95].

We use LooPo's Feautrier allocator [Wie95, Fea94] to compute suitable allocation functions for the LU example:

$$\sigma_l(i,j) = i \quad (6.5)$$
$$\sigma_u(i,j) = j \quad (6.6)$$

We combine schedule and allocation to a single transformation matrix that is used to perform a coordinate transformation of the index pairs (i,j) into (t,p). The variables t and p correspond

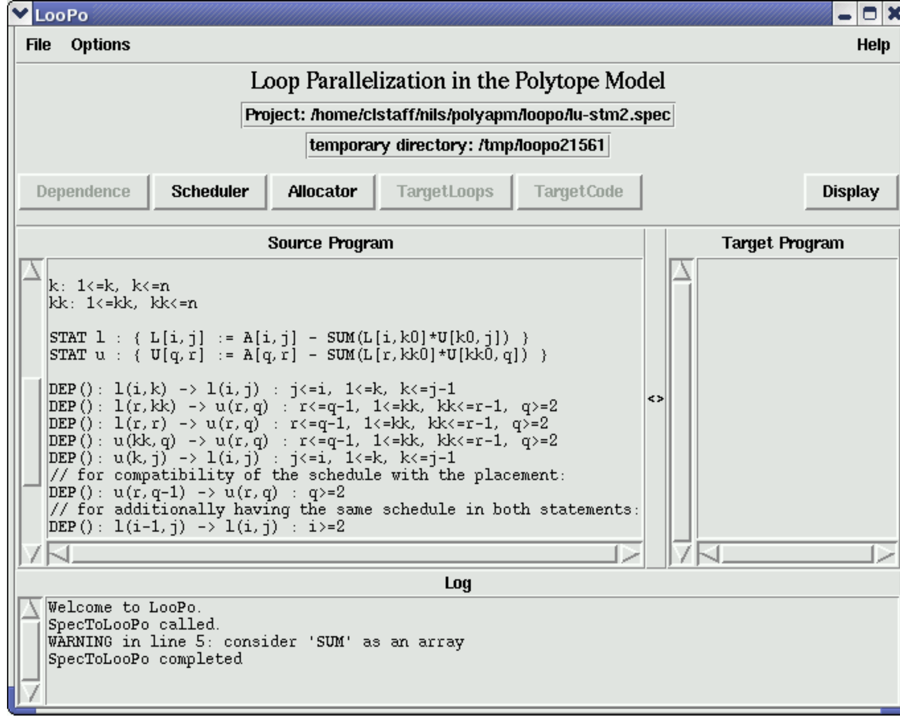


Figure 6.4: LooPo with Loaded `.spec` File

to the enumeration of time and processors, resp. The scheduler module generates a schedule for each computation, so that we have two matrices, one for L and one for U. Generally, the relation

$$T \cdot \binom{i}{j} + \vec{d} = \binom{t}{p}$$

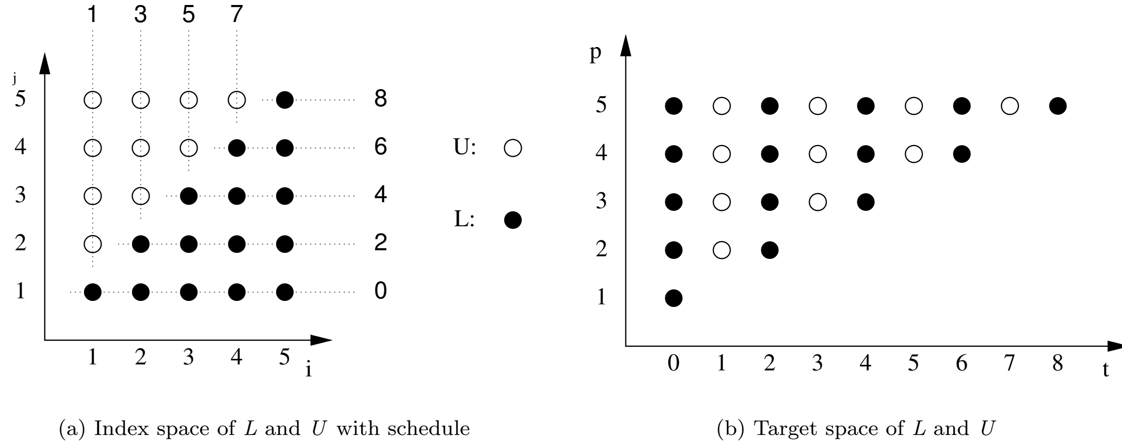
holds, in this case, denoting a mapping from the index space $\{(i,j) \mid (i,j) \in \mathbb{N}_+^2\}$ to the target space $\{(t,p) \mid (t,p) \in \mathbb{N}^2\}$. We obtain:

$$T_L = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix} \ ; \ \vec{d}_L = \begin{pmatrix} -2 \\ 0 \end{pmatrix} \ ; \ T_U = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \ ; \ \vec{d}_U = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

This matrix presentation combines the data necessary for the coordinate transformation.

To see the effect on the index space, Figure 6.5(a) presents the computations of L and U in a single index space (their domains do not intersect, so that they could even be stored in a single matrix). In this example, we choose a value of 5 for n. The data points which are independent of each other lie on the same dotted schedule line. The target space in Figure 6.5(b) depicts the points after the coordinate transformations. Data items with the same schedule now have the same value t, meaning that they are computed at the same time.

Thus, as a result of the parallelisation, the computation of the LU decomposition of an n by n matrix has been accelerated from n^2 to $2n-1$ virtual time units, where one unit is the time to compute a single data item. This time unit depends on the level of abstraction that we have chosen here. In the Haskell program in Figure 6.1, each computation of L and U contains the summation of a list whose length is $O(n)$. Taking this into account, a refined parallel solution requires about $3n$ time units, which corresponds to the results of others [Che86].



(a) Index space of L and U with schedule

(b) Target space of L and U

Figure 6.5: Before and After the First Space-Time Mapping

6.2.3 Second Schedule and Allocation (STM2)

This subsection explores a second and possibly better space-time mapping. The allocation of the first space-time mapping (STM1) results in two of the five dependences being *cut*, i.e., the computations connected with the source and destination of a dependence are allocated to the same processor so that no communication is necessary. In general, it is desirable to cut as many dependences as possible to reduce the number of communications. A closer look at the dependences, as shown in Figure 6.2, hints that it is possible to cut three of the five dependences. This can be achieved by a different space-time mapping.

Finding this second space-time mapping (STM2) is slightly different from the first, which is LooPo's default output. The scheduling and allocation techniques used by LooPo are sensitive to the order in which the dependences are listed. We change the order of the dependences until the resulting allocation cuts the desired number of dependences.

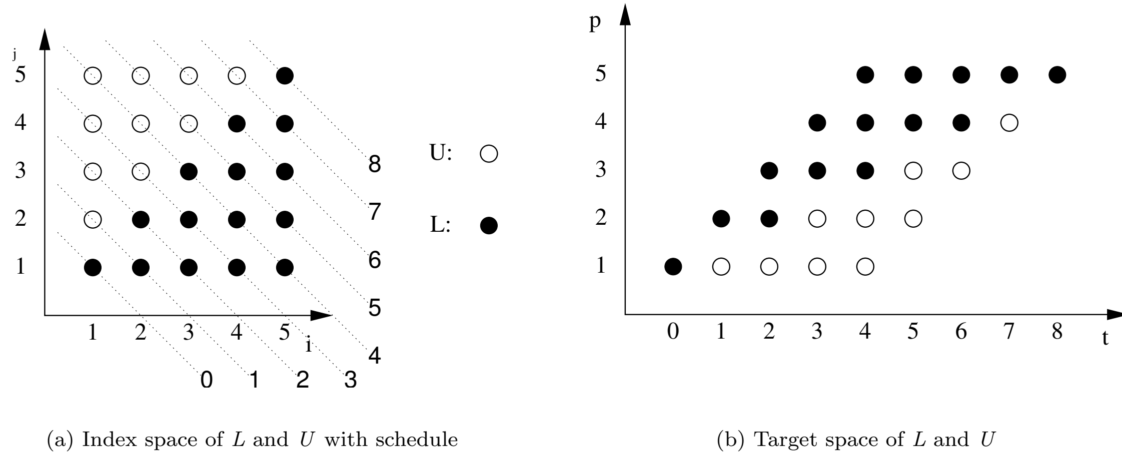
As the LooPo scheduler and allocator are independent phases, it frequently happens that schedule and allocation are linearly dependent. As they have to be linearly independent to form a linear mapping, such result pairs have to be disregarded. In our particular case, we have to add two dummy dependences to avoid such an illegal space-time mapping. The first additional dependence enforces linear independence and the second ensures that both schedules are identical. The latter is important as we wish to place both computations within one loop nest. The order of dependences used to produce the first space-time mapping avoids these problems directly. The resulting `.spec` file is shown in Appendix A.2. It is important to note the the tricks used here to force a certain parallelisation are not part of PolyAPM. In order to compare two different parallel loop nests of the same algorithm, we used LooPo to generate an alternative parallelisation. The tricks were necessary to force LooPo into considering the alternative. Help from the LooPo maintainer was required to succeed [Gri02].

With this input we use the same scheduler and allocator as in Section 6.2.2 and get a new space-time mapping with the desired properties. The new schedule is

$$\theta_l(i,j) = i+j-2 \quad (6.7)$$
$$\theta_u(i,j) = i+j-2 \quad (6.8)$$

and likewise the new allocation is

$$\sigma_l(i,j) = i \quad (6.9)$$
$$\sigma_u(i,j) = i \quad (6.10)$$



(a) Index space of L and U with schedule

(b) Target space of L and U

Figure 6.6: Before and After the Second Space-Time Mapping

We use them to construct the new space-time mappings:

$$T_L = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \; ; \; \vec{d}_L = \begin{pmatrix} -2 \\ 0 \end{pmatrix} \; ; \; T_U = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \; ; \; \vec{d}_U = \begin{pmatrix} -2 \\ 0 \end{pmatrix}$$

The index space of L and U augmented with the new schedule is depicted in Figure 6.6(a). The new target index space is shown in Figure 6.6(b). Note that, compared to STM1, the execution time and the load balancing did not change, but we cut more dependences than before, which gives hope for fewer communications.

6.3 Initial Code Generation

Given the schedules and allocations as determined in the previous section, we want to construct two synchronous loop nests as the first level of a set of APM programs. Since we obtained one-dimensional schedules and allocations, the loop nest consists of one outer sequential and one inner parallel loop. The loop bounds are computed by applying the space-time mapping to the index space bounds.

The generation of the APM programs is straightforward. The calculations of the loop bounds can be done by hand using Fourier-Motzkin elimination [Sch86] or one can use LooPo's target code generation. The result is one loop nest for each space-time mapping. Within this loop nest, a body function containing the statements is inserted. Statements are given by the expression that computes one array element. This is the same granularity as provided to the scheduler and allocator. Within the statements all references to index variables have to be space-time mapped as well. The inner scalar product still remains a Haskell array comprehension. In addition, all computations within the body have to be augmented with user provided computation costs. They are added to the current processor's entry in the cost array.

As a result we get two synchronous APM programs. For the STM1 version see Figure 6.7. The loop nest is two-dimensional, comprising an outer sequential loop ($0 \leq t < 2*n$, stride 1) and an inner parallel loop ($\lfloor \frac{t}{2} \rfloor + 1 \leq p < n+1$, stride 1). The body function of SynAPM takes some *state*, consisting of memory, structure parameters and computation statistics, and a list of current values of all surrounding loops, to return an updated state. Here, the memory is defined as `LUmem`, a triple of the three arrays A, L and U.

```
--                      k-loop          -   if
lcost t = (fromIntegral (t`div`2)*lfact +1 + 2)
--                       k-loop             -  /   if
ucost t = (fromIntegral ((t-1)`div`2)*ufact +1 +1 + 5)

data LU_SynAPM = LU_SynAPM
instance Synchronizing LU_SynAPM LUmem -- id default

loop_s = LP [(Seq, \([],(n:_))-> 0,
                   \([],(n:_))-> 2*n,
                   \([],(n:_))-> 1),
                (Par, \([t],(n:_))-> t`div`2+1,
                      \([t],(n:_))-> n+1,
                      \([t],(n:_))-> 1)]
              (BD body_s)

body_s:: (LUmem, [Idx], CompStat) -> [Idx]
        -> (LUmem, [Idx], CompStat)
body_s  ((a,l,u),(n:splist),ca) [t,p] =
    if (t `mod` 2 == 0)
    then ((a,l_new,u), (n:splist), ca//[(p,(lcost t) +ca!p)])
    else if (t+2)`div`2 == p
         then ((a,l,u),(n:splist),ca)
         else ((a,l,u_new), (n:splist), ca//[(p,(ucost t)+ca!p)])
    where l_new = stmnt1 a l u (t,p,n)
          u_new = stmnt2 a l u (t+1,p,n)
          stmnt1 a l u (t,p,n) =
              l //  [((p,t`div`2+1) , a!(p,t`div`2+1)
                     - sum ([ l!(p,k)*u!(k,t`div`2+1)
                              | k <- [1..t`div`2]]))]
          stmnt2 a l u (t,p, n) =
              u // [(((t+1)`div`2, p), (a!((t+1)`div`2, p)
                                   - sum ([ l!((t+1)`div`2,k)*u!(k,p)
                                            | k <- [1..(t-1)`div`2]]))
                    / l!((t+1)`div`2, (t+1)`div`2))]
```

Figure 6.7: Synchronous APM Code of LU Decomposition (STM1)

```
l2cost t p = fromIntegral (t+2-p-1)*lfact+1
u2cost t p = fromIntegral (p-1)*ufact+1

loop_s2 = LP loop_bounds_s2 (BD body_s2)
loop_bounds_s2 =
    [(Seq, \([],(n:_))-> 0,
           \([],(n:_))-> n*2-2+1,
           \([],(n:_))-> 1),
      (Par, \([t],(n:_))-> min (max 1 (2+t-n)) (ceildiv (2+t) 2),
            \([t],(n:_))-> (max (min n (1+t)) ((1+t) `div` 2))+1,
            \([t],(n:_))-> 1)]

body_s2::(LUmem, [Idx], CompStat) -> [Idx]
         -> (LUmem, [Idx], CompStat)
body_s2 ((a,l,u),(n:splist), ca) [t,p] =  --trace (show (t,p))
   ((a,l_new,u_new), (n:splist), ca//[(p,cost_new +ca!p)])
    where (l_new,u_new,cost_new) = inner_body_STM2 (a,l,u) n [t,p]

inner_body_STM2 (a,l,u) n [t,p] =
    if ((ceilDiv (2+t) 2) <= p)
    then (stmnt1 a l u (t,p,n), u, (l2cost t p)+6.0)
    else (l, stmnt2 a l u (t, p, n), (u2cost t p)+6.0)
    where stmnt1 a l u (t,p,n) =
              let i = p
                  j = t+2-p
              in l  //  [((i,j) , a!(i,j)
                          - sum ([ l!(i,k)*u!(k,j)
                                   | k <- [1..(j-1)]]))]
          stmnt2 a l u (t,p, n) =
              let r = p
                  q = t+2-r
                  i = r
                  j = q
              in u// [((i,j), (a!(i, j)
                            - sum ([ l!(i,k)*u!(k,j)
                                     | k <- [1..(i-1)]]))
                          / l!(i,i))]
```

Figure 6.8: Synchronous APM Code of LU Decomposition (STM2)

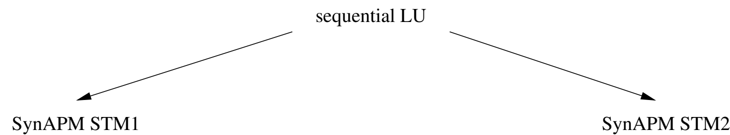
Any predefined synchronous machine requires an instance declaration of class `Synchronizing`. In order to distinguish different instances using the same memory data type, we define a label type for every program and include it in the instance declaration. Here the label type is `LU_SynAPM`. As we do not require any memory synchronisation for this program, no body for `synchronizeMem` is supplied. In that case the run time system inserts the default instance of `synchronizeMem` which is the identity.

The computational work for elements of L and U differs; the two functions `lcost` and `ucost` capture these cases. The main cost stems from the scalar product ranging over k. In the case of L, for each of the $\frac{t}{2}$ k's there is a multiplication of cost `lfact`, plus some additional cost for the subtraction of the scalar product and the `if` guards governing the statement.

Most of the details of the STM1 program above also apply to the STM2 version as depicted in Figure 6.8. The different space time mapping results in different loop bounds: ($0 \leq t < 2 * n - 2$, stride 1) and an inner parallel loop ($\mathsf{min}(\mathsf{max}(1, 2 + t - n), \lceil \frac{2+t}{2} \rceil) \leq p < \mathsf{max}(\mathsf{min}(n, 1 + t), \lfloor \frac{1+t}{2} \rfloor)$, stride 1). The indices within the loop body are adapted and, due to the different computation structure, the computation cost differs from the STM1 version.

6.3.1 Evaluation of Two Alternative Programs

At this point we have two alternative SynAPM programs that lead to separate development branches further on. We can either pursue both or try to decide upon one of them.



One instrument of decision support in the PolyAPM framework is the execution of an APM program in order to get the estimated run time cost. To compare the two programs, we run them with an input of a 512×512 matrix A on four processors. The cost model instance used is again the SCI Linux cluster. First the result of STM1:

```
Original: final of L: 6.0247297, final of U: -2.4276931

SynAPM - STM1 - Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total (no comms): 1.5718965e8, agg. work: 1.351341e8
```

This simulation took 7 minutes on a Dual Intel Xeon 2.4 GHz. We observe that the results of the SynAPM program coincide with the sequential LU program. The anticipated cost is about $1.571 * 10^8$. The corresponding result of STM2 is:

```
SynAPM - STM2 - Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total (no comms): 1.5703896e8, agg. work: 1.3565974e8
```

In this case, the estimated cost is $1.570 * 10^8$. Considering that the PolyAPM cost model is quite simplistic and produces only rough estimates, these two results are too close to justify any bias towards either space-time mapping. In the following we will pursue both branches.

In this programming model, each virtual processor is considered to be a real one and thus is communicating. The SynAPM program does not define any communication operations, as it is considered to be shared memory, but a global synchronisation is necessary after each time step to enforce correctness. The amount of aggregated work on all processors is about $1.35 * 10^8$, which is

less than the PolyAPM cost, yielding a theoretical speedup of 0.86. The reason is that the amount of work on each of the 512 processors at each time step is outweighed by the synchronisation cost. So, at this stage, the parallelisation was not yet worth the effort.

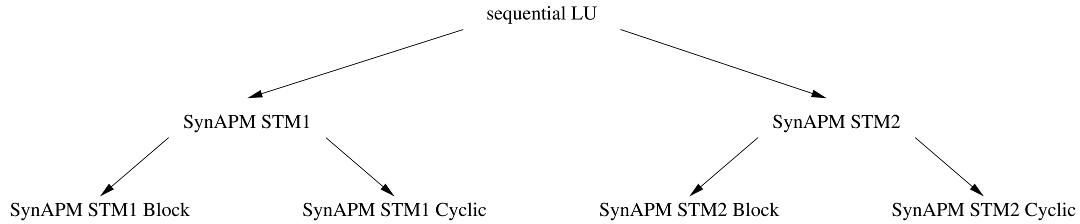
These are just example data points. More runs with different input sizes show a similar picture, although the speedup gets slightly better the bigger the input size is. So, from examining this very abstract program with different input sizes, we can already deduce that we can expect better speedups with greater problem sizes. Note that there is no analytical evidence of this, we just draw conclusions from operational observations.

6.4 First APM Transformation: Processor Tiling

The PDG in Figure 3.1 contains several tiling choices. We will explore two of them in this case study, namely blocked tiling and cyclic tiling.

Both have a similar effect on the synchronous code: the parallel loop performs fewer iterations and an additional inner sequential loop is inserted to compensate for the missing iterations. As a result, fewer parallel processors do more work at each time step. Since these changes do not require new machine characteristics, the resulting tiled programs also run on SynAPM.

Each of the two space-time mapped programs can have any of the two tiling methods applied to them, thus yielding four new APM programs:



The tiled APM programs differ from their predecessors only in their loop bounds. Therefore, we just present the loop bounds of the four tiled programs, starting with STM1.

Here are the STM1 block tiled loop bounds with tile size defined as $\lfloor \frac{n-1}{\text{maxp}} \rfloor + 1$. Note the stride of 1 in the innermost loop – neighbouring virtual processors are mapped onto the same physical processor.

```
loop_bounds_s_tl =
    [ (Seq, \([],(n:maxp:_))-> 0,
            \([],(n:maxp:_))-> 2*n,
            \([],(n:maxp:_))-> 1),
         (Par, \([t],(n:maxp:_))-> 0,
               \([t],(n:maxp:_))-> maxp,
               \([t],(n:maxp:_))-> 1),
            (Seq, \([t,rp],(n:maxp:_))-> max (rp*(tilesize_lu n maxp)+1)
                                             (t`div`2+1),
                  \([t,rp],(n:maxp:_))-> min ((rp+1)*(tilesize_lu n maxp)+1)
                                              (n+1),
                  \([t,rp],(n:maxp:_))-> 1)]
```

In the STM1 cyclic tiled loop nest, those virtual processors whose processor numbers differ by a multiple of `maxp` are mapped onto the same physical:

```
loop_bounds_s_ctl =
    [ (Seq, \([],(n:maxp:_))-> 0,
            \([],(n:maxp:_))-> 2*n,
            \([],(n:maxp:_))-> 1),
       (Par, \([t],(n:maxp:_))-> 0,
             \([t],(n:maxp:_))-> maxp,
             \([t],(n:maxp:_))-> 1),
        (Seq, \([t,rp],(n:maxp:_))-> 1+rp+ (ceildiv (t`div`2+1-rp-1)
                                                    maxp)*maxp,
              \([t,rp],(n:maxp:_))-> n+1,
              \([t,rp],(n:maxp:_))-> maxp)]
```

The STM2 loop bounds have corresponding properties. First, the block tiled version:

```
loop_bounds_s2_btil =
    [ (Seq, \([],(n:maxp:_))-> 0,
            \([],(n:maxp:_))-> n*2-2+1,
            \([],(n:maxp:_))-> 1),
       (Par, \([t],(n:maxp:_))-> max (ceildiv (-(tilesize_lu n maxp)+1)
                                              (tilesize_lu n maxp))
                                     (ceildiv (t-n-(tilesize_lu n maxp)+2)
                                              (tilesize_lu n maxp)),
             \([t],(n:maxp:_))-> (min (t `div` (tilesize_lu n maxp))
                                      ((n-1) `div` (tilesize_lu n maxp)))+1,
             \([t],(n:maxp:_))-> 1),
        (Seq, \([t,rp],(n:maxp:_))-> max (max (t-n+2) 1)
                                         ((tilesize_lu n maxp)*rp+1),
              \([t,rp],(n:maxp:_))-> (min (min n (t+1))
                                          ((tilesize_lu n maxp)*rp+
                                           (tilesize_lu n maxp)))+1,
              \([t,rp],(n:maxp:_))-> 1)]
```

And the STM2 cyclic tiled loops:

```
loop_bounds_s2_ctil =
                [ (Seq, \([],(n:maxp:_))-> 0,
                        \([],(n:maxp:_))-> 2*n-2+1,
                        \([],(n:maxp:_))-> 1),
                  (Par, \([t],(n:maxp:_))-> 0,
                        \([t],(n:maxp:_))-> maxp-1+1,
                        \([t],(n:maxp:_))-> 1),
                  (Seq, \([t,rp],(n:maxp:_))->
                      max (ceildiv (t-rp-n+1) maxp)
                          (ceildiv (-rp) maxp),
                        \([t,rp],(n:maxp:_))->
                          (min ((-rp+n-1) `div` maxp)
                               ((t-rp) `div` maxp))+1,
                        \([t,rp],(n:maxp:_))-> 1)]
```

These different processor tilings result in different computational loads. Cyclic tiling is often able to smooth an unbalanced load distribution. The STM1 index space in Figure 6.5(b) shows a continuous increase of load for higher processors. It is obvious that a cyclic tiling changes the load imbalance.

At this time, one would like to terminate the development of at least some of the branches in the evolving program tree. A cost model evaluation of the tiled APM programs provides some

guidance for this purpose. In the following, we present APM simulations on four processors and an input size of 512.

```
SynAPM - STM1 - Blocked Tiling Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total (no comms): 7.891381e7, agg. work: 1.351341e8
----------------------------------------
SynAPM - STM1 - Cyclic Tiling Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total (no comms): 3.4998304e7, agg. work: 1.351341e8
----------------------------------------
SynAPM - STM2 - Blocked Tiling Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total (no comms): 7.599721e7, agg. work: 1.3565974e8
----------------------------------------
SynAPM - STM2 - Cyclic Tiling Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total (no comms): 3.4982304e7, agg. work: 1.3565974e8
```

The total PolyAPM costs show that, in the cost model, the two cyclic tiled programs are far superior to their blocked counterparts. It further shows that the PolyAPM cost is less than the accumulated workload, so that we can expect speedups in all four cases. However, a distinction between the two space-time mappings cannot be made. It is also good practise not to regard the absolute cost values too seriously as the cost model is simplistic and not a complete hardware simulation.

A shared memory OpenMP program written in C could be derived directly from the tiled programs, and the above results clearly suggest to use a cyclic tiling.

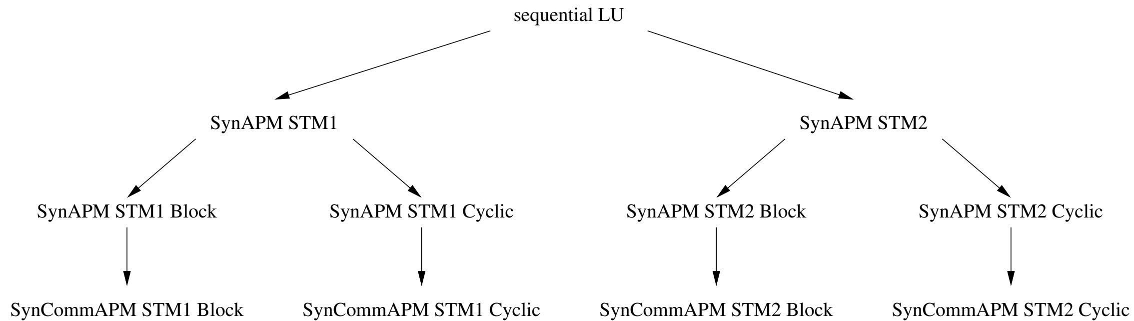
In practise, one might want to pursue only the two cyclic branches. In this case study, however, we follow all four branches to see whether the cost model is right in its bias.

6.5 Second APM Transformation: Generation of Communications

The second transformation performs the generation of communications, aiming at a point-to-point message passing library, while still keeping a shared memory. The changes of the previous tiling transformation only affected the loops. With the generation of the messages, we need to:

- change the memory data type into a state data type that combines memory and message queue and change the body function accordingly,
- add the implementation of `generateMsg`, and
- `updateMem` and change the APM to SynCommAPM.

This transformation only allows for one possible result for each input program so that the four tiled APM programs lead to four SynCommAPM programs:



It is not necessary to report on all four code examples to present the effects of this transformation. We show just the changed and added parts of the STM1 block tiled program in the order of the list above.

6.5.1 From Abstract Memory to Machine State

The memory type of the previous LU APM programs is called `LUmem` and contains a triple of the three arrays A, L and U. The new state that SynCommAPM uses is `GlobStateSC` and it comprises the global shared memory of type `LUmem` and a message queue for the communications. The loop body does not deal with communications, so the only change is that a different data structure now contains the memory and only that memory is updated. The message queue is left untouched by the body function and just passed through.

```
body_s_sc:: (GlobalStateShM LUmem b c d, [Idx]) -> [Int]
             -> ((GlobalStateShM LUmem b c d, [Idx]), [Int])
body_s_sc (GStateShM (a,l,u) msgs
           ((Comps crange (ca:compsal)):clist),[n,maxp]) (idxlist@[t,rp,p]) =
  ((GStateShM mem msgs
    ((Comps crange ((ca//[(rp,cost+ca!rp)]):compsal)):clist), [n,maxp]), idxlist)
   where (mem,cost) = ...
```

6.5.2 Generating Point-to-Point Messages with genMsg

After each body iteration the interpreter calls generateMsg to create messages that send the values just computed, if necessary. In the polytope model communication takes place along data dependences in the target space. Relevant are only those dependences whose source and destination are located on different processors. To match this model, we define a list of dependences `deps_s_sc` and a function `genMsgforDep` that generates necessary messages for one such dependence. All that the function `generateMsg` has left to do is to call `genMsgforDep` for each dependence and place all results in a single list.

```
deps_s_sc = [(2, \([t,rp,p],(n:maxp:_)) ->(t `mod` 2 == 1)&&(2*p-2>t), U,
                 \([t,rp,p],(n:maxp:_)) -> [(rp', t+1) | rp' <- [rp+1 .. maxp-1]]),
             (3, \([t,rp,p],(n:maxp:_)) ->(t `mod` 2 == 0)&&(t/=p+1), L,
                 \([t,rp,p],(n:maxp:_)) -> [(rp', t+1) | rp' <- [rp+1 .. maxp-1]]),
             (5, \([t,rp,p],(n:maxp:_)) ->(t `mod` 2 == 0)&&(t==p+1), L,
                 \([t,rp,p],(n:maxp:_)) -> [(rp', t+1) | rp' <- [rp+1 .. maxp-1]])
             ]

instance Sendable LU_blocked_SynCommAPM LU_Dom (Int,Int) Float LUmem where
  generateMsg LU_blocked_SynCommAPM [t,rp,p] splist (a,l,u) =
      recur deps_s_sc (genMsgforDep [t,rp,p] splist (a,l,u))

genMsgforDep  [t,rp,p] splist (a,l,u) (nr, guard, dom, generator) =
               ([Msg (rp, to_p, t, to_tm, dom,
                      if dom == L then (p,t`div`2+1) else ((t+1)`div`2, p),
                      if dom == L then l!(p,t`div`2+1) else u!((t+1)`div`2, p),1)
                 | (to_p,to_tm) <- generator ([t,rp,p],splist),
                   guard ([t,rp,p],splist)
                ], (rp,gen_cost))
```

However, the above code is going to be inefficient, whether used in an APM program or translated to a function in a target program. The automatic generation helps to derive a correct program, but it has only few optimisation options to offer. Instead, we retreat to a semi-automatic generation. After testing the SynCommAPM program with messages as generated above, we started to derive a hand optimised version of it. It can be seen easily that dependence 5 is just an extension of dependence 3, so that both can be combined. Furthermore, in all cases the just computed value is sent to processors with a higher id than the sender. Dependence 2 does this during odd numbered time steps, and dependences 3 and 5 do this for the even numbered ones. It is just the additional boolean guard $2p-2>t$ in dependence 2 that needs special attention. It prevents message generation at some points of the index space that the loop bounds enumerate. Based on these findings, we now present a hand optimised version of `generateMsg`. The generation costs have been adapted. A detailed comparison and many tests confirmed the notion that both versions of `generateMsg` are equivalent.

```
instance Sendable LU_blocked_SynCommAPM LU_Dom (Int,Int) Float LUmem where
  generateMsg LU_blocked_SynCommAPM [t,rp,p] (n:maxp:_) (a,l,u)  =
      if(t`mod`2==0)          -- L computation? Deps 3 and 5
      then ([Msg (rp, to_p, t, t+1, L, (p,t`div`2+1), l!(p,t`div`2+1),1)
           | to_p <- [rp+1 .. maxp-1]], (rp, gen_cost_B1))
      else if (2*p-2>t)       -- U computation? Dep 2
           then ([Msg (rp, to_p, t, t+1, U, ((t+1)`div`2, p), u!((t+1)`div`2, p),1)
                 | to_p <- [rp+1 .. maxp-1]],(rp, gen_cost_B1+3.0))
           else ([], (rp, 5.0))
     where gen_cost_B1 = fromIntegral ((maxp-1)-(rp+1))  -- msgs cost
                         + 2.0                           -- 1st if guard
```

6.5.3 Receiving Messages with `updateMem`

When the interpreter scans through the global message queue at the end of each time step, it identifies all messages that are to be delivered before the next time step. Those messages are removed from the queue and the function `updateMem` is called on each of them to deliver the message, i.e., to make the message's content persistent in memory. At this point, a run time check is included that prevents the input array A from being updated.

```
instance Updatable LU_blocked_SynCommAPM LU_Dom (Int,Int) Float LUmem where
  updateMem LU_blocked_SynCommAPM
            msg@(Msg (from_p, to_p, from_tm, to_tm, dom, idx, val, cost)) (a,l,u)  =
                      (case dom of
                       A -> error "no update of A!"
                       L -> (a,l//[(idx,val)],u)
                       U -> (a,l,u//[(idx,val)]))
```

6.5.4 Evaluating the Transformation

The introduction of communications has an impact on the calculation of the cost model. The cost model computation must now take the sending and receiving of messages into account. The calculated cost for each run differs therefore from the tiled program's results. To get more insight into the operational behaviour of the program, the following numbers are presented as lists with one entry per processor:

- the total computation cost (work totals): this is a sum of the entire computation cost per processor
- the number of sent messages per processors (send totals)
- the number of received messages per processors (recv totals)
- the sum of the communication costs of all time steps per processor (comm totals)

The following results are taken from a simulation with four processors and an input size of 512. For STM2 we have omitted the communication details.

```
SynCommAPM - STM1 - Blocked Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total: 8.010185e7
 work totals: [2302656.0,1.5246016e7,4.0706752e7,7.868487e7]
 send totals: [765,1022,767,0] = 2554
 recv totals: [0,255,766,1533] = 2554
 comm totals: [977420.94,1092062.5,1238611.3,1417066.8]
----------------------------------------
SynCommAPM - STM1 - Cyclic Pt2PT SHMEM Comms Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total: 3.6484896e7
 work totals: [3.4075012e7,3.4272008e7,3.4469752e7,3.4668264e7]
 send totals: [3051,3049,3047,3045] = 12192
 recv totals: [3039,3045,3051,3057] = 12192
 comm totals: [1809754.8,1812032.5,1814315.6,1816597.0]
----------------------------------------
SynCommAPM - STM2 - Blocked Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total: 7.801139e7
...
----------------------------------------
SynCommAPM - STM2 - Cyclic Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total: 3.6487444e7
...
```

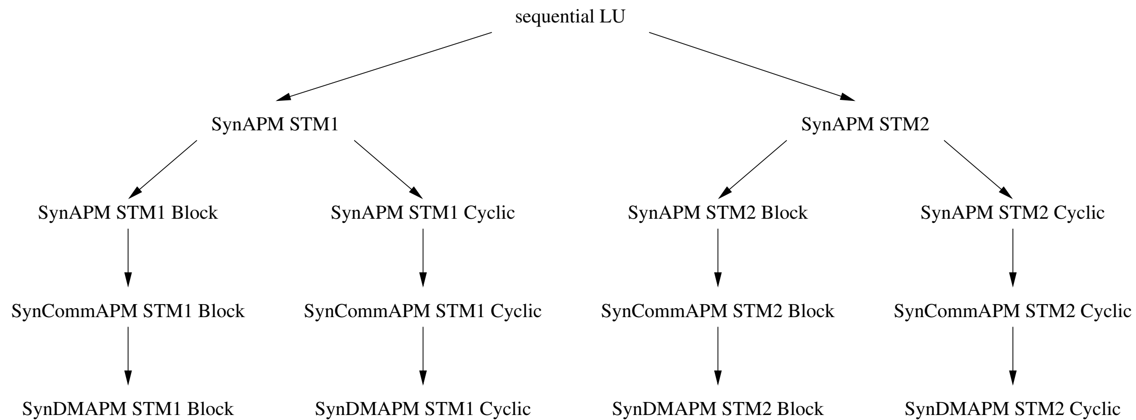
We observe that the cyclic program's workload is balanced over the processors, while this is not true for the blocked programs (line `work totals`). Similarly, sending and receiving messages are unevenly distributed in the blocked programs and more evenly in the cyclic programs (lines `send` and `recv totals`). Although the number of messages in the cyclic case is almost five times higher than in the blocked case ($12192 > 2554$), this ratio is not reflected in the communication costs (line `comm totals`). This is because their largest component is the cost for the synchronisations, which is equal for all processors. The only difference in the total PolyAPM cost between the tiled SynAPM program and the SynCommAPM program is the message cost.

The overall judgement of the four programs does not change with the introduction of communications: the cyclic programs are still ahead and close to each other, and the blocked programs remain also close, but way behind. This is because the Scali cluster has fast and low latency communications. Using a cost model profile of a machine with slower communications would exhibit a greater difference in the performance prediction.

6.6 Third APM Transformation: Memory Distribution

So far, the global shared memory consists of a triple of n by n matrices. The SynDMAPM requires a local memory for each processor. The simple approach is to replicate this triple on all processors, which is what we do in the current program. A further transformation could eliminate unused parts of each matrix by performing a memory footprint analysis. However, the index expressions would become more complicated so that there is a trade-off between memory consumption and computational complexity.

Code-wise the changes are located in those parts where memory is accessed. This applies to the `body`, `updateMem` and `generateMsg` functions. However, only the memory accesses have to be adapted to the new memory layout. The rest of these functions remains unchanged. Each SynCommAPM program turns into one SynDMAPM program, yielding a new set of four LU program variations:



As for the actual code changes, we present the STM1 blocked program for SynDMAPM as an example. The `GlobStateSC` is replaced by the local `DM_State` that contains a computation cost list (`CCL`) to store the computation cost of a body call. At the end of each time step, all local CCLs are emptied and their values are aggregated into the global cost structure.

```
body_s_dm::  (LU_DM_State, [Idx])
             -> [Idx]
             -> ((LU_DM_State, [Idx]), [Idx])
body_s_dm   (LState (a,l,u) msgs (CCL statl),[n,maxp]) (idxlist@[t,rp,p]) =
    ((LState mem msgs (CCL (cost:statl)),[n,maxp]), idxlist)
    where (mem,cost) = ...
```

The other function that needs to be changed is `updateMem`, where also a trivial change from `GlobStateSC` to `DM_State` is necessary. Interestingly, the function `generateMsg` does not need to be changed: as the new, user defined local memory type is the same as the former global shared memory type (a triple of arrays), the instance from SynCommAPM can be used.

The cost model values of SynDMAPM programs are typically the same as the ones of SynCommAPM programs, unless due to an increase of complexity of index values (see above) the computation costs are adjusted. The split of the shared memory did not impact the cost model computations. In the following results, we have omitted the detailed communication information:

```
SynDMAPM - STM1 - Blocked Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total: 8.010185e7
----------------------------------------
SynDMAPM - STM1 - Cyclic Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total: 3.6484896e7
----------------------------------------
SynDMAPM - STM2 - Blocked Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total: 7.801139e7
----------------------------------------
SynDMAPM - STM2 - Cyclic Result:
final of L: -558.78735, final of U: 5.9737787
Cost model, total: 3.6487444e7
```

While the cost model results did not change with the last transformation, the simulation times using the APM interpreters did. In general, SynCommAPM programs are slower than their SynDMAPM counterparts, because the interpreter keeps only one global message queue, which is inefficient compared to separate message queues for each processor as in SynDMAPM.

6.7 The Results of the APM Transformations

The following table is an overview of the cost model results we have obtained so far:

	STM1		STM2	
SynAPM	15.718965e7		15.703896e7	
tiling	block	cyclic	block	cyclic
SynAPM	7.891381e7	3.4998304e7	7.599721e7	3.4982304e7
SynCommAPM	8.010185e7	3.6484896e7	7.801139e7	3.6487444e7
SynDMAPM	8.010185e7	3.6484896e7	7.801139e7	3.6487444e7

It is obvious that the transformations after tiling did not change the order in which the cost model ranks the different programs. Therefore our observation after the tiling transformation, namely that the cyclic tiled programs should be used and the block tiled programs disregarded, still holds.

The introduction of communication did not significantly change the total cost model values. This is because the Scali cluster's cost model profile allots relatively small send and receive costs for a message compared to the synchronisation costs. The latter are used from SynAPM programs onwards, so that only the message costs are new in SynCommAPM programs.

In the next section we show how to derive an MPI program and how that compares with the performance predictions of our cost model.

6.8 Generating C+MPI Target Code

Starting from the four SynDMAPM programs, we create C+MPI programs using the `mpi_apm` library (Section 4.3.1). In the following, we describe how the key parts of an SynDMAPM program – loops and body, `updateMem` and `generateMsg` – are transformed to C+MPI.

The body and its statements can be easily translated to C. The data types `Int` and `Double` have correspondences in C, and so do arrays. The list comprehension is translated into a for loop. The `if` structures correspond directly to their APM counterparts.

```
void stmnt1(int t, int p, int n){
  double sum=0;
  int k;
  for(k=1; k<= (t/2); k++){
    sum += l[p][k]*u[k][t/2+1];
  }
  l[p][t/2+1] = a[p][t/2+1] - sum;
}

void stmnt2(int t, int p, int n){
  double sum=0;
  int k;
  for(k=1; k<= ((t-1)/2); k++){
    sum += l[(t+1)/2][k]*u[k][p];
  }
  u[(t+1)/2][p] = (a[(t+1)/2][p] - sum)/l[(t+1)/2][(t+1)/2];
}

void body(int t, int rp, int p, int n){
  if ( /*(t/2 +1 <= p) &&*/ t% 2 == 0 ) { /* L */
    stmnt1(t,p,n);
  } else /*if ((t/2 < p) && (t%2==1)) */{ /* U */
    if ((t+2)/2 != p)
      stmnt2(t+1, p, n) ;
  };
}
```

The `apm_main_loop_block` follows the structure of all APM main loops as outlined in Figure 4.8. Inside the time loop, at first all communication buffers are cleared. The virtual processor loop calls the body function and subsequently generates messages to distribute the new values. In the communication phase we store the number of messages the processor is about to send in the non-zero entries of the send-number buffer. In the global communication routine all messages are exchanged and the processors are synchronised, thereby concluding the time step.

```
void apm_main_loop_block(int n){
  int t,p,ub,lb,i;
  int rp = rank;
  for (t = 0; t < MIN( (2*n), (2*((maxp_s_mm+1)*tilesize_s_mm(n)))); t++) {
    clear_senddata();
    lb = MAX(rp*tilesize_s_mm(n)+1, (int)(t/2 +1));
    ub = MIN((rp+1)*tilesize_s_mm(n)+1,n+1);
    for(p=lb ; p < ub; p++) {
      body(t,rp,p,n);
      genmsgs_blocked(t,rp,p,n);
    }

    for(i=0; i<maxp; i++){
      if(sendnr_buf[i]>0) sendnr_buf[i] = sendmsg_cntr;
    }
    send_msgs_a2a(n, maxp,
                  sendmsg_cntr, sendnr_buf,
                  displ, sendbuf_size, rcvnr_buf,
                  psendmsg_buf, prcvmsg_buf,
                  MYCOMM);
  }
}
```

Just like the corresponding APM function, `genmsgs_blocked` generates messages to be placed in the send buffer. In the C program each message is a struct of type `Msg`, comprising an integer valued domain, the array indices and a message value of type `double`. For each generated message, the global `sendmsg_cntr` is incremented, so that at the end of each time step the number of messages to be sent is known. The ith entry of `sendnr_buf` is set to one. This means that the current message is to be sent to processor i. At the end of the time step, the complete message buffer is sent to all processors that receive at least one message. The send buffer's fill size is denoted by `sendmsg_cntr`, so this value is the length of the MPI message that the current processor sends.

```
void genmsgs_blocked(int t, int rp, int p,int n){
  int i;
  if(t%2 == 0){ /*  L   extra guard aus dep? */
       psendmsg_buf[sendmsg_cntr].msgdomain = L ;
       psendmsg_buf[sendmsg_cntr].xidx   = p ;
       psendmsg_buf[sendmsg_cntr].yidx   = t/2+1 ;
       psendmsg_buf[sendmsg_cntr].value  = l[p][t/2+1] ;
       sendmsg_cntr++;
       for(i=rank+1; i<maxp; i++){
         sendnr_buf[i]=1;  /* Deps 3,5 */
       }
  }else{ /*  U */
       psendmsg_buf[sendmsg_cntr].msgdomain = U ;
       psendmsg_buf[sendmsg_cntr].xidx   = (t+1)/2 ;
       psendmsg_buf[sendmsg_cntr].yidx   = p ;
       psendmsg_buf[sendmsg_cntr].value  = u[(t+1)/2][p] ;
       sendmsg_cntr++;
       for(i=rank+1; i<maxp; i++){
         sendnr_buf[i]=1;  /* Deps 2 */
       }
  }
}
```

The `updateMem` function receives a message struct and updates the memory accordingly. The function's structure is a direct correspondence to the APM version.

```
void updateMem_with_msg(struct Msg a){
  if (a.msgdomain==L){
    l[a.xidx][a.yidx]=a.value;
  } else if(a.msgdomain==U){
    u[a.xidx][a.yidx]=a.value;
  } else {
    printf("proc %d received unknown message domain: %d.\n",rank,a.msgdomain);
  }
}
```

6.9 Benchmarking the C+MPI Target Code

We have presented the development of four branches of abstract programs for the LU decomposition problem and given rough cost estimates based on a calibration for a ScaMPI/Linux cluster. The ultimate goal of parallel programming is to have good speedups on a parallel machine. In this section, we present benchmarking results of the final C+MPI target programs and compare them with the cost model estimates.

size	sequential time
64	0.001
128	0.008
256	0.114
512	1.522
1024	12.295
2048	98.150
4096	1237.396

Figure 6.9: Sequential C-Program LU Execution Times (in seconds)

The benchmarks are performed with all four program variations of the LU decomposition. Each of the four programs is run with problem sizes from $n = 64$ up to 4096 and on 1 to 16 processors. As run time we consider only the execution time of the `apm_main_loop`, but not initialisation and output. These times are measured with the `MPI_Wtime()` call. We have also implemented a sequential LU decomposition in C and benchmarked its executions times with the same problem sizes. The results are depicted in Figure 6.9. Therefore we can compare the multi-processor execution times with the single-processor sequential time to obtain absolute speedups. For the complete benchmark results see Figure 6.11. All times shown are rounded to an accuracy of 10^{-2}, but the values used in calculations have a precision up to 10^{-6}s.

The benchmarks reveal significant differences between the four variations. All of them yield speedups if the problem size is big enough, but the two cyclically tiled versions are certainly faster than the blocked ones. Good speedups are only achieved with problem sizes of $n \geq 512$. This behaviour is often seen with parallel programs, as with small problem sizes the amount of computational work is not big enough to outweigh the constant communication overhead.

The highest speedups and efficiency yields the cyclic STM1 program for big problem sizes. In the case of $n = 4096$, the efficiencies range from 73.5% to 44.7%, while cyclic STM2 is a close runner up with 69% to 42%. The speedups of the four programs in this case are presented in Figure 6.10.

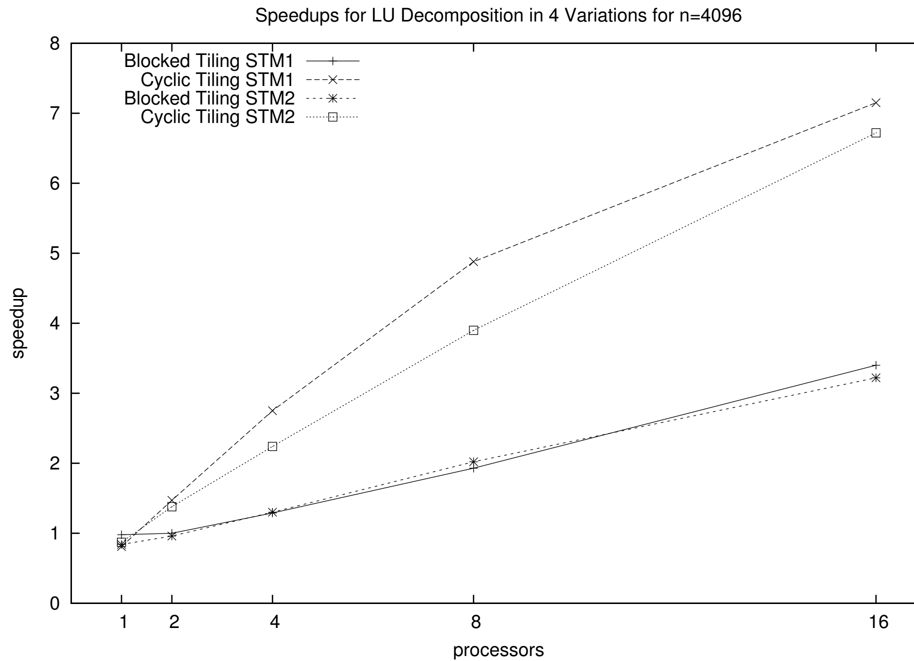


Figure 6.10: LU Speedups for $n = 4096$

At first glance we see two results: for one, the run time efficiency of the four programs is quite different, basically the cyclic programs are good and the blocked programs are less so. Secondly, the absolute efficiency of the fastest program (cyclic STM1) is quite satisfactory.

A conclusion is that the PolyAPM development process is able to produce an efficient target program in C, starting out from a sequential Haskell program. As a side note: the speedups of the final C+MPI program, when compared to the sequential Haskell specification, are quite good. As an example, take the decomposition for $n = 512$ on four processors: the C+MPI programs take between 0.88 and 1.38 seconds, the sequential C program 1.5 seconds and the sequential Haskell program with standard Haskell arrays finishes after 1 minute and 45 seconds of computation. This means that we can safely use a high level (and often comparably slow) specification language as our initial implementation without worrying that the efficiency of the target program might be negatively affected by this choice.

The cost model predictions are now to be compared with the benchmark results to find whether both comply. The largest problem size that we are able to use with the APM interpreters on the available hardware is 512. Figure 6.12 relates the cost model values of the four `SynDMAPM` programs, measured in ticks, to the benchmarked run time of the C+MPI program, measured in seconds. The purpose of the cost model is to provide a performance prediction in the form of an integer value that relates to the run time. We have stated earlier that the cost model's accuracy should not be overestimated since only few hardware properties are taken into account. Because of this, we propose the use of the model chiefly for relative comparisons. However, the expectation is that, if the prediction is accurate, the cost model values will still relate to each other in the same way the benchmark times do. In other words, the quotient of each cost model prediction and the corresponding run time should be a fixed value. We have calculated these quotients, divided by 10^6 (see Figure 6.12).

We see that in practise the quotients differ. The more similar the four quotients are, the more accurate the cost model prediction is. Unfortunately the quotients differ significantly for a given number of processors. While in all three cases the blocked programs are correctly considered more expensive than the cyclic ones, the order among the blocked and cyclic programs is often wrong.

	1 processor		2 processors		4 processors		8 processors		16 processors	
STM1 blocked tiling										
size	time	speedup	time	speedup	time	speedup	time	speedup	time	speedup
64	0.001	0.73	0.01	0.08	0.08	0.01	0.25	0.00	0.53	0.00
128	0.008	0.95	0.02	0.39	0.09	0.09	0.27	0.03	0.59	0.01
256	0.116	0.99	0.13	0.85	0.17	0.67	0.31	0.37	0.70	0.16
512	1.555	0.98	1.42	1.07	1.07	1.42	0.90	1.69	1.14	1.33
1024	12.386	0.99	11.09	1.11	8.08	1.52	6.72	1.83	8.04	1.53
2048	97.908	1.00	86.57	1.13	64.57	1.52	35.45	2.77	27.13	3.62
4096	1265.065	0.98	1235.29	1.00	961.91	1.29	640.04	1.93	363.92	3.40
cyclic tiling										
size	time	speedup	time	speedup	time	speedup	time	speedup	time	speedup
64	0.001	0.72	0.01	0.07	0.07	0.01	0.25	0.00	0.55	0.00
128	0.008	0.94	0.02	0.36	0.09	0.09	0.30	0.03	0.57	0.01
256	0.117	0.97	0.11	1.06	0.18	0.63	0.33	0.34	0.74	0.15
512	1.555	0.98	1.12	1.36	0.88	1.74	0.82	1.85	1.18	1.28
1024	12.384	0.99	8.44	1.46	6.01	2.05	5.20	2.36	7.87	1.56
2048	106.067	0.93	65.48	1.50	40.13	2.45	22.82	4.30	17.73	5.53
4096	1529.823	0.81	843.82	1.47	449.51	2.75	253.53	4.88	173.08	7.15
STM2 blocked tiling										
size	time	speedup	time	speedup	time	speedup	time	speedup	time	speedup
64	0.002	0.61	0.02	0.07	0.07	0.01	0.25	0.00	0.49	0.00
128	0.010	0.82	0.02	0.34	0.08	0.10	0.34	0.02	0.58	0.01
256	0.218	0.52	0.19	0.61	0.19	0.61	0.31	0.37	0.74	0.15
512	2.189	0.70	1.83	0.83	1.38	1.11	1.11	1.37	1.24	1.22
1024	17.303	0.71	14.31	0.86	10.51	1.17	8.24	1.49	8.93	1.38
2048	136.367	0.72	111.58	0.88	76.86	1.28	47.76	2.06	30.46	3.22
4096	1466.411	0.84	1285.57	0.96	953.60	1.30	611.62	2.02	384.63	3.22
cyclic tiling										
size	time	speedup	time	speedup	time	speedup	time	speedup	time	speedup
64	0.001	0.85	0.02	0.07	0.08	0.01	0.26	0.00	0.53	0.00
128	0.007	1.07	0.02	0.38	0.08	0.10	0.27	0.03	0.58	0.01
256	0.209	0.55	0.16	0.71	0.19	0.61	0.31	0.36	0.68	0.17
512	2.107	0.72	1.39	1.09	1.11	1.37	0.91	1.67	1.14	1.33
1024	16.894	0.73	10.68	1.15	8.08	1.52	6.31	1.95	8.24	1.49
2048	133.784	0.73	84.86	1.16	57.95	1.69	31.36	3.13	21.42	4.58
4096	1422.203	0.87	895.04	1.38	551.95	2.24	317.14	3.90	184.22	6.72

Figure 6.11: LU ScaMPI Benchmarks (time in seconds)

	2 processors			4 processors			8 processors		
	CM	BM	quot.	CM	BM	quot.	CM	BM	quot.
B1	$11.98 \cdot 10^7$	1.42	84.59	$8.01 \cdot 10^7$	1.07	74.97	$4.77 \cdot 10^7$	0.90	52.82
C1	$6.93 \cdot 10^7$	1.12	61.76	$3.65 \cdot 10^7$	0.88	41.64	$2.10 \cdot 10^7$	0.82	25.51
B2	$11.33 \cdot 10^7$	1.83	61.78	$7.80 \cdot 10^7$	1.38	56.64	$4.73 \cdot 10^7$	1.11	42.58
C2	$7.04 \cdot 10^7$	1.39	50.55	$3.65 \cdot 10^7$	1.11	32.74	$2.05 \cdot 10^7$	0.91	22.45

Figure 6.12: Cost Model and Benchmark Comparison for $n = 512$

While the big picture is correct, on a smaller scale the results are not, so that they should not be relied upon.

The reason for this discrepancy may be the sum of several causes:

- The cost model calibration was done for a large number of messages and the problem sizes used for the APM predictions might not have been big enough.
- Even though the benchmarked times are averages of 10 runs, the absolute run times are very small (usually less than a second), so that other processes' interference on the Linux cluster nodes becomes significant.
- Due to the small problem size, buffers and caches might not or only sometimes be full. Such effects are ignored by the cost model.

Unfortunately the available hardware restricted LU APM simulations to a problem size of $n \leq 512$. However, the four programs' cost model predictions do not change too much in relation to each other if we increase the problem size. Therefore we can expect that the absolute cost model values for a problem size of $n = 4096$ are higher, but their relations will be similar to the 512 case. Based on this argument, in Figure 6.13 we compare the cost model values from the problem size $n = 512$ to benchmark values for $n = 4096$.

	2 processors			4 processors			8 processors		
	CM	BM	factor	CM	BM	factor	CM	BM	factor
B1	$11.98 \cdot 10^7$	1235.29	0.097	$8.01 \cdot 10^7$	961.91	0.083	$4.77 \cdot 10^7$	640.04	0.075
C1	$6.93 \cdot 10^7$	843.82	0.082	$3.65 \cdot 10^7$	449.51	0.081	$2.10 \cdot 10^7$	253.53	0.083
B2	$11.33 \cdot 10^7$	1285.57	0.088	$7.80 \cdot 10^7$	953.60	0.082	$4.73 \cdot 10^7$	611.62	0.077
C2	$7.04 \cdot 10^7$	895.04	0.079	$3.65 \cdot 10^7$	551.95	0.066	$2.05 \cdot 10^7$	317.14	0.065

Figure 6.13: Cost Model for $n = 512$ and Benchmark for $n = 4096$ Comparison

In this case we observe a surprisingly good match. As the problem size is significantly bigger and the benchmark run time longer, all of the three reasons mentioned above for the discrepancy do not apply anymore. Thus we have a situation where the cost model predictions – quite independently from the simulated problem size – match large problem sizes on the target machine. As parallel programs only deal with large problem sizes, this fact is welcome.

In this case study we have shown that some properties of the final parallel program can be observed in the intermediate APM programs.

Chapter 7

Evaluation of PolyAPM Based on the Case Studies

The initial presentation of PolyAPM in Chapter 3 outlines a programming approach that needs to prove its usefulness in practise. We now present our experiences based on the case studies in Chapters 5 and 6 as well as our general experiences with using the implementation, and subject them to a critical evaluation. We divide the presentation of our experiences into more general remarks, that refer to the PolyAPM approach itself and that are not directly related to the current implementation, and into more specific remarks that deal with our set of Haskell interpreters and the `mpi_apm` library.

7.1 Experiences with the General PolyAPM Approach

Although all of our experiences relate to the practical work we did in the case studies, some of them are of a more general nature and do not refer to specific implementation properties. It should be noted that our example implementation of APM interpreters and the `mpi_apm` library are replaceable. All the ideas that we have described in Chapter 3 are independent of our specific implementation. The following remarks refer to the PolyAPM ideas themselves.

Separating concerns using multiple transformations worked. The transformations were in fact simple enough for manual coding. In Chapters 5 and 6 we go through sequences of transformations of APM programs. Each change is explained in detail, in particular all parts of the APM program that are affected by the transformation are mentioned. The list of changes is usually short. A student who has a background in parallel programming was able to write APM programs only after a short introduction. This suggests that the main challenge in writing APM programs is dealing with the transformations themselves, not with implementing their result as APM programs.

Many functions comprising APM programs can be reused. The additional effort of the exploration of alternative transformation paths scales sub-linearly. In addition to saving work, the reuse of functions structures the program generation. We get more insight into how different branches of the PDG relate than if we just produce variations of the final target code.

To demonstrate how frequent the reuse is, Figure 7.1 contains a table with all APM functions of the various APM programs from the LU case study in Chapter 6. A function name in roman letters means that the function is defined in that particular program, while a name in italics means that a function defined elsewhere is reused. An arrow leads to the reused

function. A dashed arrow denotes a function reuse within a small wrapper function, so there is a little bit of additional code. A total of 44 functions is listed in the table, and 24 functions reuse other implementations.

	STM 1		**STM 2**	
SynAPM	loops body		loops body	
	block tiled	**cyclic tiled**	**block tiled**	**cyclic tiled**
SynAPM (tiled)	loops body	loops *body*	loops body	loops *body*
SynCommAPM	*loops* body updateMem generateMsg	*loops* *body* *updateMem* generateMsg	*loops* body *updateMem* generateMsg	*loops* *body* *updateMem* generateMsg
SynDMAPM	*loops* body updateMem *generateMsg*	*loops* *body* *updateMem* *generateMsg*	*loops* body *updateMem* *generateMsg*	*loops* *body* *updateMem* *generateMsg*

Figure 7.1: Code Reuse in the LU Case Study

The initial effort of manual APM programming may be substantial. First off, if our provided set of synchronous APMs does not meet the requirements, one has to implement new machine interpreters, one for each needed APM.

In addition, to derive even a single target program using PolyAPM, several intermediate APM programs have to be written. In the case of LU, we have one declarative specification, followed by a sequence of four APM programs before writing the target program. The traditional approach in parallel programming is to write the target program directly. This additional effort is pays off if several target programs are derived and alternative transformations are pursued, but for the generation of a single target program it probably does not.

Significant decision support by the cost model. The LU case study shows that the cost model serves the purpose of providing support for the decision between several transformations. Sometimes its results are quite accurate, but in general one cannot completely rely on them. We have found the error margin in some cases to be a factor of 2, so that we cannot use the cost function as a clear prediction of target code efficiency. However, we use it in PolyAPM primarily for relative comparisons of transformations, so that absolute accuracy is not really needed. For this purpose the cost model is well suited.

As we show in both case studies, the predictions are more accurate the larger the problem size is. Reasons for the inaccuracy of the model are discussed in detail in the next section.

From the programmer's point of view, the prediction's accuracy depends on the precise provision of computation and communication cost in the APM programs. In the LU case

study, we find that the computation cost for generating messages does not contribute much to the final cost function result, so one might argue to omit it. However, with the finite difference program, message generation is comparably costly and must not be left out. The problem is finding the right level of detail for the costs. But even with foremost care there remains some inherent inaccuracy due to simplicity of the model. This will also be discussed in more detail below.

7.2 Experiences with our Implementation

Our APM infrastructure as depicted in Chapter 4 leads to observations that we discuss now. These remarks are rather implementation specific and do not relate to general PolyAPM properties.

Advantages of using Haskell. There are many advantages of using Haskell for the APM interpreters. The syntax is concise and programs tend to be quite short. This enables the display of significant parts of the work in papers, while the gist of the programs can be grasped by readers not familiar with Haskell. The advanced type system helps to embed the APM programs within a Haskell program. The APM program can be type checked and is accessible as a data item to other Haskell functions, such as the APM interpreters. In Chapter 3, we motivate the use of Haskell for writing the source programs. In that case, the computations can be transferred directly into the APM program without a rewrite in a different language.

A specific feature of the type system that enables the separation of APM interpreters and programs is described further below.

The combined interpreter/compiler of the Glasgow Haskell Compiler [GHC] provides an excellent development platform. The interpreter is able to load precompiled modules automatically for better run time efficiency, but retreats to interpretation if the object file is not recent. This combines the advantages of interpreters and compilers.

Profiling information provides insight into the algorithm's behaviour. The APM interpreters provide more profiling data than just the mere cost function value. The `SynAPM` displays the total computation cost over all processors. This is equivalent to the total computational work, and therefore the workload of a sequential program. The SynCommAPM and SynDMAPM programs output details of the communication: the sum of sent and received messages per processor. Together with the workload, these values can be used to identify computation and communication hot spots and, therefore, the possible need for a load balancing scheme. In the LU case study, we can observe that the cyclic programs are much more balanced than the blocked ones.

The cost model results must be interpreted in the right context. We argue in the previous section that the inaccuracy of the cost model is no impediment, since the results are sufficient for our purpose and are actually of surprising quality considering the model's simplicity. As long as the results are seen in this context, the use of the model is of value. We now want to provide reasons for the inaccuracy.

- The cost model assumes that the performance of the machine scales linearly with the problem size. This linearity is a simplification. There are many buffers and caches in contemporary machines. The performance of accesses through buffers and caches takes a leap whenever certain thresholds are reached, such as a buffer being full. These leaps contribute to non-linear behaviour, especially if many such effects superimpose. Other sources of non-linearity include the non-exclusive use of the machine and reaching the bandwidth of internal buses and external communication infrastructure. In short, every cost model short of a complete hardware simulation will exhibit inaccuracies in the prediction.

- The cost model is less accurate for small problem sizes, but more accurate for large ones. As for the communication costs, we argue in Chapter 6 that we have calibrated the message costs for a large number of PolyAPM messages. With bigger problem sizes we usually get more messages, so that the predictions will be more accurate. The computation costs were measured under high load. We have performed many computations in a loop, taking care that at least one operand was not in the processor cache, so that a memory access was needed. The factors which determine the efficiency of such a computation are the ratio of cache misses and the percentage of memory bandwidth being used. Again, our calibration setup is geared towards a high load situation.

APM program syntax with pros and cons. A design goal for APM programs was to achieve executability while retaining human readability. This was made possible through embedding the APM programs as algebraic data types into the language. The programs are interpretable, but the concise syntax is accessible to a reader with some familiarity with Haskell. This way, one even gets type checking of the APM programs for free.

But we learnt from feedback that some parts of an APM program are not easy to understand for readers not familiar with Haskell. Most notably, the loop bounds that are represented as lambda-abstractions may be difficult to grasp. While readability is always debatable, we expect that the current APM programs, that are embedded in Haskell, will always require some basic knowledge of the language. There is room for future work to find an alternative, more accessible representation for APM programs and means to convert different representations into each other.

Additionally, any automatic transformation would need to parse such a program, apply the transformation to the parse tree, and generate again an APM program. Similarly to the point above, this issue is about a more machine accessible representation. Ideally, there would exist all three representations and conversion functions between them.

Haskell's laziness complicates debugging of APM programs. While laziness undoubtably is a powerful evaluation strategy, we found laziness a drawback when debugging programs with array accesses. Lazy evaluation means that expressions are evaluated as far as needed, which in practise means that an expression in the course of an program execution is at first only partially evaluated, maybe later on further evaluated, before finally a value is obtained. Over time a lot of partially evaluated computations build up. Driven by the need to output results, eventually the partially evaluated expressions are evaluated to completion.

When, due to a programming error, an undefined array element is referenced and the programs aborts with a run time error, the expression which contains the erroneous reference cannot be easily identified. This is because the evaluation order of all expressions and subexpressions is difficult to follow. Haskell provides the `trace` function to output debug information before evaluating an expression. But the output of several `trace` calls may occur in surprising order. Strict programs have a clearer connection between the flow of the program's syntax and the control flow. Thus, debug output helps better to locate an erroneous expression.

The available options in Haskell include: forcing strictness with the `seq` operator, creating debug output with the `trace` function, and more involved debugging with tracers (most notably HAT [HAT]). But we found `seq` to be only of limited use because it forces only a weak head normal form and does not completely evaluate the expression immediately. The complex evaluation order makes the tracing through expressions very difficult. In addition, the tracers do not support all extensions of the ghc compiler that we use in our implementation.

Efficiency problems using APM interpreters. The delayed complete evaluation of expressions due to lazy evaluation is intended to save computations. If a computation is delayed until it is needed, it may not be done at all. The problem is that the intermediate, partially evaluated expression consumes significant amounts of memory. If the result of an expression

is needed anyway, it would be more space efficient to evaluate it completely right away. Again, strictness annotations can only reduce the inefficiency problem, but not solve it.

Another source of inefficiency are the updates of standard Haskell arrays. This problem was solved using more efficient, linear arrays.

However, even with linear arrays the space inefficiency restricts the maximal problem size significantly. The LU decomposition program with an input size of 512×512 gets dangerously close to the 2 GB process size limit of a 32-bit operating system. Larger problem sizes cannot be dealt with.

To increase the space efficiency, we see two options for the future: either the interpreters are rewritten in Haskell using the state transformer monad that can force evaluations immediately and employs in-place array updates, or the interpreters are written in a strict functional language like OCaml [OCa]. The second option restricts the generality as only those computations can be allowed in the Haskell specification that are easy to translate to OCaml as a part of an APM program.

Haskell enables tight relation of interpreters and programs. The APM interpreters are part of a library that a Haskell module containing APM programs imports. The APM programs, comprising the loops and a body function, are passed to the interpreter. The interpreters also need to call other functions of the APM program, such as memory update and message generation. They need to match the loop nest. The link, together with the data types that comprise a message, is the common memory type that they all operate on. The interpreter library needs to be able to call the correct, say, memory update function to a given loop program. In order for this to be type safe, it must be ensured that the update function is written for the same memory and message types.

Haskell's type classes solve the problem for just one type. The Glasgow Haskell Compiler has an extension for *multi-parameter type classes* [WB89, PJJM97]. They help to solve the problem nicely. Every APM program provides the necessary instance declarations of such classes. In these instances the memory update functions are defined (see Section 4.1). The type classes attach these functions to the memory type and export them globally, so that the APM interpreters automatically call the correct ones.

A solution to inefficient array accesses. The inefficiency of array updates in non-strict, purely functional languages has been the source of many research papers (e.g., [EL97]). Conceptually, every update results in a new array which differs only in the updated element from the old one, all other elements are copied. With Haskell arrays, this copying takes place with each update. Old arrays are eventually garbage collected, but often not immediately.

In many cases, one would prefer to have in-place updates without the space and time overhead of an additional array copy. In general, this is not possible with Haskell arrays. Even after the update, references to the old array exist and may still be used for some time, thus the need for the copy. However, APM programs are imperative by nature, so that old arrays are never referenced after an update. The arrays are used linearly, so that the update could be made in place, without copying the whole array. Haskell does provide such a kind of arrays in the state monad, but this means to convert the entire interpreter into monadic style.

We use an alternative array implementation named `LArray` in Haskell that implements linear arrays (see Section 2.1.2). Here, the user must ensure that the arrays are only used linearly, i.e., old versions of the array are never referenced, otherwise the result might be wrong. An important property of this implementation is that it overloads the syntax of standard Haskell arrays, so that the interpreters need not be changed. After switching from standard Haskell arrays to `LArray`s, the APM interpreters use significantly less memory and run time.

Good speedups using `mpi_apm`. Our implementation uses our collective communication library `mpi_apm` (Section 4.3.1) to ease the transition from the SynDMAPM program to a target program in C+MPI. The communication structure of the APM program is based on one-sided

calls performed by the sender. The MPI-2 standard defines one-sided calls, but the ScaMPI implementation on our cluster lacks their implementation. The collective communications are chosen because we can mimic easy-to-program one-sided calls with them.

Most programs do not exhibit an alltoall communication pattern. When compared to traditional send/receive communication, a collective call in such programs sends many superfluous messages. However, the detailed LU benchmarks show that nevertheless good speedups were obtained. The efficiency exceeds 50% in some cases, which is a good result for a communication intensive program.

The explanation is that `MPI_All_to_All` is very fast on the Scali cluster. Since it is simpler to use than a collection of send/receive pairs, one should seriously consider this programming style with synchronous programs on low latency cluster hardware.

7.3 Evaluation Summary

The purpose of the PolyAPM approach is outlined in Chapter 3. Based on the above experiences, we now summarise our findings.

- We provide a framework in which an end user can implement an algorithmic specification as a non-parallel, declarative program. The most important aspect is the ease of programming, while the drawback may be unsatisfactory sequential run time and space efficiency. This declarative program can then be transformed into an APM program and be subjected to several code transformations until a target program is generated. Such a target program runs efficiently on a parallel machine, but would be difficult to write directly.
- The programming process is structured into a sequence of source-to-source transformations. Input and output of the transformations are programs for abstract parallel machines. These intermediate programs can be accessed, evaluated and even executed using APM interpreters.
- The PolyAPM framework is flexible enough to cater for manual, semi-automatic and automatic parallelisation, depending on the number of program transformations that are automated.
- There are several options for decision support when selecting one of multiple alternative transformations. The options are manual code inspection of the APM programs, run time behaviour observation using the APM interpreters with real input, and a cost model analysis targeting a real machine.

Chapter 8

Related Work

There exists work in several different areas that is related to PolyAPM. First we provide an overview of different ways to write a parallel program manually. The focus is laid upon standardised libraries for imperative languages and several approaches in parallel functional programming. The next section deals with compilation systems which can be subdivided into parallel compilers and more academic program generation frameworks. Finally we discuss other work on abstract parallel machines.

8.1 Manual Parallel Programming

Originally, parallel computers were programmed manually. There was no compiler support whatsoever and the distribution of the computation across different processors as well as the communication between the processors had to be done explicitly. Usually, the manufacturer of the parallel machine provides proprietary libraries to access the machine's features efficiently. While this happens until today, for a long time such programs are known not to be portable to other parallel machines and the programming interfaces are often difficult to deal with and change frequently. Thus, there is a need for more hardware abstraction to ease programming and to provide portability.

8.1.1 Standardised Libraries for Parallel Programming

The need for standardised and portable programming produced two major message passing libraries: PVM (Parallel Virtual Machine) [G$^+$94] and MPI (Message Passing Interface) [DHHW93]. The decline of the use of PVM in recent years leaves MPI as the de facto standard. MPI is designed as an SPMD communication library for distributed memory machines. The standard defines an API for three kinds of communication: sends/receives, collective communication and remote memory access. The most common language bindings are for C and Fortran. The MPI implementations try to map the communication functions efficiently on given hardware, a task in which proprietary implementations of MPI succeed than portable ones. However, all of these libraries have their drawbacks: explicit communications are error prone, and programming with over 150 MPI functions is a complex task.

Based on these experiences, the BSP (Bulk Synchronous Programming) [SHM97] library was designed with two main differences in mind: a much smaller API (around 50 functions) and a simpler communication mechanism. A BSP program consists of a series of *supersteps*, each starting with a computation phase and ending with a synchronised, global communication. This programming style provides less flexibility than MPI in that the communication pattern is fixed. BSP received particular attention in the academic world.

Another library, OpenMP [DM98], is a multi-vendor effort to provide a standard for SMP programming. It consists of compiler directives and a corresponding library for C/C++ and Fortran. The directives are provided as special key words in the program's comments, so that an OpenMP program can be compiled with a sequential compiler without any change.

8.1.2 Parallel Functional Programming

In the 1980s, parallel programming with functional languages was thought to have a bright future. Due to their different semantics compared to imperative languages, it is easy to identify independent expressions that can be evaluated in parallel. In fact, most functional programs are *embarrassingly parallel*. However, the early hopes were not fulfilled, mainly for two reasons. For one, most functional languages require a big run time system that adds quite some overhead in contrast to imperative programs. The second reason is that it turned out to be the main challenge to *contain* the parallelism, not to find it.

Sisal [BOCF92] is a single-assignment, first-order, monomorphic functional language designed for scientific computation. Its syntax resembles Fortran. As it lacks a lot of advanced concepts of modern functional languages, it could be implemented very efficiently.

Most existing functional languages received extensions to incorporate explicit parallelism. As an example, the team of the Glasgow Haskell Compiler [GHC] developed the *Glasgow Parallel Haskell* specification, whose only implementation is GUM [THM+96]. It works by inserting the infix operators `seq` and `par` into the program to denote expressions that have to be evaluated sequentially (the first argument's evaluation is forced) or may be done in parallel (which is a hint to the run time system to create a separate thread). The difficulty with this approach is that you only give hints to the run time system concerning parallel execution, but the run time system may choose to ignore the hints, while it incurs quite some overhead at the same time.

Eden [BLOMP98] is an extension of GHC supporting parallel programming by explicit process creation and communication between processes with uni-directional channels. The channels behave like lazy lists and model a stream based data exchange. Sending and receiving is done by manipulating the channel lists. Lower level communication management is not necessary. Eden enforces strict evaluation in places where laziness would restrict parallelism. Explicit machine control like process placement is not possible.

Another approach is to find a common execution pattern in a set of algorithms, then to implement this pattern efficiently as a parallel program, and subsequently to instantiate this pattern with the specifics of a required algorithm. For all other algorithms later on, only the instantiation has to be done again. These patterns are called *skeletons* [Col89]. A skeleton is defined as a program that takes functions as parameters to be instantiated for a specific algorithm. The program then consists of the skeleton together with the instantiating functions. If specified in a functional language, skeletons are implemented as higher-order functions. There exist compilation systems that transform skeleton programs into target code suitable for a parallel machine [Her01]. Others perform the implementation manually, but in a systematic way [Gor98].

The aim of the skeletons community is to have a library of skeletons, each providing an efficient implementation for a set of algorithms. Thus, a programmer may choose the right skeleton and instantiate it, yielding an efficient parallel implementation. It is possible to combine skeletons to form larger programs.

Aldinucci et al. presented the *Functional Abstract Notation* [AGLP01], which implements skeletons based on higher-order functions like `map`, `reduce`, etc. as Haskell programs. His META tool [Ald02] extends his work with a graphical environment which lets the user select code transformations. A simple cost model associates the transformations with a performance prediction. Sequences of such transformations yield a final target program.

8.2 Compilation Systems

Systems to compile programs for parallel machines fall into two categories: production grade compilers, often for general purpose languages with some support for parallel execution, and research systems with the focus on understanding the parallelisation rather than on producing code.

8.2.1 Commercial and Academic Compilers

Most compilers focus on the generation of efficient SPMD programs from source languages like Fortran. The source programs usually have to be augmented with parallelisation and data distribution annotations. However, some systems feature "automatic parallelisation" switches that enable either simple or semi-automatic parallelisation schemes. This group includes among others Adaptor [Bra98], Polaris [BEF+95] and Parafrase [PGH+90]. A special role plays Bert77 [Ber02], a Fortran77 source-to-source compiler that employs both static and dynamic parallelisation schemes, and focuses on performance prediction. A graphical user interface guides the semi-automatic process to improve the parallel performance based on a machine dependent cost model.

The SUIF [WFW+94] system serves as a compiler's workbench; the SUIF kernel defines an "intermediate representation" of a program between compiler phases and provides functions to access and manipulate it. SUIF is distributed with a set of example phases which includes a data dependency analysis and simple parallelisation techniques. However, parallelisation is not the foremost goal of the SUIF project.

In any case, each compiler has been designed with a rather fixed compilation process in mind. The compilation phases usually can be influenced by run time options, but more flexibility is rare. The most flexible parallelising compiler appears to be Bert77, which is claimed to be able to choose automatically between three different parallelisation schemes.

8.2.2 Parallelisation Systems in Research

Automatic parallelisation systems still remain mostly a research topic. They are very specialised and work only on a restricted set of input programs. On this set, they usually provide an effective detection of parallelism. However, as they rely on a particular parallelisation method, their selection and ordering of transformations is mostly fixed. In this respect, they are more restricted than PolyAPM. However, they are often workbenches for the transformation development of researchers, so that for their domain these systems provide the state of the art.

PAF [FCB+98] is an automatic paralleliser for a restricted class of Fortran program based on the *polytope model* [Len93]. The LooPo [GL96] system extends this to a variety of input and output languages. It implements several alternative algorithms for space-time mapping and code generation. It can deal with more general programs than PAF. PIPS [ACC+96] is a compiler's workbench with many different transformations. It works on Fortran code and aims primarily at the signal processing domain.

OPERA [LM96] was an academic system designed for the parallelisation of affine recurrence relations. It consists of a dependence analysis, scheduler and allocator as well as a tool to visualise the index spaces.

8.3 Abstract Machine Models

This section is about the closest work of all that are presented here. They are program transformation systems targeting abstract machines.

John O'Donnell and Gudula Rünger have presented *Abstract Parallel Machines* (APMs) [OR97], providing a starting point for others to work on parallel compilation using an abstract machine approach. APMs are defined as simple distributed memory machines without buffers and caches, connected with a network of unspecified topology. Semantically, APMs are defined by their input/output characteristics. An APM program consists of one or several *parallel operations* (ParOps), each of which are designed for small tasks. Chapter 3 has more details.

Joy Goodman extended the above work [Goo01], included input and output via monads and investigated and formalised the decision making process. She focused on formalising the transformations. Her APM programs are Haskell programs of a certain form so that there is no need for APM interpreters. The cost of the programs is manually derived by inspecting the algorithmic behaviour of the APM programs, but no formal cost model is defined.

Noel Winstanley also uses the APM methodology in his PEDL system [Win01]. He compiles array-based numerical programs to the parallel, imperative target language SAC. However, he uses a special, restricted source language, tailored for his specific problem domain, and focuses on a high degree of optimisation and automation of the compilation.

Our work differs from the above in several respects: PolyAPM defines loop programs for APMs that are embedded in Haskell, but need interpreters to be run. These interpreters are used to observe the behaviour of the APM program on real input data and they provide profiling data of these executions. We provide a cost model that uses this profiling data to make a performance prediction for a chosen machine. The APM programs as defined in PolyAPM are more general than Winstanley's so that a greater application domain is applicable. PolyAPM defines no fixed target language, so our approach is more flexible. We address the parallel program generation for scientific algorithms with special focus on the comparison and selection of alternative transformations.

8.4 Compiler Generation

Although the construction of a complete compiler by automating all transformations was no goal set for the current work, it is an area of possible future application. We will therefore briefly describe some significant advances.

Most tools supporting compiler construction are designed to generate scanners and parsers. Tools for code transformations, be it on source-to-source level or on abstract syntax trees, are quite rare.

The Synthesizer Generator [RT89] is a system to automate the creation of language-aware editors. Its editors use attributed grammars as their internal representation and are able to perform rule-based source-to-source transformations on the edited programs. With a suitable set of formalised transformations, one could build an editor that can perform these transformations directly on the code. Such an editor can be seen as a user-guided transformation tool.

Baxter et al. present the Design Maintenance System [BP97]. It is a transformational development process focusing on incrementally adapting the design. From each design an implementation is to be derived automatically. Some program generation tools have been written, but the vision of DMS has not yet been accomplished.

Chapter 9

Conclusions

PolyAPM is a framework in which code transformation techniques for a parallelising compiler can be implemented and evaluated. For a given problem, one can explore alternative transformations and determine which selection of transformations is best suited. It is even conceivable to make statements about which one of a set of machines seems to be best suited for a given class of problems.

9.1 Summary

We have presented an approach for the systematic development of parallel programs by applying a sequence of source-to-source transformations, which provides for a demand-driven selection process of transformation techniques as well as means of evaluating and profiling the intermediate representation.

9.1.1 Main Contributions

The following lists briefly the main contributions of this work.

- We have presented PolyAPM, a flexible framework for manual, semi-automatic and automatic parallelisation. Its main features are:
  - Organisation of the compilation process into a sequence of source-to-source transformations, some with alternatives. The transformations construct a directed graph, usually a tree.
  - Source and target of each transformation are abstract programs that can be manually written, inspected and executed with real input on abstract machine interpreters.
  - Effects of transformations can be immediately observed, evaluated and compared.
  - Investigation of alternatives is done by navigating through the program graph. Alternative transformations create independent sub-graphs. If a particular path that is chosen later turns out not to be optimal, the program generation only needs to backtrack to the point where the decision for this path was taken. Then, one of the alternative paths may be pursued. Thus, one backtracks along the program graph only as far as necessary. New programs need not always be compiled or generated from the initial specification. This saves significant work if several program alternatives are manually derived.

  - We provide a cost model that predicts the run time efficiency of target programs on target machines evolving from a given abstract program. This cost model can be used independently from PolyAPM.

- We have performed two case studies which validated the framework.

- We have shown that, on a low latency communication cluster, it is very efficient to use collective communications. We claim that it is also easier opposed to programming with send/receive pairs.

- We have shown that implementing numerical algorithms in a high-level, declarative programming language is easy and does not impose superfluous data dependences by an unnecessary sequencing of computations. Writing a functional program before transforming it into an APM program is therefore beneficial for the programming and parallelisation process.

9.1.2 Main Features of the PolyAPM Framework

The main contribution, the PolyAPM framework, has several key properties that distinguish it from available alternatives, such as traditional parallelising compilers. These properties are now highlighted in more detail.

Evaluation of transformation effects: In contrast to classic compilation systems, one can easily observe the effects of a transformation in PolyAPM by looking at the APM program code, as well as by retrieving simple profiling information from the interpreter. This is achieved by running an APM program with different run time options and inputs using the interpreter for the corresponding APM and by comparing the interpreter's cost model predictions. Especially with a long sequence of transformations, it is often difficult to deduce from a final sub-optimal compilation result which particular transformation has had the negative effect. PolyAPM enables the user to evaluate each transformation step as if looking inside a compiler.

Well-founded selection of transformations: Based on the fact that we can evaluate transformations, we can decide against bad ones and go for a different transformation path. In case of a manual compilation process, a lot of unnecessary work can be avoided by cutting off the unwanted branch.

Furthermore, the modularity and extensibility of PolyAPM enables us to provide alternatives for transformations where traditional monolithic compilers just provide one fixed, built-in transformation. A selection may be based on different criteria: the problem domain, the characteristic properties of available parallel machine(s) (possibly of importance are, among others, the number of processors, network topology, memory hierarchy), the properties of a preferred message passing library, and so on. In addition, if the right choice is not evident, one has the opportunity to continue with a breadth-first search style of programming by selecting, transforming and evaluating a program, possibly followed by a backtracking step if the evaluation was not satisfactory.

A problem arises when two (or more) transformations at different levels of the PDG interact in such a way that the choice of the latter influences the validity of the evaluation of the former. In this case, a design decision cannot be based solely on the evaluation of the first transformation. Even if these interactions are not known beforehand, one has to be aware that they might exist, as rare as they may be.

Step-by-step automation of the program generation towards a compiler: We have applied our PolyAPM transformations manually. But the system is designed to keep the changes to the program introduced by one transformation as small as possible. One reason for this is that they can then be easier performed automatically. This will lead to a potentially large, but finite number of automatic transformations. If all transformations along one path in the

derivation tree are automatic, we have created a compiler. But, also a mix of manual and automatic transformations is possible, which allows for a stepwise development of a compiler. For special transformations an automatic solution might be too difficult and rarely needed, so that one is satisfied with the manual solution, rendering the entire system semi-automatic.

9.2 Who Should Use PolyAPM?

Due to its generality, the presented approach is suitable for a variety of different users. The following is a short list of potential users in different scenarios.

Developers of compilation techniques for parallel programming, especially if the infrastructure for a complete compiler is missing. For example, if a specific tiling technique is being developed, an evaluation of its effects is difficult without a compiler infrastructure. Incorporating the technique into an existing compiler may be difficult, and writing a new and simple compiler may be too much work for a research project. PolyAPM can be used to deal with a transformation manually without the overhead of an existing compiler.

Programmers who need to write many variations of a parallel program and/or are targeting many different hardware platforms. Variations also include consecutive improvements of a target code program. The use of PolyAPM can save work and provide a better understanding for the effects of the transformations.

Experienced programmers and researchers who need manual to semi-automatic compilation, e.g., in order to combine some automated standard program transformations with their expertise in cases where automation is not feasible.

9.3 Outlook

The following lists proposals for future work extending or building upon PolyAPM. The experiences with the current prototype motivate these extensions.

- Use additional formats to represent APM programs. The current Haskell-embedded programs are suitable for interpretation, but less so for human reading and automatic program transformation (see Section 7.2). Two additional formats might be sensible: a data structure containing the parse tree of the program, so that automated program transformations are easier to implement, and a more readable textual representation with less syntactic overhead. Automatic conversions between the three formats should be provided.
- Explore the cost model on a variety of different hardware platforms. Develop a process where the selection of the most suitable hardware platform for a given problem is possible. With all due caution regarding the sufficient accuracy of the cost model, such predetermination of suitable target hardware matches the requirements of Grid computing.
- Re-implement the APM interpreters in either completely monadic style or in a strict programming language. The deferred evaluation of Haskell yields space-inefficient interpreters.
- Implement more APMs and transformations, such as an asynchronous line of interpreters.
- Automate some or all transformations: a case study could show that an iterative compiler construction is feasible. This could be done by automating some transformations and providing tool support to connect manual and automatic transformation processes.
- The correctness of transformations can be ensured by using equational reasoning on the Haskell APM programs. A further case study could analyse whether a proof is worth the effort.

Appendix A

LooPo Specifications

A.1 LooPo Specification for STM1

```
// first STM for LU
CONSTANT n
LOOPS i,j(i),r,q(r)
STATS l(i,j),u(r,q)
PSEUDOS k,kk

i: 1<=i, i<=n
j: 1<=j, j<=i

// q=j, r=i fuer u(i,j)
// index space for u is enumerated reversely
q: 2<=q, q<=n
r: 1<=r, r<=q-1

k: 1<=k, k<=n
kk: 1<=kk, kk<=n

STAT l :  L[i,j] := A[i,j] - SUM(L[i,k0]*U[k0,j])
STAT u :  U[r,q] := A[r,q] - SUM(L[r,kk0]*U[kk0,q])

DEP(): l(i,k) -> l(i,j) : j<=i, 1<=k, k<=j-1
DEP(): u(k,j) -> l(i,j) : j<=i, 1<=k, k<=j-1
DEP(): u(kk,q) -> u(r,q) : r<=q-1, 1<=kk, kk<=r-1, q>=2
DEP(): l(r,kk) -> u(r,q) : r<=q-1, 1<=kk, kk<=r-1, q>=2
DEP(): l(r,r) -> u(r,q) : r<=q-1, 1<=kk, kk<=r-1, q>=2

//result:
//schedule for statement 1: t(i,j,r,q) = 2*j-2
//schedule for statement 2: t(i,j,r,q) = 2*r-1   i.e., 2*i-1
//placement for statement 1: p(i,j,r,q) = i
//placement for statement 2: p(i,j,r,q) = q      i.e., j
//deps 1,3 cut, 2,4,5 not cut
```

A.2 LooPo Specification for STM2

```
// second STM for LU
CONSTANT n
LOOPS i,j(i),r,q(r)
STATS l(i,j),u(r,q)
PSEUDOS k,kk

i: 1<=i, i<=n
j: 1<=j, j<=i

// q=j, r=i for u(i,j)
// index space for u is enumerated reversely
q: 2<=q, q<=n
r: 1<=r, r<=q-1


k: 1<=k, k<=n
kk: 1<=kk, kk<=n

STAT l :  L[i,j] := A[i,j] - SUM(L[i,k0]*U[k0,j])
STAT u :  U[q,r] := A[q,r] - SUM(L[r,kk0]*U[kk0,q])

DEP(): l(i,k) -> l(i,j) : j<=i, 1<=k, k<=j-1
DEP(): l(r,kk) -> u(r,q) : r<=q-1, 1<=kk, kk<=r-1, q>=2
DEP(): l(r,r) -> u(r,q) : r<=q-1, 1<=kk, kk<=r-1, q>=2
DEP(): u(kk,q) -> u(r,q) : r<=q-1, 1<=kk, kk<=r-1, q>=2
DEP(): u(k,j) -> l(i,j) : j<=i, 1<=k, k<=j-1
// for compatibility of the schedule with the placement:
DEP(): u(r,q-1) -> u(r,q) : q>=2
// for additionally having the same schedule in both statements:
DEP(): l(i-1,j) -> l(i,j) : i>=2

// result:
// schedule for statement 1: t(i,j,r,q) = i+j-2
// schedule for statement 2: t(i,j,r,q) = r+q-2  i.e., i+j-2
// placement for statement 1: p(i,j,r,q) = i
// placement for statement 2: p(i,j,r,q) = r     i.e., i
// deps 1-3,6 cut ; 4,5 not cut
```

Bibliography

[ACC+96] Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, François Irigoin, Pierre Jouvelot, and Ronan Keryell. PIPS: A framework for building interprocedural compilers, parallelizers and optimizers. Technical Report A/289, Centre de Recherche en Informatique, Ecole des Mines de Paris, April 1996. http://www.cri.ensmp.fr/~pips/.

[AGLP01] Marco Aldinucci, Sergei Gorlatch, Christian Lengauer, and Susanna Pelagatti. Towards parallel programming by transformation: The FAN skeleton framework. *Parallel Algorithms and Applications*, 16:87–121, 2001.

[Ald02] Marco Aldinucci. Automatic program transformation: The meta tool for skeleton-based languages. In *Constructive Methods for Parallel Programming Advances in Computation: Theory and Practice*. NOVA Science Publisher, Huntington, NY, USA, 2002. ISBN 1-59033-374-8.

[Ame92] American National Standards Institute, Inc. ANSI X3.198-1992, 1992.

[Ban93] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1993.

[Bar84] H. P. Barendregt. In J. Barwise, D. Kaplan, H. J. Keisler, P. Suppes, and A. S. Troelstra, editors, *The Lambda Calculus – Its Syntax and Semantics (revised edition)*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B.V., 1984.

[BEF+95] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LNCS 892, pages 141–154. Springer-Verlag, 1995.

[Ber02] Bert77: Automatic and Efficient Parallelizer for FORTRAN, 2002. Paralogic Inc., Bethlehem, PA, USA, http://www.plogic.com.

[BLOMP98] Silvia Breitinger, Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña. Eden – Language Definition and Operational Semantics. Technical Report 96-10, Fachbereich Mathematik und Informatik, Philipps Universität Marburg, 1998.

[Boa00] OpenMP Architecture Review Board. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, 2000. http://computer.org/cise/cs1998/c1046abs.htm.

[BOCF92] A. P. Wim Böhm, Rodney R. Oldehoeft, David C. Cann, and John T. Feo. *SISAL Reference Manual, Language Version 2.0*. Colorado State University – Lawrence Livermore National Laboratory, 1992.

[BP97] Ira D. Baxter and Christopher W. Pidgeon. Software change through design maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 250–259. IEEE Computer Society Press, 1997. Also on www.semdesigns.com.

[Bra98] Thomas Brandes. *ADAPTOR Programmer's Guide, Version 6.0*, June 1998. Available via anonymous ftp from ftp.gmd.de as gmd/adaptor/docs/pguide.ps.

[Che86] Marina C. Chen. Placement and interconnection of systolic processing elements: A new LU-decomposition algorithm. Technical Report YALEU/DCS/RR-498, Dept. of Computer Science, Yale University, October 1986.

[Col89] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[Col95] Jean-François Collard. *Parallélisation automatique des programmes à contrôle dynamique*. PhD thesis, Université Paris VI, 1995.

[DD95] John R. Davy and Peter M. Dew, editors. *Abstract Machine Models for Highly Parallel Computers*. Oxford Science Publications, 1995.

[DHHW93] Jack J. Dongarra, Rolf Hempel, Anthony J. G. Hey, and David W. Walker. A proposal for a userlevel, message passing interface in a distributed memory evironment. Technical Report TM-12231, Oak Ridge National Laboratory, February 1993.

[DHS00] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementations. *Future Generation Computer Systems*, 16:739–751, 2000.

[Dij68] Edsger W. Dijkstra. The structure of the "THE"-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[DM98] Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, January/March 1998.

[DR95] Michèle Dion and Yves Robert. Mapping affine loop nests: New results. In Bob Hertzberger and Giuseppe Serazzi, editors, *High-Performance Computing & Networking (HPCN'95)*, LNCS 919, pages 184–189. Springer-Verlag, 1995.

[EL97] Nils Ellmenreich and Christian Lengauer. On indexed data structures and functional matrix algorithms. Glasgow Functional Programming Workshop 1997, October 1997. http://www.dcs.gla.ac.uk/fp/workshops/fpw97/.

[FCB+98] Paul Feautrier, Jean-François Collard, Michel Barreteau, Denis Barthou, Albert Cohen, and Vincent Lefebvre. The Interplay of Expansion and Scheduling in PAF. Technical Report 1998/6, Laboratoire PRiSM, Université de Versailles, 1998.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[Fea92] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, 1992.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[Fos95] Ian Foster. *Design and Building Parallel Programs*. Addison-Wesley, 1995.

[G+94] Al Geist et al. *PVM: Parallel Virtual Machine*. MIT Press, 1994.

[Ger78] Curtis F. Gerald. *Applied Numerical Analysis*. Addison-Wesley, 2nd edition, 1978.

[GHC] The Glasgow Haskell Compiler. www.haskell.org/ghc.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GL96] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo. In Michael Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers (CPC'96)*, Konferenzen des Forschungszentrums Jülich 21, pages 311–320. Forschungszentrum Jülich, 1996.

[Goo01] Joy Goodman. *Incremental Program Transformations using Abstract Parallel Machines*. PhD thesis, Department of Computing Science, University of Glasgow, September 2001.

[Gor98] Sergei Gorlatch. Programming with divide-and-conquer skeletons: An application to FFT. *J. Supercomputing*, 12(1–2):85–97, 1998.

[Gri02] Personal communication with martin griebl, 2002.

[Gro03] Personal communication with Armin Größlinger, 2003.

[HAT] HAT – A Haskell Tracer. York Functional Programming Group, University of York, United Kingdom. www.haskell.org/hat.

[Her01] Christoph Armin Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, March 2001.

[HPF97] High Performance Fortran Forum. *HPF Language Specification*, 1997. http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/hpf2/index.cfm.

[LAM] LAM MPI. Indiana University, USA. www.lam-mpi.org.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[LM96] Vincent Loechner and Cathérine Mongenet. OPERA: A toolbox for loop parallelization. In Innes Jelly, Ian Gorton, and Peter Croll, editors, *Proc. 1st Int. Workshop on Software Engineering for Parallel and Distributed Systems*, pages 134–145. Chapman & Hall, 1996.

[Map] Maple Version 6. Waterloo Maple Software, Ontario, Canada. www.maplesoft.com.

[Moo03] Gordon E. Moore. No exponential is forever: but "Forever" can be delayed! In *Proceedings of the IEEE Solid-State Circuits Conference '03 (ISSCC)*, volume 1, pages 20–23. IEEE International, 2003.

[MRRV99] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A cost model for asynchronous and structured message passing. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99 Parallel Processing*, LNCS 1685, pages 552–556. Springer-Verlag, 1999.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[OCa] O'Caml Programming Language. INRIA, France. www.ocaml.org.

[OR97] John O'Donnell and Gudula Rünger. A methodology for deriving abstract parallel programs with a family of parallel abstract machines. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *EuroPar'97: Parallel Processing*, LNCS 1300, pages 662–669. Springer-Verlag, 1997.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Par76] David L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, 2(1):1–9, 1976.

[PGH+90] Constantine Polychronopoulos, Milind B. Girkar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, and Dale A. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'90)*, Research Monographs in Parallel and Distributed Computing, pages 423–453. Pitman, 1990.

[PJHe99] Simon Peyton-Jones and John Hughes (editors). Haskell 98, A Non-Strict, Purely Functional Language, February 1999. http://www.haskell.org/onlinereport.

[PJJM97] Simon Peyton-Jones, Mark Jones, and Eric Meijer. Type classes: Exploring the design space. In *Proceedings of the Second Haskell Workshop*, Amsterdam, June 1997.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator – A System for Constructing Language-Based Editors*. Texts and Monographs in Computer Science. Springer Verlag, 1989.

[Sca] ScaLi. Oslo, Norway. www.scali.com.

[Sch86] Alexander Schrijver. *Theory of Linear and Integer Programming*. Series in Discrete Mathematics. John Wiley & Sons, 1986. Section 12.2.

[SHM97] David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

[THM+96] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadephia, USA, May 1996.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2nd edition, 1999.

[Tur37] Alan M. Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4), December 1937.

[WB89] P. Wadler and S. Blott. How to make *ad hoc* polymorphism less *ad hoc*. In *Proceedings of the 16th ACM Symposium on the Principles of Programming Languages (POPL '89)*, pages 60–76, January 1989.

[WFW+94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *Proc. Fourth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 31–37. ACM Press, 1994. http://suif.stanford.edu/suif/.

[Wie95] Christian Wieninger. Automatische Methoden zur Parallelisierung im Polyedermodell. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[Win01] Noel Winstanley. *Staged Methodologies for Parallel Programming*. PhD thesis, Department of Computing Science, University of Glasgow, April 2001.

[Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[Wol89] Michael Wolfe. Iteration space tiling for memory hierarchies. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 357–361. SIAM, 1989.

[Wol95] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.

Danksagung

Mein besonderer Dank gilt Professor Christian Lengauer, der als Doktorvater diese Arbeit über mehrere Jahre freundschaftlich betreut hat und ohne dessen Unterstützung sie in dieser Form nicht möglich gewesen wäre. Diskussionen mit meinen späteren Zweitgutachter John T. O'Donnell über Haskell und APMs waren für diese Arbeit richtungsweisend. Professor Paul Feautrier hat mit seiner Erfahrung im Bereich der Schleifenparallelisierung die Arbeit am PolyAPM-Entscheidungsgraphen entscheidend mit beeinflusst.

Mein langjähriger Bürokollege Martin Griebl hat als Mitautor eines Papieres, als Ratgeber, LooPo-Hotline und Freund meine Arbeit wesentlich erleichtert. Die Mitarbeiter am Lehrstuhl für Programmierung, Peter Faber, Armin Größlinger, Sergei Gorlatch und Christoph Herrmann, haben mit kritischem Rat und praktischer Hilfe ihren Anteil an dieser Arbeit. Mein studentischer Mitarbeiter Michael Claßen unterstützte mich bei der Durchführung der Benchmarks und der Programmierung einer Fallstudie. Im Lehrstuhlsekretariat hat Johanna Bucur oftmals als gute Fee geholfen, wenn es zeitlich eng wurde. Ihnen allen sei hiermit herzlich gedankt.

Freunde und Familie haben viel Rücksicht auf meine knappe Freizeit genommen und waren trotz mancher Schwierigkeiten immer da. Die wichtigste Stütze ist meine Frau Natalija, die die Promotionszeit mit viel Verständnis für meine Arbeit zusammen mit mir durchgestanden hat. Vielen, vielen Dank!