View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Edinburgh Research Archive

THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Task Assignment in Parallel Processor Systems

Sathiamoorthy Manoharan

Ph. D.
University of Edinburgh
1992

EX BIBL. UNIV. EDINBURGH

Abstract

This thesis studies the problem of assigning programs onto parallel processor systems. It develops a generic simulation environment to model parallel systems and uses this environment to assess various assignment techniques.

Graphs are used in modelling programs, and based on these program models, a taxonomy for assignment schemes is proposed. Assignment schemes are broadly classified into schemes dealing with dependency graphs and schemes dealing with interaction graphs. Desirable properties for efficient assignments under different program models are discussed.

In contrast to the assignment of an interaction graph, an assignment of a dependency graph, in general, can be proved to be close to the optimal assignment. Moreover, the explicit temporal information made available by dependency graphs helps in establishing better assignment heuristics. The thesis thus focuses on the assignment of dependency graphs.

Most of the published schemes for assigning dependency graphs are work-greedy. Their heuristics is based on satisfying the following rule of thumb: keeping the processors busy will lead to a 'good' assignment. These schemes do not let a processor idle if there is a task the processor could execute. New analytical results bounding the performance of work-greedy assignment schemes are derived. It is shown that, when communication costs cannot be ignored, work-greedy assignment schemes may not perform well. An alternative assignment scheme which has a time-complexity lower than those of the work-greedy schemes is proposed.

A generic object-oriented simulation platform is developed in order to conduct experiments on the performance of assignment schemes. The simulation platform, called Genesis, is generic in the sense that it can model the key parameters that describe a parallel system: the architecture, the program, the assignment scheme and the message routing strategy. Genesis uses as its basis a sound architectural representation scheme developed in the thesis.

The thesis reports results from a number of experiments assessing the performance of assignment schemes using Genesis. The comparison results indicate that the new assignment scheme proposed in this thesis is a promising alternative to the work-greedy assignment schemes. The proposed scheme has a time-complexity less than those of the work-greedy schemes and achieves an average performance better than, or comparable to, those of the work-greedy schemes.

To generate an assignment, some parameters describing the program model will be required. In many cases, accurate estimation of these parameters is hard. It is thought that inaccuracies in the estimation would lead to poor assignments. The thesis investigates this speculation and presents experimental evidence that shows such inaccuracies do not greatly affect the quality of the assignments.

Acknowledgements

To my supervisors Nigel Topham and Roland Ibbett for their time and helpful advices; to Peter Thanisch for our fruitful discussions and his many useful comments; to David Skillicorn for his help throughout the development of Genesis; to Roy Campbell for object-orienting me; to Tom Waring and Leslie Goldberg for their advice, suggestions and encouragement; and to everyone who helped me with my work.

To the Department of Computer Science for providing me with an interesting work environment.

I was supported by a University of Edinburgh Postgraduate Studentship and an Overseas Research Students Award.

Table of Contents

Abstract i

1. Introduction 1

2. Models and Schemes for Assignment 8

2.1 Assignment of Dependency Graphs 9

2.1.1 Preemptive and Nonpreemptive Assignments 10

2.1.2 Work-greedy Assignments 11

2.1.3 Non-work-greedy Assignments 12

2.1.4 Assignment of Independent Tasks 13

2.2 Assignment of Interaction Graphs 13

2.2.1 Assignment of Regular Graphs 15

2.3 A Taxonomy for Assignment 16

2.4 Desirable Properties for Efficient Assignments 17

2.4.1 Interaction Graphs 18

2.4.2 Dependency Graphs 21

2.5 Examples from the Literature 22

2.5.1 Interaction Graphs 22

2.5.2 Dependency Graphs 25

2.6 Summary 27

3. Assignment of Dependency Graphs **29**

3.1 On the Assignment of Dependency Graphs 31

3.2 Work-Greedy Assignments 36

3.2.1 Brief Reviews of Some Work-Greedy Assignments 36

3.3 General Bounds on the Makespan of Work-Greedy Assignments . . 39

3.3.1 Independent Tasks 40

3.3.2 Dependency Graphs with Zero Communication Times 44

3.3.3 Dependency Graphs with Unit Computation and Communication Times 47

3.3.4 Dependency Graphs with Arbitrary Computation and Communication Times 47

3.3.5 Implications of the Bounds on Makespans 51

3.4 A Non-Work-Greedy Scheme for Assignment 53

3.4.1 DFBN: The New Scheme 55

3.4.2 Processor Ordering 58

3.4.3 Task Ordering 59

3.4.4 Time-complexity of DFBN 60

3.4.5 Performance Guarantee 60

3.5 Summary 61

4. Representation of Parallel Architectures **64**

4.1 A Survey of Some Architectural Classification Schemes 65

4.1.1 Flynn's Scheme 65

4.1.2 Hockney's Scheme 66

4.1.3 Skillicorn's Scheme 67

4.1.4 Dasgupta's Scheme 70

4.2 A Representation Scheme for Parallel Architectures 72

4.2.1 A Refined Set of Atoms 73

4.2.2 The Representation Scheme 75

4.3 Summary 78

5. A Modelling Environment for Parallel Systems **80**

5.1 On the Design Choices 81

5.1.1 Object Orientation: Objects, Classes and Hierarchies 83

5.1.2 The Modelling Approach 83

5.2 Software Representation: The Software Hierarchy 85

5.3 Hardware Representation: The Hardware Hierarchy 86

5.3.1 Building a Hardware Model 89

5.4 Specifying Assignment and Routing Schemes 91

5.5 Implementation Notes 93

5.5.1 Definition of the Class Atom 95

5.5.2 An Example: Building a Processor Grid 96

5.6 Comparison with Related Works 101

5.7 Summary 103

6. Performance Assessment of Assignment Schemes 104

6.1 Optimal Assignments 106

6.2 Assigning Random Graphs 112

6.2.1 Assessing the Effect of Estimation Errors 114

6.3 Dependency Graphs from Programs 118

6.3.1 Effect of Estimation Errors 121

6.3.2 On the Assignment of Loops 123

6.4 Varying the Size of the Processor Graphs 126

6.5 Task Assignment on Meiko 128

6.6 Summary 131

7. Summary and Conclusions 133

7.1 Future Directions 137

A. Definition of the Watchdog 152

B. Dynamic Behaviour of a Processor 154

C. Tables 157

Chapter 1

Introduction

Execution of a program on a parallel processing system requires the program to be decomposed into several modules that can be executed concurrently by the processors. Such modules are called tasks. Most of the parallel languages – for example, Occam, Concurrent C or Modula 2+ – leave such decomposition to the user; the user should 'think in parallel' and explicitly decompose the program into parallel tasks. Other languages – for example, SISAL, IBM Parallel Fortran or Concurrent Prolog – do not support explicit decomposition; they depend on a compiler for decomposition.

Assume that the program has already been decomposed into tasks either by the user or by a compiler. The tasks comprising the program model must then be assigned to the set of processors so as to minimize the total completion time of the program. This is known as the *assignment* problem.

Let $\mathbf{T}$ be the set of n tasks $\{T_1, T_2, \ldots, T_n\}$ and $\mathbf{P}$ be the set of m processors $\{P_1, P_2, \ldots, P_m\}$ onto which $\mathbf{T}$ is to be assigned. Assignment is then defined to be a function

$$M : \mathbf{T} \mapsto \mathbf{P}$$

that maps the set of tasks onto the set of processors. M is defined for each task of $\mathbf{T}$. The total time the set of tasks $\mathbf{T}$ takes to execute on the set of processors

P is called the *makespan*. The objective of the assignment is to minimize the makespan.

The number of possible assignments is exponential in n. Thus, enumerating all the possible assignments and choosing the optimal one will be enormously time consuming (except for very small values of n). It is very unlikely that there could be a cleverer scheme to find the optimal assignment, since even the restricted versions of the assignment problem have been proved to be NP-complete [GJ79, Ull76,AP91]. Automating the assignment procedure is therefore hard.

Given the difficult nature of automated assignments, some parallel languages that leave the decomposition to the user require the user to specify the assignment as well. Languages such as Occam and POOL take this approach. For instance, Occam forces the user to map processes to processors and communication channels to physical links. These languages trade off portability of programs for the simplicity of compilers (and run-time systems).

Portable parallel programs require automated assignment schemes. Such automated assignment schemes, in general, use some heuristics and produce a near-optimal solution in a reasonable amount of time.

This thesis is a treatise on automated assignment schemes. It discusses the techniques and schemes for automated assignments. Throughout the thesis abstract program models, rather than specific programs, are assumed in order to maintain generality. Based on these abstract program models, the thesis presents a taxonomical framework for assignment schemes that broadly classifies the assignment schemes into schemes dealing with dependency graph models and schemes dealing with interaction graph models. The thesis then focuses on the assignment of dependency graphs, shows the impact of task ordering on the makespan and discusses the factors on which task ordering should depend. It derives new analytical results bounding the performance of a class of assignment schemes and presents a new scheme that is easy to implement.

Comparison of different assignment schemes requires extensive experiments. It is decided to carry out these experiments on a simulated parallel processor system so that the parameters of the system can be varied easily during the experiments. To this end, the thesis develops a generic object-oriented simulation environment for parallel processor systems. The simulation environment is generic in the sense that it can model different architectures, assignment schemes and message routing strategies on a single platform. The environment uses as its basis an architectural representation scheme developed in the thesis.

Performance of assignment schemes are assessed through experiments conducted using the simulation environment. The thesis reports results of many such experiments.

Thesis Outline

The rest of this thesis is organized as follows.

Chapter 2 discusses the models and schemes used by automated assignment schemes. Graphs are used in modelling parallel programs. Based on these program models, a taxonomy for assignment schemes is proposed. Assignment schemes are broadly classified into schemes dealing with dependency graphs and those dealing with interaction graphs. Desirable properties for efficient assignments under different program models are discussed. Since these desirable properties are model-specific, the approaches taken by assignment schemes under different models are seen to be distinct. Some examples from recent literature are mentioned and are related in the light of the proposed taxonomy.

As opposed to the assignment of an interaction graph, an assignment of a dependency graph, in general, can be proved to be close to the optimal assignment. Moreover, the explicit temporal information made available by dependency graphs helps in establishing better assignment heuristics.

Chapter 3 thus focuses on the assignment of dependency graphs. Since even the restricted versions of the assignment problem are NP-complete, it is hard to find optimal assignments in a reasonable amount of time. Practical assignment schemes thus go for some heuristics that picks up a near-optimal assignment in polynomial time.

The heuristics most of the current assignment schemes for dependency graphs use is based on satisfying the following rule of thumb: keeping the processors busy leads to a 'good' assignment. Such schemes are said to be work-greedy. Work-greedy assignments are important since most of them provide a solution with a guarantee. It is proved that, when communication costs can be ignored, *any* work-greedy assignment would be close to the optimal assignment by no more than a small constant factor. It is also proved that this does not hold, should the communication costs be taken into account; that is, with communication costs, a work-greedy assignment can perform worse than the optimal assignment by a large factor that depends on the communication costs along some path in the task graph.

A non-work-greedy assignment scheme whose heuristics is not based on keeping the processors busy is proposed. The scheme is based on satisfying two desirable properties put forward in chapter 2: assigning independent tasks to different processors, and assigning dependent tasks to the same processor. The new scheme has a time-complexity at least an order less than the work-greedy schemes.

Performance assessment of these assignment schemes is the goal of the remainder of the thesis. Performance of a parallel system depends on the architecture, program, the assignment scheme and the message routing strategy. We develop a generic modelling approach that lets us specify and model these parameters and use this approach to simulate program execution on some processor topologies under different assignment schemes. These simulations aid the performance assessment of the assignment schemes.

The development of a generic modelling approach requires the following.

1. A representation scheme based on an abstraction level that integrates most of the possible architectural schemes.

2. Representing software in an architecture-independent way.

3. Providing the means to specify the assignment scheme and the routing strategies.

Chapter 4 develops a structural framework for representing parallel architectures. A set of functional units forming the basic blocks of architectures is identified. These functional units serve as building blocks in constructing architectures. Structural diagrams are used in representing the architectures thus constructed. Genesis, a generic modelling environment for parallel systems, is based on the representation scheme developed in this chapter.

Chapter 5 discusses the design and implementation aspects of Genesis. Genesis takes an object-oriented view of the entire parallel system, viewing both the architecture and the software as sets of objects. Every single functional unit of the architecture is modelled by an object; software entities - tasks, task graphs and messages, for instance - too are modelled by objects. In addition, there are means to specify various assignment and routing schemes. Genesis is thus a tool to describe and model the key parameters determining the performance of a parallel system: the architecture, program, assignment method and routing scheme. It is a good laboratory for carrying out experiments in performance analysis.

Chapter 6 uses Genesis as a modelling platform to analyse the performance of the work-greedy assignment schemes and the proposed non-work-greedy scheme. Using Genesis, processor topologies are constructed and the execution of a number of task graphs is simulated under different assignment schemes. Optimal assignments are found for small task graphs and these are compared against those assignments generated by the chosen assignment schemes. The schemes are then tested with random task graphs as well as task graphs obtained from real programs. The

possibility of testing the assignment schemes on a real multiprocessor system is also investigated in chapter 6.

Static assignment schemes assume that the task graph parameters – task execution times, volumes of information transfer, etc. – are known at compile time. However, in practice, run-time dependencies prohibit accurate measurement of these parameters. One would expect that such inaccuracies would lead to poor assignments. Chapter 6 thus investigates this speculation and presents experimental evidence that shows the impact of measurement or estimation inaccuracies on the quality of assignments is small.

The final chapter concludes with a summary.

Contributions of the Thesis

The specific contributions of this thesis are as follows:

1. A taxonomy for assignment schemes [Man91].

2. Performance guarantees for the work-greedy assignments of

   - independent tasks [MT90]
   - dependency graphs ignoring communication delays, and
   - dependency graphs with communication delays.

3. A non-work-greedy assignment scheme [MT91].

4. Performance and error-sensitivity analyses of assignment schemes for dependency graphs.

5. A structural representation scheme for parallel architectures.

6. An implementation of an object-oriented environment – Genesis – to model and simulate parallel systems [Man92].

Chapter 2

Models and Schemes for Assignment

A parallel program can be best viewed as a graph: the vertices represent the tasks and the edges represent the dependencies or interactions between the tasks. This gives rise to two models that represent parallel programs: a dependency graph and an interaction graph. See figure 2–1. In dependency graphs, the edges dictate a temporal dependency on the tasks they connect, i.e. the simultaneous execution of the tasks connected by an edge is prohibited. In interaction graphs the edges simply represent the interactions between the tasks they connect. Temporal dependencies are not explicit in an interaction graph: two tasks connected by an edge are thus simultaneously executable.

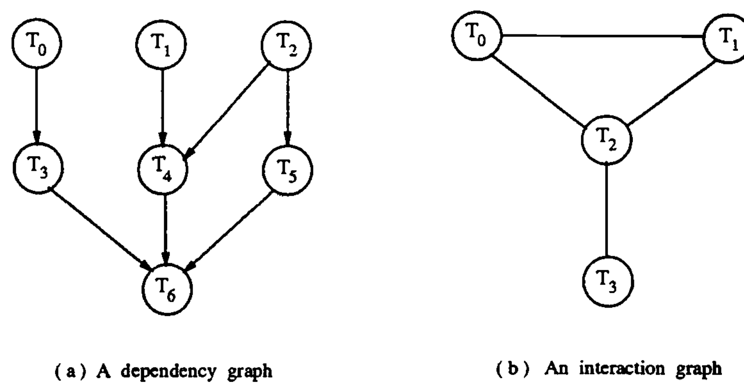


Figure 2–1: Program models: dependency and interaction graphs.

Both dependency and interaction graphs may have weights associated with their vertices and edges: the weight on a vertex indicates the amount of computation

the corresponding task performs, and the weight on an edge indicates the amount of communication between the tasks the edge connects.

Many assignment schemes are based either on dependency graph models or on interaction graph models. That is, they assume that the program has already been transformed into one of these graph forms and work their way forward to find a mapping of the task set onto the set of processors.

Some models of computation, for instance CSP [Hoa78] or CCS [Mil89], are well suited to the interaction graph forms whilst some other models of computation, for instance a dataflow computation model [GPC88,Sar89], are well suited to the dependency graph forms. The ease of transformation of the program into a suitable graph form thus depends on the user's model of computation. Programs written in Occam, for instance, are easy to model as an interaction graph whereas programs written in SISAL can be easily modelled as a dependency graph.

2.1 Assignment of Dependency Graphs

Tasks in a dependency graph have computation times associated with them. The graph edges, in addition to specifying temporal dependencies and thus a partial order on the task set, specify the volumes of information transfer that take place between the tasks they connect. Tasks receive information on their input edges and send information on their output edges. A task becomes ready to execute when all its input information is received, and finishes execution when it has produced all the required outputs. It is assumed that the task produces no output whilst it executes and then produces all outputs instantaneously when it finishes executing.

Dependency graphs are assumed to be acyclic, since this assumption makes their assignment simpler. When a program contains loops and conditional branches, this model does not seem to be realistic. However, there are techniques to convert

cyclic dependency graphs (which correspond to programs containing loops and conditional branches) into acyclic ones:

- Probabilistic techniques associate with every task graph edge a nonzero probability [ME67,Tow86]. If there is a directed edge from task T_i to task T_j, then associated with this edge is the probability that task T_j will be executed following the execution of T_i.

- Conditional branches can be collapsed into single tasks [RG69].

- Conditional branches introduce exclusive solution paths. Directed acyclic graphs for each of these paths could be obtained and mapped onto the same set of processors [SWP90].

- Loops can be unrolled [ERL91] or collapsed into single tasks. Loop-unrolling is briefly discussed in section 6.3.2.

The acyclicity constraint on dependency graphs is assumed throughout this thesis.

2.1.1 Preemptive and Nonpreemptive Assignments

Depending on whether a task's execution can be suspended or not, assignment strategies for dependency graphs take two forms: preemptive and nonpreemptive. In the nonpreemptive case, a task is executed continuously from start to finish on the same processor. In the preemptive case, execution of a task can be interrupted under the assumption that it will be resumed at a later time on some (not necessarily the same) processor.

Preemptive assignments may have makespans shorter than those of nonpreemptive ones. Consider the assignment of three independent tasks T_0, T_1 and T_2 each of execution time 2 on two identical processors P_0 and P_1. The Gantt charts in fig-

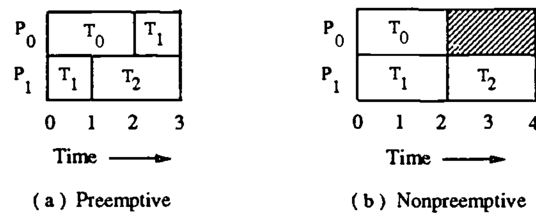


Figure 2–2: Preemptive assignments may be better than nonpreemptive ones

ure 2–2, adapted from [Cof76], demonstrate that the makespan of the preemptive assignment is shorter than that of the nonpreemptive one. However, not all tasks can be preempted. If a task is atomic, i.e. indivisible, then it cannot be preempted. Besides, preemption is not free: it involves some context-switching overheads. It is also hard to decide whether and when to preempt a task.

2.1.2 Work-greedy Assignments

Most of the known nonpreemptive assignment schemes for dependency graphs are work-greedy. The heuristics used by a work-greedy assignment scheme is based on satisfying the following rule of thumb: keeping the processors busy leads to a 'good' assignment. That is, a work-greedy assignment does not let a processor idle if there is a task it could execute. Work-greedy schemes, in general, generate assignments with a guarantee: the assignments can be provably close to the optimal assignment.

It can be shown analytically that no work-greedy assignment can be worse than the optimal assignment by more than a constant factor. When the communication costs are fixed, this constant factor is small. Assignment of independent tasks [MT90], assignment of dependency graphs with arbitrary computation costs but zero communication costs [Gra76], and assignment of dependency graphs with unit computation and unit communication costs [RS87] are some cases where the existence of the small constant factor has been proved. In all these cases, it can be shown that

$$\frac{\omega'}{\omega} \leq 2$$

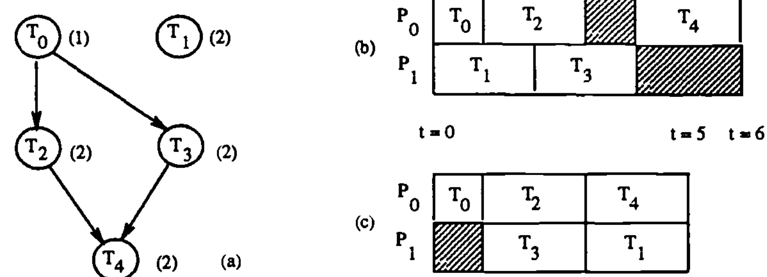
where ω' is the makespan of a work-greedy assignment and ω is the makespan of the optimal assignment. However, poor assignment strategies may have greater impact on the makespan when the communication costs are arbitrary [BMRS88]. The next chapter looks at this issue in detail and proves tighter bounds on makespans.

Given the varying nature of communication costs, it is not always easy to guarantee that a processor will not idle when there is a task it could execute. For instance, a work-greedy assignment may, at compile-time, assign a task T to a processor P such that the start time of T is the earliest when T executes on P. This also ensures that P will be kept busy as much as possible. However, at run-time, due to routing decisions and contention in the network, T may not be able to start on P at the predicted time. It may also be possible, under the prevailing network conditions, that there could be a processor P' on which T would have started earlier than it would have on P. That is, processor P' may be idling even though there is a task T in the system that it could execute.

2.1.3 Non-work-greedy Assignments

Keeping the processors busy is not the prime goal of non-work-greedy assignment schemes. That is, a non-work-greedy assignment may have a processor idling even when there is a ready task that the processor could execute. This may seem inefficient at first sight. The following example illustrates that, in fact, a non-work-greedy assignment may perform better than a work-greedy assignment. See figure 2–3. The task graph is assigned to two identical, connected processors P_0 and P_1. Note that, under the non-work-greedy assignment, processor P_0 idles during the time interval $(0, 1)$ although task T_3 is executable during this interval.

The work-greedy assignment shown in figure 2–3(b) is the best any work-greedy assignment can generate. The makespan of this assignment is larger than that of the non-work-greedy assignment. The existence of non-critical, ready tasks make work-greedy assignment schemes fair poorer than the non-work-greedy schemes.



(a) Task graph (b) A work-greedy assignment (c) A non-work-greedy assignment
(Numerals in parenthesis denote task execution times.
Number of processors is two.
Communication delay is assumed to be zero)

Figure 2–3: A comparison of work-greedy and non-work-greedy assignments

However, predicting whether or not it is desirable to delay the execution of a non-critical task is not easy. Therefore, non-work-greedy schemes may be more complicated than the work-greedy schemes.

2.1.4 Assignment of Independent Tasks

An important and well-studied class of program graphs arises when all the tasks are independent, that is, when there are no dependencies or interactions between the tasks. Since the execution times of these tasks carry all the temporal information required by dependency graph models, the assignment of independent tasks is indeed a special case of the assignment of dependency graphs.

2.2 Assignment of Interaction Graphs

Tasks in an interaction graph have an average computational load associated with them. Each graph edge specifies the volume of information transfer that takes place between the tasks that it connects.

Tasks execute simultaneously by going through a series of ***compute*** and ***communicate*** steps. The completion time of a task in an interaction graph is indeterministic. Therefore, the makespan of an assignment of an interaction graph is indeterministic. It can be neither calculated nor expressed in terms of the interaction graph parameters. Thus, assignment schemes for interaction graphs set their objective to satisfy a set of desirable properties that can be expressed in terms of the graph parameters, rather than to achieve the minimum makespan. An objective function that satisfies the set of desirable properties is formulated and used by the assignment schemes. The objective function evaluates the quality or the cost of an assignment. An optimal assignment refers to the assignment that optimizes this objective function rather than the assignment that minimizes the makespan.

Several objective functions have been used in the literature. They can be classified into three groups. The first group of functions aims to balance the computation costs among the processors and the second group aims to minimize communication costs. The third group of functions aims to balance the computation costs whilst minimizing communication costs.

Just balancing the computation costs will result in an assignment that has all its task distributed across the available processors. Therefore, when the inter-task communication costs are large, the first group of objective functions may not do well. Similarly, just minimizing the communication costs will result in a trivial assignment that clusters all the tasks into a single processor (or a group of processors). Thus, the second group of objective functions, used alone, cannot be a reasonable goal for assignment. Good assignment schemes, therefore, use objective functions of the third group.

A naïve approach to arrive at the optimal assignment is through an exhaustive search for the assignment that minimizes the objective function. Unfortunately, given n tasks and m processors, the number of possible assignments is m^n. Thus the naïve approach will be time consuming.

A straightforward way of reducing the search time is to use established search

improvement techniques, such as branch-and-bound search with underestimates or the A* search [Win84]. These techniques reduce the best-case search time. Yet, the worst-case time remains exponential.

Another technique widely employed by assignment schemes is iterative improvement. These schemes start from an initial assignment and improve its quality by iteratively moving tasks between processors. The improvement is measured by the objective function. Iterative improvement methods may not work always, for there are chances of getting stuck at a local optimum of the objective function. Probabilistic jumps to nearby solutions may permit further improvement in such cases. Simulated annealing [K+83] is a technique to get around the local optima in a systematic way. In the worst-case, all iterative improvement techniques take exponential time. But in practice, by choosing appropriate improvement mechanisms, speedy solutions are possible.

2.2.1 Assignment of Regular Graphs

The above discussions apply for any arbitrary interaction graph. However, simpler assignment techniques can be used for those interaction graphs that are regular. In a regular graph, all the tasks have the same characteristics and all the inter-task communications are of the same volume. Regular graphs form the models of iterative parallel programs in which the computation and communication patterns are regular and identical during each iteration. The regular nature of the graphs, in most cases, permits one to consider assignment as a simple geometric partitioning problem.

2.3 A Taxonomy for Assignment

Based on the discussion in the previous section, a taxonomy for assignment schemes is proposed here. The related earlier works on taxonomies for assignment schemes include [CK88], [WM85] and [SE87].

The taxonomy presented by Casavant and Kuhl is based primarily on solution techniques [CK88]. By 'solution technique', we mean the methods and ways of arriving at a solution. Optimal, heuristic and graph-theoretic methods are some examples. The taxonomy is partly hierarchical and partly flat. The characteristics that do not fit uniquely under any particular branch of their hierarchical taxonomy are placed in the flat part of the taxonomy. In fact, the characteristics forming the flat part could be branches beneath several leaves of the hierarchy.

Since similar solution techniques apply to different assignment schemes, the base of their hierarchical taxonomy is filled with many identical leaves. Most of the known solutions to the assignment problem are heuristic sub-optimals. Thus the taxonomy of Casavant and Kuhl places most of the assignment schemes in the heuristic sub-optimal class. A taxonomy based solely on solution techniques often groups assignment schemes that are not particularly related.

Wang and Morris present a taxonomy for load balancing [WM85]. Load balancing, however, is just one criterion for efficient assignment of certain program models. Thus their taxonomy cannot categorize most of the assignment methods.

Sadayappan and Ercal mention a taxonomy based partly on program models [SE87]. However, the structure of their taxonomy is not sufficiently expressive. For instance, assignment of independent tasks is not classified under the assignment of dependency graphs; rather, it is treated as special, at the top level of the taxonomy.

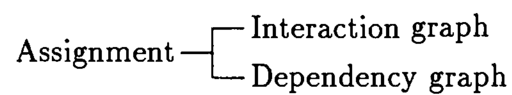


Figure 2–4: A broad classification for assignment based on program models

Figure 2–4 illustrates our broad taxonomy based on program models. Assignment schemes for dependency graphs are classified further: figure 2–5 illustrates the taxonomy.

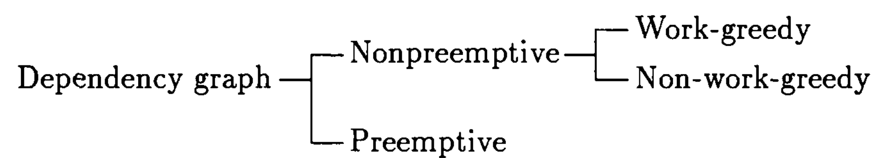


Figure 2–5: A broad taxonomy for dependency graph assignment

Task graphs can be generated either at compile time or at run time. Static assignment schemes use task graphs generated at compile time; dynamic schemes use the graphs generated at run time. Thus the taxonomy classifies both static and dynamic assignment schemes in the same framework.

2.4 Desirable Properties for Efficient Assignments

The heuristics used by most of the assignment schemes are based on satisfying two desirable properties: balancing the computation costs among the processors and minimizing the communication costs. To balance the computation costs needs an even distribution of tasks across the available processors; and the minimization of communication costs requires to cluster tasks together onto a single processor. These two are contradictory goals. The efficiency of an assignment scheme depends on how well the scheme exploits the graph structure to arrive at an assignment that achieves both these goals. It involves trade-offs between satisfying the two

stated desirable properties. This section discusses some desirable properties for efficient assignments in the light of the program models that have been described earlier.

The desirable properties depend also on the parallel system onto which the program is assigned. In particular, the properties that relate to the minimization of communication costs may vary according to the type of the parallel system. In distributed-memory systems communication between any two tasks executing on two different processors depends on the *distance* between the two processors. In most of the shared-memory systems, where the tasks are held in a common pool and communication is via a shared address space, communication cost between any two tasks is independent of where the tasks execute. This communication cost depends upon the architectural characteristics and the workload of the memory system and the interconnects. Resource contentions and conflicts in the memory and interconnects may increase the communication cost. However, if the shared-memory systems exploit local storage (caches, registers, etc.) for local communication, then communication cost between the tasks placed in the same processor can be substantially less [Squ90].

Therefore, the desirable properties that aim to minimize communication costs must take into account the properties of the parallel system.

2.4.1 Interaction Graphs

In order to exploit the potential parallelism in an arbitrary interaction graph, the processors need to be equally loaded. This property is often referred to as *load balancing*. This is essentially the distribution of the tasks evenly across the processors so that each processor has an equal share of the total computational load.

To minimize the communication costs in a distributed-memory system, tasks with heavy interaction must be assigned to the same processor (or adjacent processors).

In addition, in systems where the interconnection is a topology, mapping of the task graph edges onto single processor links will minimize communication costs.

In shared-memory systems, if some local storage (registers, cache, etc.) is used for local communication, then it is desirable to assign those tasks that interact heavily to the same processor; otherwise communication costs are irrelevant as far as the assignment is concerned.

Regular Interaction Graphs

Regular graphs, by their very nature, permit simpler assignment techniques to be employed. For example, in systems comprising identical processors with regular communication network (regular topologies and shared-memory systems, for instance), assignment of regular graphs can be viewed as a simple geometric partitioning problem. Each partition generated by the assignment scheme is assigned to a suitable processor (chosen by the assignment scheme) for execution. In general, partitions and processors are so chosen that the communication is restricted to the nearest neighbours.

As an example, consider the regular interaction graph of figure 2–6(a). Tasks in this graph represent iterative processes that communicate with their nearest neighbours. For the processor topology of figure 2–7(b), the partition of figure 2–6(b) would suit best; for the processor topology of figure 2–7(c), the partition of figure 2–6(c) would suit best. In both these cases both the computation and communication loads are balanced. For the processor topology of figure 2–7(a), the partition of figure 2–6(c) is more suitable than that of figure 2–6(b); even though both partitions balance the computation costs, the communication cost per processor graph edge is less in the case of partition 2–6(b).

The computation time of a partition is proportional to the area of the partition; and the inter-partition communication time is proportional to the perimeter of the partition. Here 'area' means the number of task vertices within the partition, and

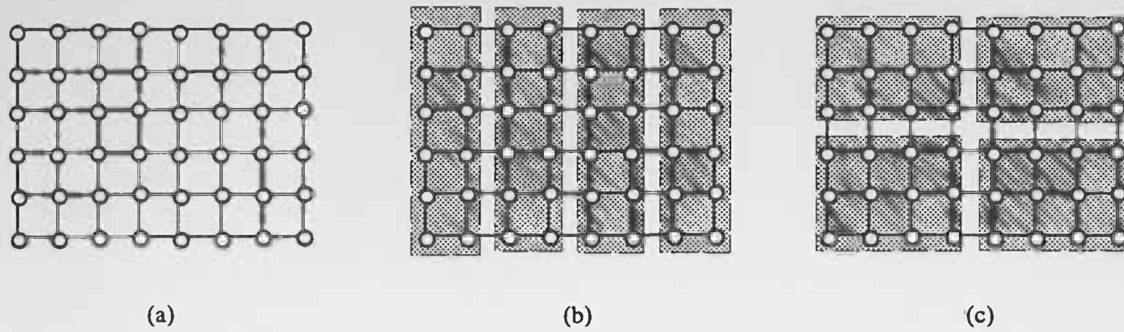


Figure 2–6: Partitioning regular graphs: Task graph and partitions.

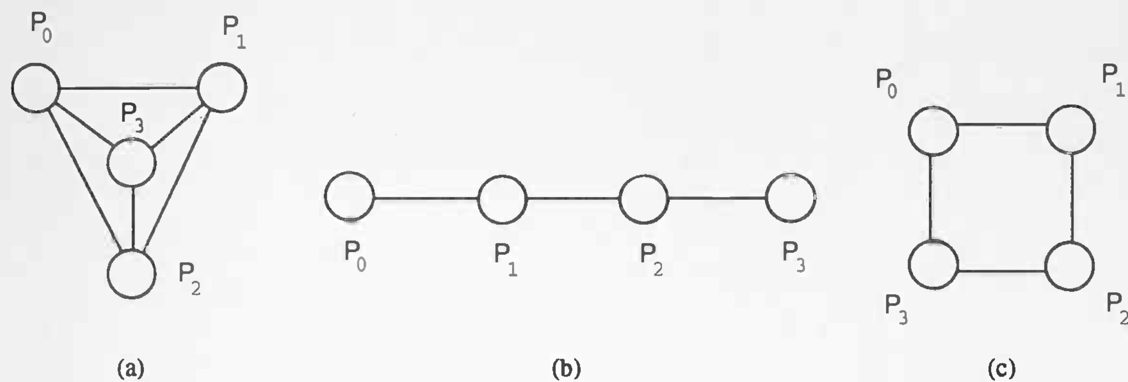


Figure 2–7: Partitioning regular graphs: Processor topologies.

'perimeter' means the number of vertices along the partition boundary. The task vertices along the partition boundary are responsible for the inter-partition communication (and, since each partition is assigned to a processor, inter-processor communication). The partition can be of different shapes: square, strip, rectangular, etc. (see figure 2–6). The processor topology and communication pattern will determine the exact shape of the partition [RF87].

Since both the computational load and the communication load need to be balanced, the following desirable properties apply for regular graphs:

- The sizes of the partitions need to be the same.
- The shapes of the partitions need to be the same.

2.4.2 Dependency Graphs

The temporal dependencies present in a dependency graph dictate a partial ordering on the tasks. This partial ordering does not permit equal loading of processors without either increasing the makespan or wasting processor resources. Therefore, load balancing makes little sense for dependency graphs. Assignment schemes for dependency graphs should thus employ a different criterion to minimize the computation costs.

Assume a dependency graph in which communication costs are negligible compared to the computation costs. The following desirable property that minimizes just the computation costs is thus adequate for such a graph:

- Processors must be kept as busy as possible.

The heuristics used by work-greedy assignment schemes is based on satisfying this property. Under this property tasks are executed at the earliest possibility.

When the communication costs vary, approaches that minimize the communication costs while balancing the computation costs are needed. One such approach is to extend the above desirable property to take communication costs into account. Work-greedy schemes that consider communication costs take this approach.

Another approach is to start with a different set of desirable properties that aims to balance computation costs and minimize communication costs. Such a set of desirable properties is stated as follows:

- Assignment of independent tasks to different processors.

  Independent tasks are concurrently executable. To minimize the makespan requires the concurrent execution of these independent tasks. Thus these tasks should be assigned to separate processors.

- Assignment of dependent tasks to the same processor.

> Dependent tasks can only be executed in sequence. They would gain nothing by being assigned to separate processors; and, more importantly, they could incur extra communication delays if they are executed in separate processors. Thus the dependent tasks should be assigned to the same processor.

2.5 Examples from the Literature

In this section, some recent literature on assignment is briefly reviewed and classified according to the proposed taxonomy. The distinctions and relationships between the published assignment schemes will then be easy to appreciate. Some of these schemes are dynamic and the rest are static.

Note that an extensive survey is *not* intended. Only a few typical examples are cited.

2.5.1 Interaction Graphs

Shen and Tsai view the assignment of interaction graphs as a type of graph matching problem called *weak homomorphism* [ST85]. If a graph G_T can be mapped onto another graph G_P such that there is a many-to-one mapping of the edges of G_T onto the edges of G_P, then there is a weak homomorphism from G_T to G_P. They use the A* algorithm [Win84] to find the minimum cost weakly homomorphic mapping.

Most of the heuristic assignment schemes for arbitrary graphs use a two step procedure: an initial assignment and an iterative improvement. Some schemes minimize the communication costs in the first step, and in the second step balance the computational load. Others balance the computational load first and then iteratively exchange tasks to minimize communication.

The scheme proposed by Efe first clusters heavily-communicating tasks together to form an initial assignment and then uses a task reassignment algorithm to obtain iteratively an assignment with balanced computational load [Efe82]. Sadayappan and Ercal address the problem of assigning non-uniform, irregular finite element meshes onto processor graphs [SE87]. The initial assignment is improved by boundary refinement to balance the computational load.

Bokhari presents a heuristic algorithm that improves an initial assignment through pairwise interchanges, the objective function being the number of task graph edges that fall on processor graph edges [Bok81]. He uses probabilistic jumps to guide the objective function out of local optima. Lee and Aggarwal propose a similar assignment scheme [LA87]. Unlike Bokhari, they examine only selected pairs for interchange; although, in the worst case, all the pairs could be selected. They also take the possibility of network contention into account.

Simulated annealing is a technique that generalizes the probabilistic jump approach to get around local optima. Donnet et al. show results indicating the effectiveness of simulated annealing over other iterative improvement methods [DSS88]. However, simulated annealing is time consuming. Parallel versions of simulated annealing are thus being employed by some assignment schemes [HMS].

Lin and Keller propose a dynamic scheme for assignment based on the so-called gradient strategy: a local, demand-driven load balancing method [LK87]. They assume locality of interactions among the task vertices, and thus aim to achieve a global load balance by successive localized balances. Kalé proposes a similar dynamic assignment scheme called *Contracting Within a Neighbourhood* (CWN) [Kal87].

Regular Interaction Graphs

Most of the assignment schemes published for the parallel numerical solutions of partial differential equations are good examples of this category [RF87]. The

solution domain of a partial differential equation (PDE) can be discretized into a 'grid' of points. The value at each grid point is updated in each iteration using the values at neighbouring points. These points can be updated in parallel. The computational work associated with each point is the same throughout the grid. Each of these grid points is a task to be executed; and these tasks interact with their neighbours. Hence the PDE grid forms a good example of a regular graph.

Vrsalovic et al. consider the assignment of regular graphs onto a shared bus multiprocessor [V^+85,V^+88]. They define speedup in terms of computation and communication decomposition functions. The computation (communication) decomposition function is the ratio of processing (data access) time for a single processor system to the processing (data access) time for a multiprocessor system. They consider three different partition shapes. For three different combinations of decomposition functions, they derive speedup considering both exclusive global data access and data access with local copying. In [Cve87], Cvetanovic extends the work done by Vrsalovic et al. Her analysis is not limited to shared buses. She defines speedup in terms of the bandwidth of the interconnection network and the computation and communication decomposition functions.

Reed et al. consider the assignment of a PDE solution grid onto both shared memory and message passing architectures [RF87,RAP87]. Their analysis differs from [V^+85,V^+88,Cve87] in that they consider the effect of *stencils* (the number of neighbours with whom a grid point interacts) on the speedup. They conclude that stencils, partition shape and architecture must be considered together for generating optimal assignments. In [NW88], Nicol and Willard derive expressions for optimal speedup and optimal number of processors for the assignment of PDE solution grids.

In [Bok88], Bokhari considers the problem of assigning a chain-structured parallel or pipelined program onto a chain of processors. Both the task and processor graph have nearest-neighbour communications. With the constraint that each processor should be assigned a contiguous subchain of tasks, Bokhari develops a

simple algorithm for finding the optimal assignment. The algorithm uses a layered graph to search for the optimal solution.

Regular graph structures may as well arise at an intermediate stage during the assignment process. For example, chain-structured graphs arise as an intermediate form in some of Bokhari's assignment schemes [Bok88]. The schemes generate optimal assignments of some classes of programs onto host-satellite processor systems with certain constraints. Program transformation techniques that transform classes of programs into pre-defined regular graph structures have been proposed elsewhere as well [Col89].

2.5.2 Dependency Graphs

Optimal polynomial time assignments of dependency graphs are available only for restricted cases. For example, in [Cof76] two such restricted cases are given: when the task graph is a forest, and when there are only two processors available. In both these cases task execution times are fixed at unity and communication costs are assumed to be zero.

Many of the assignment schemes use heuristics to arrive at near-optimal solutions.

Work-greedy Assignments. Most of the published work-greedy assignment schemes assume that the communication costs can be ignored. Coffman gives a good account of such work-greedy assignment schemes that ignore communication costs [Cof76]. Shirazi et al. [SWP90] and Adam et al. [ACD74] present comparative analysis of such assignment schemes. Work-greedy schemes that ignore communication delays have been used in scheduling dataflow graphs onto dataflow architectures [GKS87] and scheduling instruction streams onto pipelined processors [Kri90].

Rayward-Smith considers work-greedy assignments of dependency graphs with

unit execution and unit communication times [RS87]. Lee et al. [LHCA88], Wu and Gajski [WG88], Hwang et al. [HCAL89] and El-Rewini and Lewis [ERL90] propose work-greedy assignment schemes taking arbitrary communication costs into account. The next chapter will briefly describe these schemes.

Kruatrachue and Lewis introduce a work-greedy assignment scheme called *Duplication Scheduling Heuristics* (DSH) that replicates execution of some of the tasks so as to minimize the communication costs [KL87]. If a task T's execution on a processor P is delayed due to communication from a predecessor task T_p of T, then DSH examines if replication of T_p (and possibly T_p's predecessors and T_p's predecessors' predecessors and so on) on P will make T start earlier. The solutions that DSH generates are very good if the communication costs are large compared to computation costs. The trade-off here is the high time-complexity of the algorithm[1].

All the above assignment schemes are static. Chou and Abraham describe a dynamic assignment scheme [CA82]. They introduce probabilistic fork and join points in the task graph in order to model the probabilistic nature of the program. Partitions of the task graph are found using results in Markov decision theory. Communication costs are assumed to be zero in this scheme.

Non-work-greedy Assignments. Kim [Kim88] and Sarkar [Sar89] propose algorithms for non-work-greedy assignments and show that they perform well for task graphs with heavy communication. These algorithms follow a two step approach. The first step assigns the tasks onto an unbounded number of virtual processors. These virtual processors are completely connected and have equal interprocessor communication costs. The second step maps the virtual proces-

[1]The time-complexity of DSH is $\Theta(n^4 m)$, where n is the number of tasks and m is the number of processors.

sors onto the real processors. Yang and Gerasoulis propose a non-work-greedy algorithm called *Dominant Sequence Clustering* (DSC) for the first step [YG91].

The two-step non-work-greedy schemes are complex and involve large time-complexities. They do not provide any analytical performance guarantee as do the work-greedy schemes. Moreover, no experimental comparison between these schemes and work-greedy schemes have been reported.

Preemptive Assignments. Sahni [Sah84] and Blażewicz et al. [BDW86] address preemptive assignment of independent tasks. Sahni assumes the context-switching time to be non-zero and develops an algorithm to obtain a sub-optimal solution of known accuracy. Blażewicz et al. present a scheme for a system where tasks may need more than one processor at a time for their processing.

2.6 Summary

The models and schemes used for the solution of the assignment problem have been discussed. Graphs are used in modelling parallel programs. Based on these program models, a taxonomy for assignment is proposed. Assignment schemes are broadly classified into schemes dealing with dependency graphs and those dealing with interaction graphs. Desirable properties for efficient assignments under different program models are discussed. Since these desirable properties are model-specific, the approaches taken by assignment schemes under different models are seen to be distinct. Some examples from recent literature are mentioned and are related in the light of the proposed taxonomy.

The distinction between assignment techniques brought to light by the taxonomy is important. It aids research to take the right path in choosing a proper technique and not spending too much time over the others.

As opposed to interaction graphs, a dependency graph permits finding an assignment with a guarantee: the assignment can be proved to be close to the optimal assignment. Moreover, the explicit temporal information made available by dependency graphs helps in establishing better assignment heuristics. We thus choose to analyse in detail the problem of assigning dependency graphs.

The next chapter is a treatise on the assignment of dependency graphs. The impact of task ordering on the partitions of dependency graphs is shown. The factors that should determine the ordering are discussed. Work-greedy assignment schemes, particularly those that take the communication costs into account, are discussed. Solution guarantees are proved for work-greedy schemes. A single-step non-work-greedy assignment scheme, whose heuristics is based on satisfying the desirable properties put forward in section 2.4.2, is proposed.

Chapter 3

Assignment of Dependency Graphs

The previous chapter classified assignment schemes broadly into those dealing with interaction graphs and those dealing with dependency graphs. As opposed to interaction graphs, a dependency graph permits finding an assignment that can be provably close to the optimal. Besides, the temporal dependencies made available by dependency graphs help in finding better assignment heuristics. Thus, this chapter chooses to examine in detail the assignment of dependency graphs. During its course, it shows the impact of task ordering on the makespan and discusses the factors on which task ordering should depend. It presents some new results bounding the performance of work-greedy assignment schemes and proposes a new non-work-greedy assignment scheme. The time-complexity of the new scheme is at least an order less compared to the work-greedy schemes.

Some notations that need to be used subsequently are defined first. Other notations will be defined in context.

Notations.

n	number of tasks
m	number of processors
$\mathbf{T}$	set of tasks $\{ T_0, T_1, \ldots, T_{n-1} \}$
$\mathbf{P}$	set of processors $\{ P_0, P_1, \ldots, P_{m-1} \}$
τ_i	execution time of T_i assumed common on all P_j
s_i	memory space requirement of T_i
v_{ij}	volume of information transfer between T_i and T_j
c_{ij}	amount of information that can be transferred between P_i and P_j per unit time
μ_i	memory capacity of P_i
G_T	task graph depicting tasks and the dependencies among them
G_P	processor graph depicting processors and their interconnections
ω	the total execution time of G_T on G_P (i.e. the makespan)

The primary architectural considerations are the set of processors and the topology in which the processors are connected. The processor topology is modelled as a graph with vertices representing the processors and weighted edges representing the interconnections between the processors. All the processors are assumed to be capable of doing the functions required by the tasks.

Task graphs are assumed to be acyclic. A dataflow execution model is assumed for the execution of task graphs. That is, a task can begin its execution when all its inputs are available, and finishes only when it has produced all the required outputs. Communication delay may occur when a task sends its output to its successor tasks. This delay is dependent on the volume of information being transferred and the distance the information needs to travel. Tasks, once scheduled, cannot be preempted. Task replication is not considered, that is, no task can execute on more than one processor.

See figure 3–1 for example task and processor graphs. Figure 3–1(a) shows the task dependency graph corresponding to the evaluation of an expression $z =$

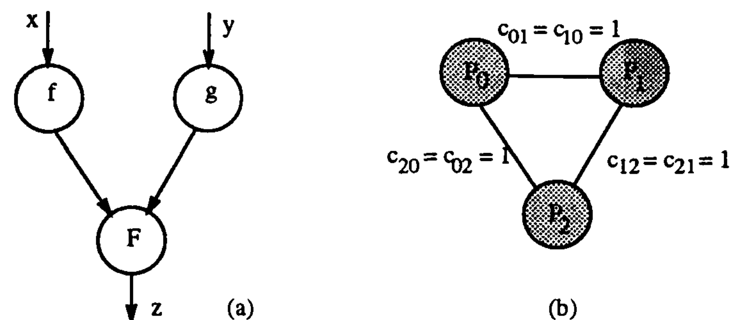


Figure 3–1: Example task and processor graphs.

$F(f(x), g(y))$. Figure 3–1(b) shows a three-processor system where all processors are equidistant[1] from each other.

3.1 On the Assignment of Dependency Graphs

An assignment divides the task set $\mathbf{T}$ into m, some possibly empty, *ordered* subsets or *partitions*. The objective of the assignment is to minimize the makespan of $\mathbf{T}$ on $\mathbf{P}$.

The following example illustrates the effect task ordering has on the makespan. Consider the assignment of the task graph of figure 3–3(a) on a two-processor system $\{P_0, P_1\}$ with zero interprocessor communication delay. The assignment

$$\{T_0, T_3, T_4\} \mapsto P_0; \ \{T_1, T_2, T_5\} \mapsto P_1$$

gives rise to a makespan of 3 units; whereas the assignment

$$\{T_0, T_3, T_4\} \mapsto P_0; \ \{T_2, T_1, T_5\} \mapsto P_1$$

gives rise to a makespan of 4 units. An increase in makespan is observed by changing the ordering of tasks belonging to the task partition mapped to P_1.

[1]Two processors, P_i and P_j, are ***equidistant*** from a processor P_k iff $c_{ik} = c_{jk} = c_{ki} = c_{kj}$.

Let $p(T, \mathcal{A})$ be the processor to which the task T is assigned under assignment $\mathcal{A}$. Two assignments $\mathcal{A}_1$ and $\mathcal{A}_2$ are said to be *equivalent* if $p(T_j, \mathcal{A}_1) = p(T_j, \mathcal{A}_2)\ \forall j$. In the example above, $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent. Equivalence of two assignments implies that the processors are assigned the same subsets of tasks under both assignments; yet the task orderings within these subsets are different.

Now consider a set of equivalent assignments $S = \{\ \mathcal{A}_1,\ \mathcal{A}_2,\ \ldots,\ \mathcal{A}_z\ \}$. Let the makespans of these assignments be $\omega_1,\ \omega_2,\ \ldots,$ and ω_z respectively. Let $\omega_{min} = \min_{i=1\ldots z} \omega_i$ and $\omega_{max} = \max_{i=1\ldots z} \omega_i$. Assume that the processors are never left idling intentionally, i.e. they idle only if there is no task they could execute. The following bound then holds.

Theorem 3.1.

$$\frac{\omega_{max}}{\omega_{min}} \leq m$$

Furthermore, this bound is tight.

Proof. This theorem follows as a special case of a theorem Jaffe [Jaf80] (and Liu and Liu [LL78]) proved. For a heterogeneous system with k types of tasks and m_i processors to execute tasks of type i, Jaffe proved that

$$\frac{\omega'}{\omega} \leq k + 1 - \frac{1}{\max_i m_i}$$

where ω is the length of the optimal makespan; and ω' is the makespan of any arbitrary assignment that assumes that the processors do not idle if there are tasks that they could execute. He also proved that this bound is tight.

Since each task of partition i of $\mathcal{A}$, belonging to the set of equivalent assignments S, can be considered to have type i, our theorem can be seen as a special case of Jaffe's theorem where $k = m$ and $m_i = 1\ \forall i$.

□

Theorem 3.1 establishes that an assignment with a poor task ordering can perform m times worse than an equivalent assignment with a good task ordering. Thus an

assignment scheme should not only determine to which processor the tasks are to be assigned but also determine the ordering of tasks assigned to each processor. This ordering is determined by giving suitable priorities to the tasks.

An obvious candidate for the top priority is the critical task. If a task's execution cannot be delayed without increasing the makespan, then the task is said to be critical. Experimental results have shown that choosing the critical task first leads to good assignments when the communication costs can be ignored [ACD74]. However, it is hard to find the critical task if communication delays are to be taken into account, since these communication delays depend on the assignment that is yet to be determined. Besides, giving the critical task top priority is not sufficient to guarantee an optimal assignment [SWP90].

We now identify those tasks that should be given priority.

1. Tasks with long execution times must get priority –

   Consider a fixed makespan. Assigning short-length tasks first leads to a state where there is no processor with enough time to fit a long-length task. Such *temporal fragmentations* increase the makespan. (See figure 3–2. A poor assignment results, if long-length tasks are not given priority.)

   Thus task T_i should be given priority proportional to τ_i.

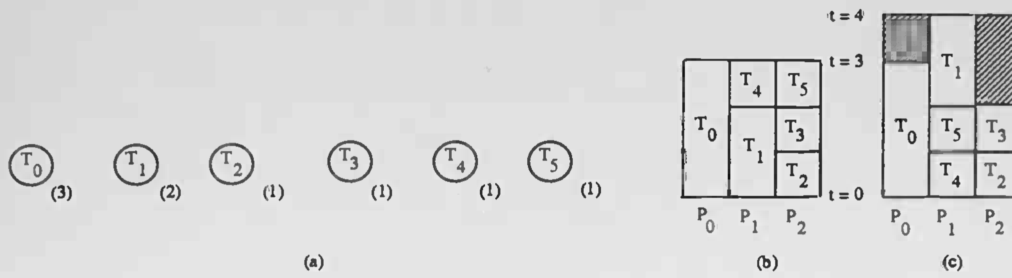
2. Tasks with large communication requirements must get priority –

   This again is due to the possible temporal fragmentation that may occur if priority is not given to tasks with large communications.

   Thus task T_i should be given priority proportional to

   $\sum_{T_j \in succ(T_i)} v_{ij}$ where $succ(T)$ denotes the set of successors of task T.

3. Tasks with large numbers of successors must get priority –



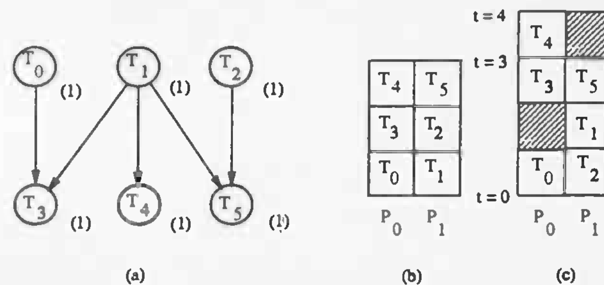
(a) Task graph (b) Optimal assignment (c) Poor assignment.
(Numerals in parenthesis denote task execution times.
Number of processors is three.)

Figure 3–2: Priority for long tasks.

By executing tasks with large numbers of successors first, task dependencies can be resolved as quickly as possible [KN84]. Thus, more tasks may become executable, reducing processor idle time and the makespan. (See figure 3–3. A poor assignment results, if tasks with large numbers of successors are not given priority.)

Thus task T_i should be given priority proportional to

$$\sum_{T_j \in succ(T_i)} 1$$



(a) Task graph (b) Optimal assignment (c) Poor assignment
(Numerals in parenthesis denote task execution times.
Number of processors is two.
Communication delay is assumed to be zero)

Figure 3–3: Priority for tasks with more successors.

4. Tasks with long-length successors must get priority

   The reason for this is the combination of the reasons given for 1 and 3 above. Thus task T_i should be given priority proportional to

$$\sum_{T_j \in succ(T_i)} \tau_j$$

5. Tasks with large memory space requirements must get priority –

   Assigning small-sized tasks first will lead to a situation where there will not be any processor with enough free memory to hold large tasks, though the total free memory space is large enough. Such *spatial fragmentations* can be reduced by assigning tasks with large space requirements first.

   Thus task T_i should be given priority proportional to s_i.

Assigning priorities to tasks is important even if the tasks are independent. The task selection mechanism in any assignment scheme should take these priorities into account. However, once an assignment is determined, task ordering becomes irrelevant in the case of independent tasks. Tasks within the partitions of an assignment can be executed in any order and this will not have any effect on the makespan. In other words, equivalent assignments of independent tasks have the same makespan.

Solving the assignment problem. A naïve approach to solve the assignment problem is to enumerate all the possible assignments and choose the assignment that gives the minimum makespan. However, this approach will take exponential time. It is very unlikely that there would be any cleverer scheme to find the optimal assignment in polynomial time, since even the restricted cases of the assignment problem have been proved to be NP-complete [Ull76,RS87]. Practical assignment schemes thus settle for heuristics that find sub-optimal assignments in polynomial time.

Most of these heuristic assignment schemes are work-greedy. The next section analyses work-greedy assignments in detail.

3.2 Work-Greedy Assignments

An assignment is work-greedy if no processor remains idle when there is a task the processor could execute. Work-greedy assignments are time-driven: tasks and processors are selected at specific time instances, i.e. when a processor becomes free or when a task finishes its execution.

Work-greedy assignment schemes, in addition to finding *where* to execute a task, attempt to find *when* to execute a task. That is, they always predict the start and finish times of the tasks. This permits computation of bounds on the makespans of work-greedy assignments.

3.2.1 Brief Reviews of Some Work-Greedy Assignments

Many work-greedy assignment schemes ignore communication delays. These schemes follow a common basic algorithm.

> Tasks are kept in a priority list. A free processor scans the list from left to right to find the first ready task to be executed. If there is a ready task, the processor executes the task until completion. Otherwise the processor idles until a task becomes ready.

This procedure ensures that the assignment is work-greedy. See [Cof76] for a good account of assignment methods not involving communication delays.

This section reviews some work-greedy assignment schemes that do take communication delays into account in arriving at an assignment. All assignment schemes

considered here are static, that is, the characteristics of the task dependency graph are assumed to be known at compile time.

Scheme ETF

The ETF (Earliest Task First) [HCAL89] algorithm uses two sets called the ***ready*** task set, A, and the free processor set, I. A task is said to be ready when all its predecessors are scheduled. The algorithm calculates the earliest start time e_s of every task that belongs to A on every processor belonging to I. The tasks are assigned in the ascending order of their earliest start times. Let the minimum of these earliest start times be $\hat{e}_s$, and the task and processor corresponding to this minimum be $\hat{T}$ and $\hat{P}$ respectively. It is worth noting here that the sets A and I change as each task finishes its execution – more tasks may become ready and at least one processor becomes free.

The algorithm uses two instances of an event clock to mark the current moment (CM) and the next moment (NM). An event is the termination of an executing task. The event clock advances with the completion times of the tasks. NM specifies the time instant to which the event clock would next advance. It is essentially the earliest time after CM at which one or more currently executing tasks finish execution. When a task finishes execution at NM, it may cause new tasks to become ready. It is possible that a newly ready task could have an e_s less than $\hat{e}_s$. In that case, it is this new task that should be scheduled first. The reason for using NM is simply to take these newly ready tasks into account. In essence, a task is scheduled at CM, if $NM > \hat{e}_s$. Otherwise the decision is postponed until the instant NM.

The time-complexity of the ETF scheme is $\Theta(n^2m)$, where n is the number of tasks and m is the number of processors.

Scheme ERT

ERT (Earliest Ready Task) [LHCA88] selects a task and processor combination so that the selected task is the earliest ready at a given moment. That is, the selection criterion in ERT is the minimum earliest ready time (not the earliest *start* time as in ETF). Thus, ERT does not postpone any scheduling decision until a further moment as ETF often does. As in ETF, task and processor selection methods are inseparable. ERT does not maintain a set of free processors. All the processors are checked against each task of the set of ready tasks to determine the best task and processor combination.

The time-complexity of the ERT scheme is $\Theta(n^2m)$.

Scheme MH

MH (Mapping Heuristic) [ERL90] selects for assignment the maximum priority task from the set of ready tasks at a given time. A task's priority is calculated from the task's level (the length of the longest path from this task to any end-task) and the number of successors the task has. The selected task is then assigned to the processor that can execute it earliest. Compare this with ERT, in which task and processor selection methods are inseparable.

In calculating communication delays, MH assumes an adaptive shortest path routing policy to deliver the messages. This takes network contention into account. The time complexity of the MH scheme is $\Theta(n^2m^3)$.

There is a restricted version of MH that does not use any adaptive routing in determining the communication delays. This scheme is called RMH.

Scheme MCP

MCP (Modified Critical Path) [WG88] is similar to RMH except for the choice of task priorities. In MCP tasks are given priority according to their latest start time. When two tasks have the same priority, the latest start times of their successors (and latest start times of successors' successors, and so on) are used to break the tie.

The time-complexity of the MCP scheme is $\Theta(n^2 \log n + n^2 m)$ of which $\Theta(n^2 \log n)$ is spent on calculating the task priorities.

3.3 General Bounds on the Makespan of Work-Greedy Assignments

A work-greedy assignment does not guarantee optimality. Yet, it is possible to prove that the makespan of a work-greedy assignment is within a constant factor of the makespan of the optimal assignment. This section presents some new results bounding the makespans of work-greedy assignments. The bounds are general in the sense that they apply for any task ordering employed. Implications of these bounds are discussed in section 3.3.5.

For the purpose of notational convenience, a work-greedy assignment is characterized as an ordered triple $\mathcal{W} = (\beta_1, \beta_2, \beta_3)$, where β_i are defined as follows.

1. β_1 characterizes the execution times of the tasks involved in the assignment. It depends solely on the dependency graph.

$$\beta_1 \in \{\text{arbitrary, unit}\}$$

2. β_2 characterizes the communication time between two tasks assigned to ***different*** processors. It depends on the dependency graph, the architecture onto

which the dependency graph is to be assigned and the assignment itself.

$$\beta_2 \in \{\text{arbitrary, unit, nil}\}$$

3. β_3 characterizes the precedence relation between the tasks. It depends solely on the dependency graph.

$$\beta_3 \in \{\text{arbitrary, nil}\}$$

Graham et al. [GLLK79] and Veltman et al. [VLL90] have used similar notational characterizations.

3.3.1 Independent Tasks

Assignments of independent tasks are characterized by $\mathcal{W}$ = (arbitrary, nil, nil). Let ω and ω' denote makespans of any two work-greedy assignments. Graham [Gra76] proved that

$$\frac{\omega'}{\omega} \leq 1 + (m-1)\frac{\max_i \tau_i}{\sum \tau_i}$$

It can be readily seen that for large values of m, this bound is not tight. In particular it is known that as $m \rightarrow \infty$, ω'/ω should reach unity. The following theorem presents an improved bound that more accurately reflects the behaviour of work-greedy assignment algorithms for large values of m.

Let $\hat{\tau}$ be the execution time of the longest task, i.e. $\max_i \tau_i$; and let $\pi = \sum \tau_i/\hat{\tau}$. Then we have

Theorem 3.2.

$$\omega'/\omega \leq 1 + (m-1)/\pi \quad \text{if } m < \pi$$
$$\omega'/\omega \leq 1 + (\pi-1)/m \quad \text{if } m \geq \pi$$

where ω is the length of the optimal makespan, that is not necessarily work-greedy; and ω' is the makespan of any arbitrary work-greedy assignment.

Proof. Consider a work-greedy assignment of makespan ω'. Let the last task to finish be T_z (i.e. T_z finishes execution at the instant ω'). The rule of work-greedy assignments dictates that no processor can remain idle before the instant $\omega' - \tau_z$ and that at least one processor will be busy until the instant ω'. Hence,

$$\sum_i \tau_i \geq (m-1)(\omega' - \tau_z) + \omega'$$

Since $\tau_z \leq \hat{\tau} \ \forall z \ (z = 1 \ldots n)$,

$$\sum_i \tau_i \geq (m-1)(\omega' - \hat{\tau}) + \omega'$$

This gives

$$\omega' \leq \frac{\sum \tau_i}{m} + \frac{(m-1)\hat{\tau}}{m} \tag{3.3.1}$$

For *any* (and thus, the optimal) assignment with a makespan of ω, the following inequality holds true:

$$\omega \geq \max\left[\frac{\sum \tau_i}{m}, \hat{\tau}\right] \tag{3.3.2}$$

When $\sum \tau_i / m \geq \hat{\tau}$, from (3.3.1) and (3.3.2) we get the bound:

$$\frac{\omega'}{\omega} \leq 1 + \frac{m-1}{\pi} \tag{3.3.3}$$

When $\hat{\tau} \geq \sum \tau_i / m$, from (3.3.1) and (3.3.2) we get the bound:

$$\frac{\omega'}{\omega} \leq 1 + \frac{\pi - 1}{m} \tag{3.3.4}$$

Both (3.3.3) and (3.3.4) always hold true. However, when $m \geq \pi$ the bound of (3.3.4) is tighter, otherwise the bound of (3.3.3) is tighter.

□

Examples can be constructed to establish that the bound of theorem 3.2 is the best possible when $m = \pi$. See [Gra76] example 3.

From (3.3.3) and (3.3.4) we get the following loose bound:

$$\frac{\omega'}{\omega} \leq 2 \tag{3.3.5}$$

When $m \geq n$, each processor is assigned at most one task. In this case, the makespan is the length of the longest task. Thus, the makespan remains constant. That is,

$$\frac{\omega'}{\omega} = 1 \quad \text{if } m \geq n$$

It may be possible to find tight bounds that are independent of π for some special cases. The following theorem presents such a bound for the case $m = n - 1$.

Theorem 3.3. If $m = n - 1$, then $\omega'/\omega \leq 3/2$. Furthermore, there exist task sets for which ω'/ω equals the above bound.

Proof. Without loss of generality, it can be assumed that

$$\tau_1 \leq \tau_2 \ldots\ldots \leq \tau_{n-1} \leq \tau_n.$$

Since $m = n - 1$, all the tasks except one will start execution at instant 0. Let the last task to be executed (i.e. the task that starts execution at an instant > 0) be T_i. T_i will be assigned to a processor that is assigned the shortest task in the set $\mathbf{T} - \{T_i\}$. Let the makespan of this assignment be ω_i.

Now, if T_1 is the last task to be executed, it will be assigned to the processor that is assigned T_2, for T_2 is the first task to finish out of the tasks already assigned. Therefore,

$$\omega_1 = \max[\tau_n, \tau_1 + \tau_2] \tag{3.3.6}$$

If T_i $(i > 1)$ is the last task to be assigned, then T_1 will be the first task to finish execution. T_i will be assigned to the processor that is assigned T_1. Therefore,

$$\omega_i = \max[\tau_n, \tau_1 + \tau_i] \tag{3.3.7}$$

From (3.3.6) and (3.3.7) it is observed that,

1. $\omega_1 = \omega_2$

2. $\omega_n = \tau_1 + \tau_n$

3. $\omega_{n-1} \geq \omega_{n-2} \ldots\ldots \geq \omega_2$ (since $\tau_j \geq \tau_{j-1} \quad \forall j > 1$)

4. $\omega_n \geq \omega_{n-1}$ (since $\omega_n = \tau_1 + \tau_n$ and $\tau_n \geq \tau_{n-1}$)

From these observations we obtain

$$\omega_n \geq \omega_{n-1} \geq \omega_{n-2} \ldots\ldots \geq \omega_2 = \omega_1 \tag{3.3.8}$$

The best-case makespan is thus ω_1, and the worst-case makespan is ω_n. Hence,

$$\frac{\omega_{worst}}{\omega_{best}} = \frac{\omega_n}{\omega_1} = \frac{\tau_1 + \tau_n}{\max[\tau_n, \tau_1 + \tau_2]} \tag{3.3.9}$$

We now find the maximum possible value of ω_n/ω_1.

When τ_1 and τ_2 are relatively small such that $\tau_1 + \tau_2 < \tau_n$, the denominator of the ratio ω_n/ω_1 is τ_n. Now let us increase τ_1 keeping τ_n constant so as to increase ω_n/ω_1. Let $\tau_n = 2k$ ($k > 0$) and $\tau_2 = \tau_1 + 2\epsilon$ ($\epsilon \geq 0$). The numerator of ω_n/ω_1 increases but the denominator remains constant at $2k$. However, when τ_1 becomes greater than $k - \epsilon$ (i.e. $\tau_1 + \tau_2$ becomes greater than τ_n) the denominator starts to increase. Now two cases need to be considered:

1. $\tau_1 < k - \epsilon$. Let $\tau_1 = k - \epsilon - \delta$ ($0 \leq \delta \leq k - \epsilon$). By substituting for τ_1 in (3.3.9),

$$\frac{\omega_n}{\omega_1} = \frac{3k - \epsilon - \delta}{2k}$$

2. $\tau_1 > k - \epsilon$. Let $\tau_1 = k - \epsilon + \delta$ ($0 \leq \delta \leq k + \epsilon$). By substituting for τ_1 in (3.3.9),

$$\frac{\omega_n}{\omega_1} = \frac{3k - \epsilon + \delta}{2k + 2\delta}$$

Since we are interested in $(\omega_n/\omega_1)_{max}$, we let $\epsilon \to 0$. Thus we have

$$\frac{\omega_n}{\omega_1} = \frac{3k - \delta}{2k} \quad \text{if } \tau_1 < k \tag{3.3.10}$$

$$\frac{\omega_n}{\omega_1} = \frac{3k + \delta}{2k + 2\delta} \quad \text{if } \tau_1 > k \tag{3.3.11}$$

From (3.3.10) and (3.3.11) we get

$$\left(\frac{\omega_n}{\omega_1}\right)_{max} = \frac{3}{2}$$

If ω and ω' are any two makespans, then

$$\frac{\omega'}{\omega} \leq \left(\frac{\omega_{worst}}{\omega_{best}}\right)_{max}$$

Thus,

$$\frac{\omega'}{\omega} \leq \frac{3}{2} \tag{3.3.12}$$

As an example, consider a set of three tasks for which execution times are given by $\tau_1 = \tau_2 = \tau_3/2 = 1$. With two processors, the best-case makespan is 2, and the worst-case makespan is 3. This example establishes that the bound stated in theorem 3.3 is the best possible.

□

3.3.2 Dependency Graphs with Zero Communication Times

Assignments of dependency graphs with zero communication times are characterized by $\mathcal{W}$ = (arbitrary, nil, arbitrary).

Let ω and ω' denote makespans of any two work-greedy assignments. Graham [Gra69,Gra76] proved that

$$\frac{\omega'}{\omega} \leq 2 - 1/m$$

The following theorem improves this bound by incorporating into it the so-called degree of average software parallelism. Informally, the degree of average software parallelism is a measure of parallelism in a task dependency graph.

Let τ^* be the execution time of the longest chain of the dependency graph; and let $\pi = \sum \tau_i/\tau^*$. Then we have

Theorem 3.4.

$$\omega'/\omega \leq 1 + (m-1)/\pi \quad \text{if } m < \pi$$
$$\omega'/\omega \leq 1 + (\pi-1)/m \quad \text{if } m \geq \pi$$

where ω is the length of the optimal makespan, that is not necessarily work-greedy; and ω' is the makespan of any arbitrary work-greedy assignment.

Proof.

For *any* (and thus, the optimal) assignment of makespan ω, the following inequality holds true:

$$\omega \geq \max\left[\frac{\sum \tau_i}{m}, \tau^*\right] \tag{3.3.13}$$

Let $\prec$ be the partial order on $\mathbf{T}$. The rule of work-greedy assignments dictates that for any arbitrary work-greedy assignment of makespan ω' there exists a chain of tasks

$$T_{c1} \prec T_{c2} \prec \ldots \prec T_{cy}$$

such that at every time instant $t \in [0, \omega']$ some T_{cj} is being executed.

Let the sum of all the processor idle times in this assignment be I. Then,

$$I \leq (m-1)\sum_{j=1}^{y} \tau_{cj} \tag{3.3.14}$$

But for any chain in an assignment, the following inequality holds true:

$$\sum_{j=1}^{y} \tau_{cj} \leq \tau^* \tag{3.3.15}$$

Now since

$$\omega' = \frac{1}{m}\left[\sum_i \tau_i + I\right]$$

using (3.3.14) and (3.3.15) we get,

$$\omega' \leq \frac{1}{m}\left[\sum_i \tau_i + (m-1)\tau^*\right] \tag{3.3.16}$$

When $\sum \tau_i/m \geq \tau^*$, from (3.3.13) and (3.3.16) we get the bound:

$$\frac{\omega'}{\omega} \leq 1 + \frac{m-1}{\pi} \tag{3.3.17}$$

When $\tau^* \geq \sum \tau_i/m$, from (3.3.13) and (3.3.16) we get the bound:

$$\frac{\omega'}{\omega} \leq 1 + \frac{\pi-1}{m} \tag{3.3.18}$$

Both (3.3.17) and (3.3.18) always hold true. However, when $m \geq \pi$ the bound of (3.3.18) is tighter, otherwise the bound of (3.3.17) is tighter.

□

π is defined to be the degree of average software parallelism. It is a lower bound on the amount of parallelism within a task dependency graph.

If $\prec$ is empty, then π becomes equal to $\sum \tau_i/\hat{\tau}$ and thus the results of theorem **3.4** and theorem 3.2 match.

Note that, from (3.3.13) and (3.3.16), we get the following loose bound:

$$\frac{\omega'}{\omega} \leq 2 \tag{3.3.19}$$

According to theorem 3.4, as $m \to \infty$, ω'/ω reaches unity (rather than 2 as Graham's bound suggests). This highlights the fact that with unlimited processing resources, any work-greedy assignment is optimal. In practical terms, a work-greedy assignment is optimal if $m \geq n$.

We can also express in terms of π a lower bound on the number of processors required to execute the task graph in the minimum possible time.

A bound on the number of processors. The number of processors required to finish executing all the tasks in the minimum possible time is bounded below by the ratio of the total execution time requirement of the tasks and the minimum

makespan [McN59]. The total execution time requirement is $\sum \tau_i$ and the minimum possible makespan is τ^*. A lower bound on the number of processors is thus given by

$$\left\lceil \frac{\sum \tau_i}{\tau^*} \right\rceil = \lceil \pi \rceil$$

That is, *any* (not necessarily work-greedy) assignment will require at least $\lceil \pi \rceil$ processors, if it is to execute the task graph in the minimum possible time.

Tighter lower bounds on the number of processors can be found in [FB73,AM90].

3.3.3 Dependency Graphs with Unit Computation and Communication Times

Assignments of dependency graphs with unit computation and unit communication times are characterized by $\mathcal{W}$ = (unit, unit, arbitrary). For this characterization, Rayward-Smith [RS87] proves the following upper bound on the makespan ω of an arbitrary work-greedy assignment:

$$\omega \leq \left(3 - \frac{2}{m}\right)\omega' - \left(1 - \frac{1}{m}\right)$$

where ω' is the makespan of the optimal assignment.

3.3.4 Dependency Graphs with Arbitrary Computation and Communication Times

Assignments of dependency graphs with arbitrary computation and arbitrary communication times are characterized by $\mathcal{W}$ = (arbitrary, arbitrary, arbitrary). The assignment schemes ETF, ERT, MH and MCP fall under this characterization.

Hwang et al. [HCAL89] and Lee et al. [LHCA88] proved bounds on the makespans of ETF and ERT. They have proved that

$$\omega' \leq \left(2 - \frac{1}{m}\right)\omega^i + C_x$$

where ω' is the makespan of the work-greedy assignment (either ETF or ERT), ω^i is the makespan of the optimal assignment *without* considering communication delays, and C_x is the communication delay along some chain in the task graph.

Expressing ω' in terms of ω^i does not reveal much. When giving a guarantee for the makespan of a certain assignment, one would want to give it in terms of the *corresponding* optimal makespan. It is more useful to give a guarantee in terms of ω, the optimal makespan *not ignoring* the communication delay.

As in theorem 3.4, the degree of average software parallelism can be incorporated into this bound so that the bound will be tighter.

Moreover, we note that the bound can be generalized for *all* the assignments characterized by $\mathcal{W}$ = (arbitrary, arbitrary, arbitrary). We thus present in the following theorem a generalized bound.

Let τ^* be the sum of execution times of tasks along the longest chain (ignoring communications) of the dependency graph and τ^+ be $\sum \tau_i$; and let $\pi = \tau^+/\tau^*$. Then we have

Theorem 3.5.

$$\frac{\omega'}{\omega} \leq 1 + \frac{(m-1)}{\pi} + m\,\frac{C_{comm}}{\tau^+} \qquad \text{if } m < \pi$$

$$\frac{\omega'}{\omega} \leq 1 + \frac{(\pi-1)}{m} + \pi\,\frac{C_{comm}}{\tau^+} \qquad \text{if } m \geq \pi$$

where ω is the length of the optimal makespan, that is not necessarily work-greedy; and ω' is the makespan of any arbitrary work-greedy assignment. C_{comm} is the maximum communication delay along some chain of tasks.

Proof.

The proof is similar to the one presented for theorem 3.4.

For *any* (and thus, the optimal) assignment of makespan ω, the following inequality holds true:

$$\omega \geq \max\left[\frac{\sum \tau_i}{m}, \tau^*\right] \tag{3.3.20}$$

Let $\prec$ be the partial order on $\mathbf{T}$. The rule of work-greedy assignments dictates that for any arbitrary work-greedy assignment of makespan ω' there exists a chain of tasks

$$T_{c1} \prec T_{c2} \prec \ldots \prec T_{cy}$$

such that at every time instant $t \in B$ some T_{cj} is being executed or is waiting for input from T_{cj-1} (that has finished executing) to start its execution. Here B is the set of all points of time in $[0, \omega']$ for which at least one processor is idle.

Let $proc(T)$ be the processor that has been assigned the task T; and let $mtt(P_i, P_j)$ be the maximum time to transfer unit information from processor P_i to processor P_j (possibly via other processors). C_{comm} is calculated as follows:

$$C_{comm} = \sum_{j=1}^{y-1} mtt(proc(T_{cj}),\ proc(T_{cj+1}))\ \ v(T_{cj},\ T_{cj+1})$$

Let the sum of all the processor idle times in this assignment be I. Then,

$$I \leq m\left(\sum_{j=1}^{y} \tau_{cj} + C_{comm}\right) - \sum_{j=1}^{y} \tau_{cj} \tag{3.3.21}$$

But for any chain in an assignment, the following inequality holds true:

$$\sum_{j=1}^{y} \tau_{cj} \leq \tau^* \tag{3.3.22}$$

Now since

$$\omega' = \frac{1}{m}\left[\tau^+ + I\right]$$

using (3.3.21) and (3.3.22) we get,

$$\omega' \leq \frac{\tau^+}{m} + \frac{(m-1)\tau^*}{m} + C_{comm} \tag{3.3.23}$$

When $\tau^+/m \geq \tau^*$, from (3.3.20) and (3.3.23) we get the bound:

$$\frac{\omega'}{\omega} \leq 1 + \frac{m-1}{\pi} + m\,\frac{C_{comm}}{\tau^+} \tag{3.3.24}$$

When $\tau^* \geq \tau^+/m$, from (3.3.20) and (3.3.23) we get the bound:

$$\frac{\omega'}{\omega} \leq 1 + \frac{\pi - 1}{m} + \pi \frac{C_{comm}}{\tau^+} \tag{3.3.25}$$

Both (3.3.24) and (3.3.25) always hold true. However, when $m \geq \pi$ the bound of (3.3.25) is tighter, otherwise the bound of (3.3.24) is tighter.

□

Construction of the chain. The set of all points in time in the interval $[0, \omega']$ is divided into two subsets A and B as follows. A is the set of points in time for which all processors are busy. B is the set of points in time for which at least one processor is idle.

Let ψ_i and ϕ_i denote respectively the start and finish times of T_i. The following algorithm constructs the chain.

1. Let the chain C be an ordered set of tasks, set to null initially.

2. $T_a \leftarrow$ a task that finishes at time ω'.

3. If $\psi_a \in B$,
   then there exists a processor which for some $\epsilon > 0$ is idle during the time interval $[\psi_a - \epsilon, \psi_a]$. This occurs only when there is a task T_b, an immediate predecessor of T_a, such that

   $$\phi_b + mtt(proc(T_a),\ proc(T_b))\ v(T_a,\ T_b) = \psi_a.$$

   Insert T_a into C, $T_a \leftarrow T_b$ and go to 3.

4. Let $u =$ l.u.b.[2] $\{x | x < \psi_a \text{ and } x \in B\}$. If u is zero, output C and stop.

[2]Least upper bound

5. Find a task T_b such that

$$\psi_b = \max_i\{\psi_i |\ T_i \text{ a predecessor of } T_a \text{ and } \psi_i < u\}.$$

   There is a sequence of tasks, T_c, $T_{j1}, \ldots T_{jz}$, such that $T_b \prec T_c \prec T_{j1} \prec \ldots \prec T_{jz} \prec T_a$. Insert T_c into C, $T_a \leftarrow T_b$ and go to 3.

The maximum time to transfer information between processors depends as well on the underlying routing strategy and the network contention. These dependencies were ignored in the proof above.

Note that the communication factor that appears in our bound is smaller than those of Hwang et al. and Lee et al. Note also that our bound is applicable to all work-greedy assignments – not just ETF and ERT.

If communication costs can be ignored, then $C_{comm} = 0$. The bounds of theorems 3.4 and 3.5 then match. Note that the value of C_{comm} depends on the assignment. Good assignments will have small values of C_{comm}. Now if

$$C^* = \max_{i,j}\left[mtt(P_i, P_j)\right] \ \max_T \left[\sum_{T_i, T_j \in \mathbf{T};\ T_j = succ(T_i)} v_{ij}\right] \quad \text{where } \mathbf{T} \text{ is any chain in } G_T,$$

then

$$C_{comm} \leq C^* \quad \text{for any chain.}$$

Thus the bounds of theorem 3.5 become

$$\frac{\omega'}{\omega} \leq 1 + \frac{(m-1)}{\pi} + m\,\frac{C^*}{\tau^+} \qquad \text{if } m < \pi \tag{3.3.26}$$

$$\frac{\omega'}{\omega} \leq 1 + \frac{(\pi-1)}{m} + \pi\,\frac{C^*}{\tau^+} \qquad \text{if } m \geq \pi \tag{3.3.27}$$

These bounds are *not* assignment-dependent.

EX BIBL. UNIV. EDINBURGH

3.3.5 Implications of the Bounds on Makespans

The hardware parallelism, m, and the degree of average software parallelism, π, have a symmetric relation in the bounds of theorems 3.2, 3.4 and 3.5. When $m > \pi$,

the makespan may be limited by software 'sequentialism'; and when $\pi > m$ the makespan may be limited by hardware inadequacy. Note that, since π is only a lower bound on software parallelism, we can find cases where $m \geq \pi$ and yet the makespan is limited by hardware inadequacy.

The loose bounds of (3.3.5) and (3.3.19) suggest that, if communication costs can be ignored, the maximum speedup an assignment scheme can achieve is no more than 2. In other words, no assignment scheme can be worse than the optimal scheme by more than a factor of two. The bound by Rayward-Smith suggests that, if communication times are assumed to be unitary and if the computation times are also unitary, this factor of degradation is no more than 3. Thus it is seen that *any* work-greedy assignment scheme can be used for the assignment of

1. independent tasks
2. dependency graphs with zero communication times, and
3. dependency graphs with unit computation and communication times

and still a performance not worse than a small constant factor would be guaranteed.

However, if communication costs are arbitrary, the performance can degrade considerably with bad assignment schemes. In this case, from (3.3.26) and (3.3.27), we have the following loose bound:

$$\frac{\omega'}{\omega} \leq 2 + \lambda, \qquad \text{where } \lambda = \min(m, \pi)\, \frac{C^*}{\tau^+} = \frac{C^*}{\tau^+ / \min(m, \pi)}.$$

λ signifies the communication to computation ratio along the critical path of the (arbitrary) assignment. Bad assignments will have large values of λ and thus they will have a poor performance compared to the optimal assignment. For instance, a work-greedy assignment scheme that ignores the communication costs when the dependency graph *does* have communication requirements may yield a large value of λ.

3.4 A Non-Work-Greedy Scheme for Assignment

Work-greedy assignment schemes try not to leave a processor idle if there is a task the processor can execute. Tasks are assigned to the processors that can execute them the earliest. Ensuring this involves extra search. Yet these schemes permit finding assignments with a guarantee.

When communication costs are taken into account, work-greedy assignment schemes lose two of their important characteristics. That is, with arbitrary communication costs,

- there is no guarantee that a processor will not idle when there is a task it could execute (see section 2.1.2), and
- a work-greedy assignment can be worse than the optimal assignment by a large factor (determined by the communication costs along some path in the dependency graph); hence, a bad work-greedy scheme could generate very poor assignments.

There is thus a case to examine an assignment scheme that moves away from the work-greedy heuristics and to see how well this scheme performs compared to the work-greedy assignments. To this end, this section proposes a simple non-work-greedy assignment scheme which is based on satisfying the desirable properties stated in 2.4.2. The scheme is easy to implement and has a time-complexity linear in the number of tasks and task graph edges.

If the goal of an assignment scheme is the minimization of the makespan, then it is desirable that the scheme should possess the following properties:

DP1. Assignment of independent tasks to different processors.

DP2. Assignment of dependent tasks to the same processor.

The first desirable property DP1 ensures that the parallelism available in the task graph is fully exploited. An assignment can possess this property only if sufficient processors exist. Intuitively, the maximum number of processors needed will be equal to the size of the largest set of independent tasks.

The second desirable property DP2 helps to minimize the communication cost. But, not all dependent tasks can be assigned to the same processor. The reason is two fold: (1) two independent tasks may have common dependencies (i.e. they may share a successor or predecessor), and (2) per-processor memory may be limited.

We move away from the work-greedy heuristics and propose an assignment scheme whose heuristics is based on satisfying the properties DP1 and DP2. The scheme assumes, as the work-greedy schemes do, that the parameters of the task graph are known at compile time.

Work-greedy assignment schemes find the start and finish times of the tasks as they proceed to find the assignment. However, due to network contention and routing decisions, these times cannot be predicted correctly at the time of assignment. The assignment schemes that claim to take network contention into account have their own adaptive routing techniques embedded into their assignment algorithms. By doing so, they map the message transfers to certain processor interconnections (or links). If this mapping is not preserved during run-time, for instance, by using a routing scheme other than the one embedded in the assignment algorithm, the start and finish times of the tasks predicted by the assignment scheme may be different from the actual times. Thus, it is noted that the start and finish times of tasks make sense only as far as determining the partitions of the tasks. Once the partitions are determined, each processor will need to perform a local scheduling: taking ready tasks one by one from the task partition assigned to the processor and

executing them. Our proposed assignment scheme, therefore, does not attempt to find the start and finish times of the tasks; it finds only partitions.

3.4.1 DFBN: The New Scheme

The scheme uses a combination of the familiar depth-first and breadth-first search algorithms to arrive at an assignment. This technique is called depth-first-breadth-next (DFBN) search.

DFBN searches the graph as follows. All start vertices of the graph are entered in a queue. An unvisited vertex v is taken from the queue and a function *visit* is called for v. This is repeated until the queue is empty. The function *visit* marks v visited, selects an unvisited successor vertex w for visiting, appends all other successors to the queue, and finally calls itself recursively for vertex w. *Visit* returns when it is called for a vertex such that its successor vertices have been visited. The traversal of the graph follows a depth-first and breadth-next order. Thus the name DFBN.

The DFBN technique is used in generating assignments for dependency graphs. Note that any single call from DFBN to *visit* for a vertex v marks a chain of vertices originating from v visited. Every call from DFBN to *visit* thus results in a traversal of a new chain. Now if the vertices represent tasks and edges represent dependencies, then all the vertices of a chain are *dependent*. Thus they could be assigned to a single processor. Since all new chains *could* be *independent* of each other, these chains are assigned to different processors. See figure 3–4 for the full algorithm.

```
procedure FormAssignment()
   Initialize taskQ with all the start vertices in it
   while taskQ not empty do
      task = pop(taskQ)
      if task is not assigned then
         processor = GetProcessor(task)
         Assign(task, processor)
         PutProcessor(processor)
      endif
   endwhile
endproc

procedure Assign(task, processor)
   Mark task assigned to processor
   if there are no successors of task then
      return
   else
      NewTask = first unassigned successor of task
   endif
   forall other unassigned successors of task do
      Push(successor, taskQ)
   endfor
   Assign(NewTask, processor)
endproc
```

Figure 3–4: The assignment scheme DFBN

Procedure *FormAssignment* performs a DFBN search. The function *visit* has been replaced by a function *Assign* that takes as its parameter a processor in addition to a task vertex. Every call from *FormAssignment* passes a new processor to *Assign*. A new processor is returned by a function *GetProcessor*.

The function *GetProcessor* could simply select the processor with minimum load, where the load of a processor is defined to be the sum of the execution times of all the tasks that had been assigned to the processor. The load of a processor does not

reflect processor idle times due to communication events. Processors can be kept in a balanced binary tree [HS78] sorted by their load. The function *GetProcessor* would remove a processor from the tree; and the function *PutProcessor* would add a processor to it.

This approach of selecting processors according to their loads has a major drawback. It can use more processors than necessary, if sufficient processors exist. Consider the task graph of figure 3–5. This graph will be assigned to three processors, if processors are available. For instance, in a three processor system with processors P_0, P_1 and P_2, a possible assignment is

$$\{T_0, T_2, T_3, T_4\} \mapsto P_0; \quad \{T_1\} \mapsto P_1; \quad \{T_5\} \mapsto P_2;$$

However, two processors are sufficient for executing this task graph in minimal time.

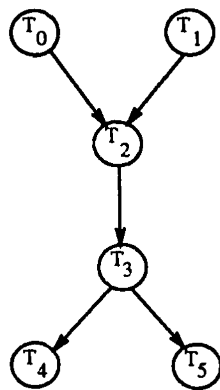


Figure 3–5: An example task graph

We get around this problem by letting *GetProcessor* choose a processor that has a *FreeInstant* just below the latest start time (LST) of the task needing a new processor. $FreeInstant_k$ of a processor P_k signifies the time instant at which P_k can finish executing the tasks assigned to it. If the *FreeInstants* of the processors are larger than the latest start time of the task, then the processor with the minimum loading is used. The latest start time, rather than the earliest start time, of the task is used so that the execution of non-critical tasks can be delayed as much as possible. As before, processors can be kept in a balanced binary tree for efficient deletion and insertion by *GetProcessor* and *PutProcessor*.

The *LST* can be estimated in time $\Theta(n + e)$ prior to the assignment. Its exact value, however, depends on the assignment that is yet to be determined. The *LST* can also be recalculated during the assignment procedure using the partial assignment (so that it can be closer to the exact value), but this will involve additional time-complexity.

Now consider again the task graph of figure 3–5. Under the second approach of processor selection, the task graph will be assigned to just two processors. The assignment, in this case, would be

$$\{T_0, T_2, T_3, T_4\} \mapsto P_0 \quad \{T_1, T_5\} \mapsto P_1.$$

If a processor does not have enough local memory to hold all the tasks belonging to a chain, function *Assign* could request new processors through *GetProcessor* and split the chain up between these processors.

When there is a need for arbitration, task and processor priorities are used. The need for arbitrating tasks arises when the function *Assign* scans through the list of successor tasks and when the start tasks are added to the *taskQ*. The need for arbitrating processors arises when there are ties in function *GetProcessor*.

3.4.2 Processor Ordering

To order the processors, a 'most capable' processor is first chosen; this processor could be the one with the maximum computational and communication capability. The other processors are arranged in an ascending order of 'distance' from the 'most capable' processor. The 'distance' between two directly connected processors P_i and P_j is defined to be $1/c_{ij}$. (Thus, two processors connected by a high capacity link will be 'closer' to each other than those connected by a low capacity link.) For processors that are not directly connected, the shortest 'distance' is found via other processors. To break ties, processors' computational and communication capabilities can be used.

3.4.3 Task Ordering

In finding a task ordering, one would want to give top priorities to the critical tasks. To determine these critical tasks, inter-task communication times need to be known. But, these communication times will not be known before the assignment is done. One can only have a guess of the critical path. A poor guess may deprive the deserving tasks of top priority and this may result in a poor assignment. Thus, instead of giving *some* tasks *absolute* priority, it is decided to give priorities to *all* the tasks depending on their *critical factors*. In other words, there will be no explicit discrimination between critical and non-critical tasks. The ***critical factor*** of a task T_q is defined as follows:

$$CF_q = \max_{i=1,n}[LCT_i - ECT_i] - (LCT_q - ECT_q)$$

Here ECT_q and LCT_q are the earliest and latest completion times of the task T_q. ($LCT_q - ECT_q$ is sometimes known as the *completion interval* of task T_q.) It should be noted that the values ECT and LCT can only be estimates. Exact values cannot be calculated before the assignment is done.

As has been noted in 3.1, a task's priority depends on other factors as well. Thus the overall priority of a task is expressed by a weighted sum of the individual priorities based on these factors as well as the critical factor. However, maximum weight is given to the critical factor. One can therefore hope that the deserving tasks will get at least top range priorities, if not the topmost priority.

If the set of successors of a task T is denoted by $succ(T)$, the priority p_i of a task T_i can be expressed as:

$$p_i = w_0(CF_i)^{\alpha} + w_1\tau_i + w_2 \sum_{T_j \in succ(T_i)} v_{ij} + w_3 \sum_{T_j \in succ(T_i)} 1 + w_4 \sum_{T_j \in succ(T_i)} \tau_j + w_5 s_i \quad (3.4.1)$$

where w's denote the weights. Prominence is given to large critical factors by raising CF_i to some power $\alpha \geq 1$. Calculation of the priorities of all the tasks can be performed in $\Theta(n + e)$ time.

Polychronopoulos and Banerjee use a similar scheme to assign priorities to tasks [PB87].

3.4.4 Time-complexity of DFBN

The total execution time of the **forall** loop in function *Assign* is $\Theta(e)$. *Assign* is called n times. *GetProcessor* and *PutProcessor* have an execution time of $\Theta(\log m)$ and are called at most n times along with *Assign*. Here n is the number of tasks, m is the number of processors and e is the number of edges in the task graph G_T. The initialization of the balanced binary tree takes time $\Theta(m. \log m)$. The time-complexity of the algorithm is thus $\Theta((n+m). \log m + e)$.

Note that the time-complexity is linear in the number of nodes and edges of the task graph. Therefore, DFBN will be a good choice for those applications that have a large number of tasks.

3.4.5 Performance Guarantee

DFBN is a single-step non-work-greedy assignment scheme. Given a task graph and a processor graph, it produces the assignment in a single step involving a very low time-complexity. The desirable properties that DFBN aims to satisfy try to exploit the parallelism visible in the graph whilst reducing the communication costs.

The makespan of an assignment can never be less than the execution time of the critical path in the task graph. The tasks belonging to the critical path are dependent. DFBN tries to assign the tasks of the critical path to the same processor, thereby reducing the inter-task communication cost within the execution of the critical path.

Since DFBN does not predict *when* a task must be executed (as all the work-greedy assignments do), it is hard to prove any analytical bound on the makespan of an assignment generated by DFBN.

Nevertheless, there is one special case for which an analytical performance guarantee can be proved for DFBN. If the processor topology is a completely connected graph with an unbounded number of processors having the same communication costs between any two of them, then DFBN generates a *linear clustering* of a task graph. In a linear clustering no two concurrently executable tasks will be in the same partition. Gerasoulis et al. prove that the makespan of any such linear clustering is within a factor of two of the optimal makespan, if communication costs are small compared to the computation costs [GVY90]. However, in a real parallel processor system where the number of processors are bounded and the inter-processor communication costs may be arbitrary, this guarantee does not hold.

Chapter 6 thus attempts to provide some experimental performance results.

3.5 Summary

The impact of task ordering on the makespan is proved and the factors upon which task ordering should depend are discussed. Tasks with long execution times, task involving large communication times, tasks with large numbers of successors, tasks with long-length successors and tasks with large memory requirements are identified to be those that need high priority in a task ordering.

The heuristics most of the current assignment schemes use is based on satisfying the following rule of thumb: keeping the processors busy leads to a 'good' assignment. Such schemes are said to be work-greedy. Work-greedy assignments are important since most of them provide a solution with a guarantee: it is proved that, when communication costs can be ignored, *any* work-greedy assignment would be

close to the optimal assignment by no more than a small constant factor. It is also proved that, should the communication costs be taken into account, this factor may no longer be small. With communication costs, a work-greedy assignment can perform worse than the optimal assignment by a large factor. This factor depends on the communication costs along some path in the task graph.

A non-work-greedy assignment scheme, called DFBN, is proposed. Its heuristics is based on satisfying two desirable properties: assigning independent tasks to different processors, and assigning dependent tasks to the same processor. The time-complexity of DFBN is at least an order less compared to the work-greedy schemes. However, there is no analytical performance guarantee for the assignments generated by DFBN.

Performance assessment of these assignment schemes is the goal of the remainder of this thesis. Performance of a parallel system depends on the architecture, the program, the assignment scheme and the routing strategy. We develop a generic modelling approach that lets us specify and model these parameters. We then use this approach to simulate program execution on some processor topologies under different assignment schemes. These simulations aid the performance assessment of the assignment schemes.

The next two chapters deal with the development of a generic modelling approach. The development of such an approach requires the following.

1. A representation scheme based on an abstraction level that integrates most of the possible architectural schemes.

2. Representing the program (or software) in an architecture independent way.

3. Providing the means to specify the assignment scheme and the routing strategies.

The next chapter proposes a representation scheme for parallel architectures. A generic modelling approach, based on this representation scheme, is presented in chapter 5. Performance assessment of assignment schemes is the theme of chapter 6.

Chapter 4

A Structural Framework for the Representation of Parallel Architectures

The problem of representation and classification arises when an area of study involves many different objects. Representation describes an object according to some meaningful rules. Classification partitions the objects into a set of classes. The primary goal of having a representation is to describe the functionalities of these objects; and the goal of having a classification is to provide a platform to compare and contrast the functionalities of the objects.

A good representation scheme is an aid in learning and modelling the behaviour of the objects under study. It is our interest to model parallel architectures in a generic way. To this end, this chapter develops a structural framework for representing parallel architectures.

With the proliferation of different parallel architectures, the distinction between representation and classification has become thin. In fact, the two terms have been used interchangeably [Das90]. Here we critically review some of the architectural classification schemes to date and build upon them a representation scheme

suitable for modelling. This representation scheme becomes an integral part of the modelling environment to be developed in chapter 5.

4.1 A Survey of Some Architectural Classification Schemes

The sheer diversity of parallel architectures makes it difficult to represent them in a unified framework. Nevertheless there have been attempts to approach this problem and classify architectures in interesting and useful ways [Fly72,Alm85, Hoc85,Hoc87,Ski88,Dun90,Das90,Dad91]. This section presents a critical review of some of these classification schemes.

4.1.1 Flynn's Scheme

The most popular classification of architectures is due to Flynn [Fly72]. His classification is based upon streams of instructions and data. Depending on the number of these streams, architectures are categorized as SISD, SIMD, MISD and MIMD.

SISD: Single Instruction stream, Single Data stream machines. The conventional von Neumann machines come under this category.

SIMD: Single Instruction stream, Multiple Data stream machines. Multiple processors simultaneously execute the same instruction on different data. Array processors come under this category.

MISD: Multiple Instruction stream, Single Data stream machines. Multiple processors simultaneously execute different instructions on the same datum. Decoupled architectures come under this category [BPTS91].

MIMD: Multiple Instruction stream, Multiple Data stream machines. Multiple processors asynchronously execute different instructions on different data items.

4.1.2 Hockney's Scheme

The main drawback of Flynn's classification is that it is too broad to describe any realistic architecture. For example, Flynn's classification fails to discriminate between the various MIMD architectures that now proliferate. There is no distinction between a shared-memory machine and a message-passing machine in Flynn's classification. They both come under the MIMD class. Hockney [Hoc85], thus, provides a structural classification scheme for MIMD architectures (Figure 4–1). In the top level, MIMD architectures are divided into *switched* systems and *networked* systems.

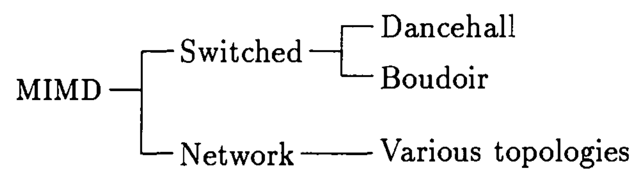


Figure 4–1: Hockney's classification

A switched system is one where "there is an identifiable and separate switch unit that connects together a number of processors and memory modules". One can view a switch as a shared set of interconnections. In general, switches are complex and may involve several stages of interconnect.

These switched systems are sub-divided depending on the way the processors, memory modules and the switch unit are organized. In the *dancehall* configuration, the processors take up one side of the switch unit and the memory modules take up the other side. In the *boudoir* configuration, processors are linked to their own local memory modules and the switch unit is used to connect the processors together. The dancehall configuration represents most shared-memory systems and

the boudoir configuration represents the distributed-memory or message-passing systems (Figure 4–2) [Alm85].

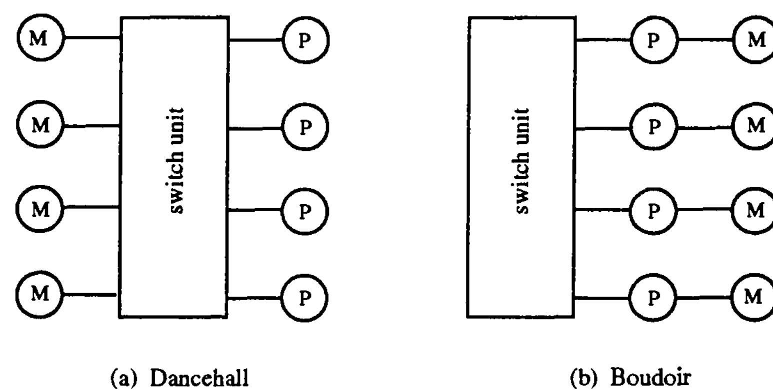


Figure 4–2: Shared-memory and distributed-memory systems

Both shared and distributed memory systems are further divided according to the type of switch unit: cross-bar, multistage or bus.

In a networked system "a number of PEs are connected together into a network with an identifiable topology". Here a PE is a processor connected to its own local memory. As one would expect, networked systems are further divided according to their topology — mesh, cube, tree, etc. The so-called reconfigurable networks, in which the interconnection pattern itself can be changed, also come under the networked category.

4.1.3 Skillicorn's Scheme

Although interesting, Hockney's classification is limited to the MIMD class. Also, it fails to categorize novel architectures such as dataflow or graph reduction machines that fall under the MIMD paradigm. Skillicorn shows that by separating the instruction-oriented and data-oriented functions of processors and memory modules, it is possible to arrive at a more general and discriminating classification of architectures [Ski88].

At the highest, most abstract level, architectures are classified with respect to the number of instruction and data processors, instruction and data memory units, and the way they are connected. The processors and memory units are collectively termed functional units.

Instruction processors (**IP**) are responsible for fetching and decoding the instructions; data processors (**DP**) are responsible for fetching the required data and executing the instructions.

A memory hierarchy is an 'intelligent' storage device that provides the instruction or data requested by the processor. The ideal von Neumann memory unit does not differentiate instructions from data. However, almost all real machines do differentiate them, at least from the user's point of view. Thus, memory units are divided into instruction memories (IM) and data memories (DM). In other words, a distinction between memory hierarchies is provided at the abstract machine level. However, in a real implementation the separation normally occurs only at the top level of the memory hierarchy and within the virtual memory system.

The interconnections represent both shared and dedicated connections (cf. switched and networked systems in Hockney's classification). Skillicorn names the interconnections *switches*. Four types of switches are identified:

1-to-1: A single functional unit is connected to another single functional unit.

n-to-n: The i-th unit of one set of functional units is connected to the i-th unit of another set. This is simply a 1-to-1 switch replicated n times.

1-to-n: A single functional unit is connected to all the n units of a set of functional units.

n-**by**-*n*: Every unit of one set of functional units communicates with every unit of the second set and vice versa.

An abstract machine is constructed by wiring the functional units with switches. Skillicorn identified 28 classes of architectures (see table 4–1) depending on the organization of the functional units and the switches.

Class	IPs	DPs	IP-DP	IP-IM	DP-DM	DP-DP	Name
3	0	n	none	none	n-n	n×n	Distributed memory Reduct/Dataflow
4	0	n	none	none	n×n	none	Shared memory Reduct/Dataflow
6	1	1	1-1	1-1	1-1	none	Von Neumann uniprocessor
8	1	n	1-n	1-1	n-n	n×n	Distributed memory array processor
9	1	n	1-n	1-1	n×n	none	Shared memory array processor
14	n	n	n-n	n-n	n-n	n×n	Distributed memory von Neumann
15	n	n	n-n	n-n	n×n	none	Shared memory von Neumann

Table 4–1: Some possible architectures under Skillicorn's classification

The lowest level in Skillicorn's classification is based on the state machine view of the functional units. This level is used to distinguish variants more precisely. For instance, the sequencing and ordering of operations performed by the instruction and data processors can be expressed by a state diagram. These diagrams help to distinguish between simple, pipelined and parallel units.

Flynn's classification is based upon how a machine relates the instructions to the data being processed. Hockney's classification is based on the structure of the machines. Skillicorn's classification is based on both.

4.1.4 Dasgupta's Scheme

Dasgupta extends Skillicorn's classification in certain ways [Das90]. He identifies seven basic functional units called *atoms*:

- iM – an interleaved memory unit,
- sM – a simple memory unit,
- C – a cache unit,
- pI – a pipelined instruction processor,
- sI – a simple (or non-pipelined) instruction processor,
- pX – a pipelined execution processor, and
- sX – a simple (or non-pipelined) execution processor.

The instruction processor is functionally identical to Skillicorn's IP. Similarly, the execution processor is functionally identical to Skillicorn's DP. Distinction is made to differentiate simple processors from pipelined processors. Note that in Skillicorn's scheme such distinctions are made only at the low level.

There is no distinction made between instruction and data memories. As has been noted earlier, Skillicorn provided this distinction since most of the programming environments enforce a separation between instruction and data. However, in real implementations of memory units instruction and data storage are differentiated only at the top level of the hierarchy, for instance, at the cache units. Thus, Dasgupta chose not to differentiate instruction and data memories. He rather chooses to differentiate caches from memories.

Using formulæ inspired by chemical notation, Dasgupta presents a new approach to the classification of architectures. There are two basic operators that operate upon atoms. A subscript operator replicates an atom. For instance, iM_3 and C_8 denote three and eight atoms of interleaved memory and cache respectively. The subscript could either be a positive integer constant or an integer-valued variable.

Replication represents a potential for multiple atoms to be used in parallel. A replicated atom is called an atomic radical. An atom can be viewed as a monoatomic radical.

The second operator is the dot operator that links two atomic radicals together. The combination is called a complex, or non-atomic, radical. Examples are C.sX, $(C.pI)_n$ and $C.(C_2.pI)_2$. The complex radicals are enclosed in parentheses when there is a need to replicate them.

Using these two operators, an architecture is expressed as a formula. A cache-processor (CP) is a combination of a cache radical and a processor radical. An example is $(C.pI)_n$. A memory-cache processor (MCP) is a combination of a memory radical and a cache-processor radical. An example is $(iM)_m.(C.pI)_n$. An I-molecule is an MCP-radical that represents a complete instruction preparation system. Similarly, an X-molecule is a complete instruction execution subsystem. Finally, a macromolecule is a single or replicated combination of an I-molecule and an X-molecule; it represents the complete architecture. The symbol string describing a particular radical or molecule is referred to as a formula.

Given a formula, Dasgupta provides construction rules. If W and Z denote two radicals, then W_n, W.Z, $W_n.Z$, $W.Z_m$ and $W_n.Z_m$ describe all the possible structures that can be constructed. W_n represents radical W replicated n times. W.Z represents two radicals W and Z connected by a simple link. $W_n.Z$ represents W replicated n times and connected to Z by a divergent link (Figure 4–3). $W.Z_m$ too describes a similar structure. $W_n.Z_m$ produces a bidivergent link structure. Note that the structures of figure 4–3 do not say whether the connections involved are dedicated or shared.

The path through which atoms combine together to produce the final macromolecule forms a hierarchy in Dasgupta's classification. For instance, the top level — the most abstract level — differentiates processor radicals, the second differentiates cache-processor radicals and the final level discriminates between macromolecules.

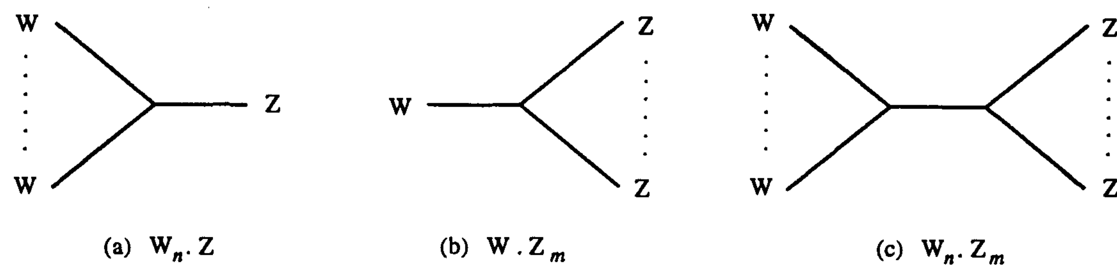


Figure 4–3: Structural diagrams for Dasgupta's formulæ

Dasgupta's scheme, in fact, *names* architectures rather than grouping them, and thus it is more of a representation scheme than a classification scheme.

4.2 A Representation Scheme for Parallel Architectures

This section proposes a representation scheme for parallel architectures. The architectural classification schemes of Skillicorn and Dasgupta form the basis of the representation scheme. In both Skillicorn's and Dasgupta's schemes, an architecture is described by a set of functional units interconnected by switches. Depending on the organization of the functional units and the switches in an architecture, the architecture is classified into some groups.

We do not intend to classify or group architectures. Rather, we are interested in describing or representing an architecture according to some meaningful rules so as to model and simulate the behaviour of the architecture. As Skillicorn and Dasgupta do, we too find a set of functional units, or *atoms*, that form the basic units of any architecture. We then use these atoms to build and represent an architecture of our choosing.

4.2.1 A Refined Set of Atoms

The main drawback of both Skillicorn's and Dasgupta's classification schemes is in the representation of interconnections. There is no clear distinction made between shared and dedicated connections. The scheme we propose here separates interconnections, however trivial they may be, from the processing and memory elements, and distinguishes between shared connections and dedicated ones. The interconnections themselves can be functional units. Thus, a refined set of atoms that serve as building blocks in constructing an architecture is obtained.

An atom is a black-box that is free to determine its internal functioning orthogonal to other atoms. It has an arbitrary number of ports through which it can be linked to the ports of other atoms. A port is simply a socket that is used to plug into another port. The only way the atoms interact with one another is via these ports.

Five basic atoms are identified:

P – a processor,

C – a cache unit,

M – a memory unit,

D – a dedicated connect, and

S – a shared connect.

D denotes a dedicated connect that links just two atoms. Thus D is comparable to Skillicorn's **1-to-1** switch. D could either be full-duplex or be half-duplex:[1] a full-duplex link is capable of sending and receiving data in both directions simultaneously; a half-duplex link too is capable of transmitting in both directions but

[1]There also can be a simplex link that is unidirectional. But a simplex link is seldom used in isolation because the receiver can in no way communicate with the transmitter to indicate errors.

only in one direction at a time. S denotes a shared connect that links more than two atoms. The number of ports of an S atom will be equal to the number of atoms it links. S realizes a many-to-many connection. Buses, concentrators and switches are some typical S atoms. Note that the connects are also functional units — their function is to pass or route the items they receive.

Processors, caches and memories are collectively termed *non-connect* atoms in order to make a functional distinction between them and the connect atoms.

Following Skillicorn and Dasgupta, instruction oriented and data oriented functions of a processor are separated. Two processor types are thus derived: an instruction processor and an execution processor. This facilitates modelling the concurrency in instruction prefetch and execution, besides enabling abstract representation of exotic architectures.

Most of the hardware — data paths, registers, caches, etc. — and programming environments enforce a separation between instruction and data. This separation becomes thin only at the low-levels of the memory hierarchy. It is observed that an instruction processor always deals with instruction memories; and an execution processor always deals with data memories. Hence, even at the low-levels of the memory hierarchy, instruction and data memories get *logically* separated. Thus it is informative to separate instruction and data memories and caches rather than having them represented by single atoms. This facilitates a better understanding of the logical organization of the system.

Separating the instruction and data oriented functions of the processor, cache and memory units leads to a secondary set of non-connect atoms:

iP – an instruction processor,

xP – an execution processor,

iC – an instruction cache,

dC – a data cache,

iM – an instruction memory, and

dM – a data memory.

The atoms iP and xP have the same meaning and functions as the corresponding atoms IP and DP defined by Skillicorn. But iM and dM denote single memory units rather than the entire memory hierarchy. If a memory hierarchy is required, a number of memory and cache units must be connected together.

There is no distinction between pipelined and simple processors. Also, there is no distinction between interleaved and simple memory units. Pipelining and memory interleaving do increase the throughput, but they do not add a new dimension to the global abstract view. Particularly, in a parallel system there are other factors (for example, number and organization of functional units) that draw more interest than pipelining or interleaving. It is thought to be more appropriate to consider such details in a low, or concrete, level of the representation.

4.2.2 The Representation Scheme

An architecture is described by two levels: (1) the top level pertains to the way the atoms are connected, and (2) the bottom level specifies how the atoms work. For example, two architectures sharing the same topology but made of different processors will be grouped together in the top level but will be differentiated at the bottom level.

The bottom level description of the representation scheme takes the state view of every individual atom comprising the architecture rather than the *entire* architecture. This is important from a modelling point of view since each atom can have its own independent dynamic behaviour, with occasional synchronizations with other atoms directly connected to it. Thus, atoms operate locally without causing any global side-effects.

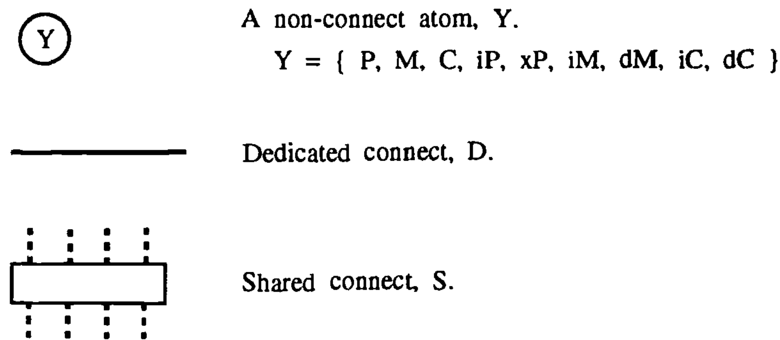
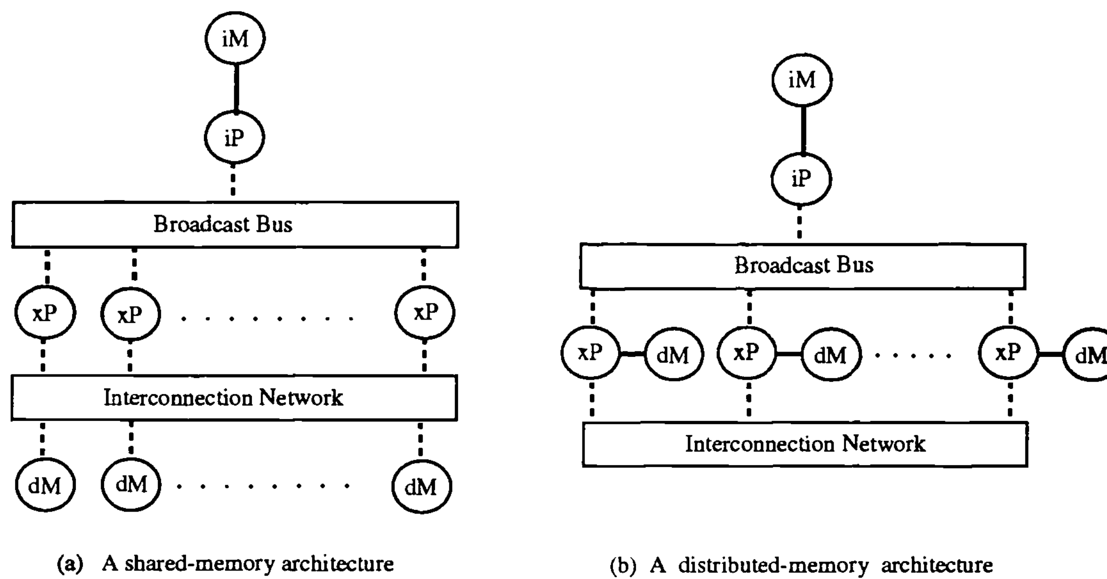


Figure 4–4: Primitives composing structural diagrams

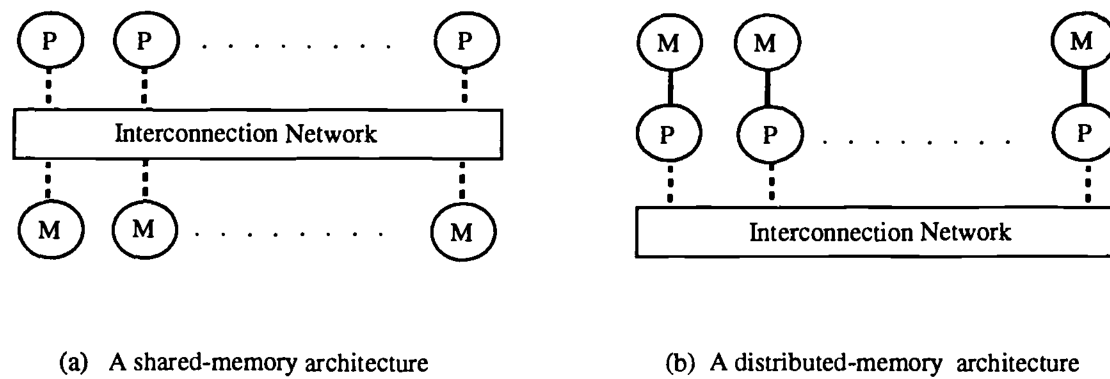
Architectures are built by connecting together an appropriately chosen set of non-connect atoms by a set of connect atoms. See figure 4–4. Structural diagrams are used in representing the architectures thus built. A structural diagram graphically displays the organization and interconnection of the atoms comprising an architecture. Use of structural diagrams permits to describe topologies. Neither Skillicorn's nor Dasgupta's scheme can describe topologies. Figures 4–5 to 4–7 depict structural diagrams of some typical architectures.



(a) A shared-memory architecture (b) A distributed-memory architecture

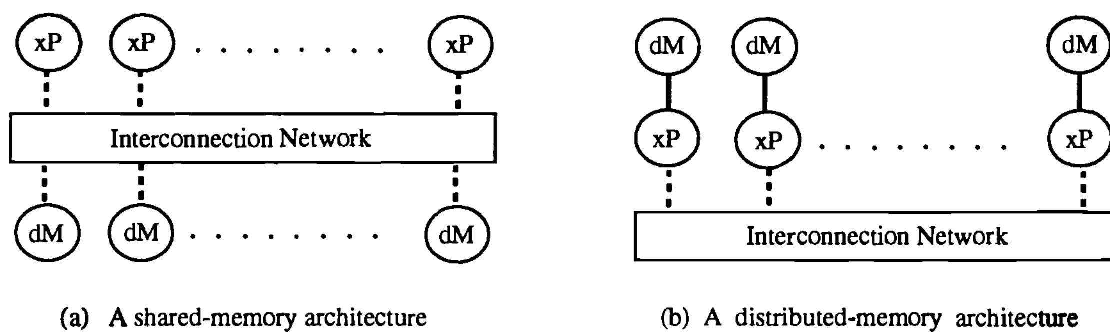
Figure 4–5: Array processors

Structural diagrams can be used to represent both the logical and the physical organizations of an architecture. The logical organization of an architecture could be represented by connecting together the appropriate non-connect atoms of the secondary set. Figure 4–8 shows the logical organization of DEC's Firefly multi-



(a) A shared-memory architecture (b) A distributed-memory architecture

Figure 4–6: Parallel von Neumann processors



(a) A shared-memory architecture (b) A distributed-memory architecture

Figure 4–7: Dataflow processors

processor workstation [TSS88]. The separation of instruction and data oriented atoms may be found only at the logical level. For instance, instruction and data memories typically share the same physical unit. Thus, the physical organization of a parallel system could be different from its logical organization. As an example, compare the logical organization of Firefly with its physical organization (figures 4–8 and 4–9). Depending on what one wants to describe or model, one can choose either of the representations.

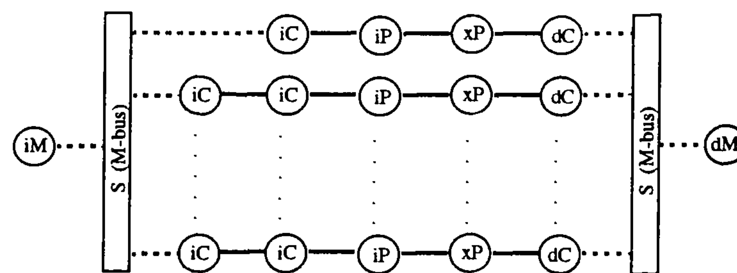


Figure 4–8: Logical organization of Firefly

Both Skillicorn's and Dasgupta's schemes describe an architecture by a set of func-

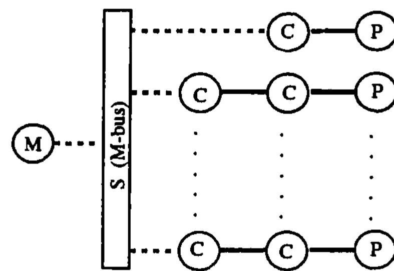


Figure 4–9: Physical organization of Firefly

tional units interconnected by switches. In our proposed scheme the switches (or connects) themselves are functional units. Thus our scheme describes an architecture as a set of interconnected functional units. This is important only from a modelling point of view, since it unifies model designs: the entire architecture is built using just functional units as building blocks.

4.3 Summary

To develop a generic approach that models parallel systems requires a well-formed and structured architectural representation scheme. This chapter reviewed some of the classification schemes for architectures and, based on them, proposed an architectural representation scheme.

A set of functional units, or atoms, forming the basic blocks of architectures is identified. The atoms are broadly divided into a set of connect atoms and a set of non-connect atoms. The set of non-connect atoms comprises processors, memories and caches; and the set of connect atoms comprises dedicated and shared connects. To model the instruction and data oriented functions of the non-connect atoms, a secondary set of non-connect atoms is derived. These non-connect atoms have their instruction and data oriented functions separated.

The atoms serve as building blocks to build architectures of one's choosing. This is achieved by connecting together the appropriate atoms. Structural diagrams are

used in representing the architectures thus built. A structural diagram graphically displays the organization and interconnection of the atoms that form the architecture. Structural diagrams of some typical architectures under the proposed representation scheme are illustrated.

Genesis, a generic modelling environment for parallel systems, is based on the representation scheme developed in this chapter. Genesis takes an object-oriented view of the architecture and models each atom of the architecture as an object. The next chapter discusses the design and implementation aspects of Genesis in detail.

Chapter 5

Genesis: A Generic Simulation Modelling Environment for Parallel Systems

This chapter discusses the design aspects of Genesis, a generic simulation modelling environment for parallel systems. The architectural representation scheme developed in the previous chapter becomes an integral part of the design of Genesis. Genesis takes an object-oriented view of the entire parallel system, viewing both the architecture and the software as sets of objects. Every single atom of the architecture is modelled by an object. Software entities – for example, tasks, task graphs and messages – are also modelled by objects. Software objects get mapped onto the hardware objects for a simulation of program execution. Mapping tasks onto non-connect atoms is the problem of assignment tackled in detail in chapters 2 and 3; mapping messages and requests (memory requests, input requests, etc.) onto connect atoms is the problem of routing. Genesis provides the means to specify both assignment and routing. A dynamic model of the entire parallel system is thus realized.

5.1 On the Design Choices

Constructing a dynamic model of a given system is called simulation modelling. The function of the model, called a simulator, is to mimic the behaviour of the system within the limitations of the system description.

A system consists of several physical entities, or components. At any given time, each of these entities has state information associated with it. For instance, a server might have two states: busy and idle. Ideally, the state of the simulator at a given simulation time should correspond to the state of the system at the corresponding real time. The change of state is called an event. An event triggers an activity – a unit of work – in the simulator. An activity will typically cause the creation of further events. A logically-related set of activities constitutes a process.

As the simulation proceeds, the simulation time advances in steps, depicting the changes in states and mimicking the corresponding activities. In a time-based or time-driven simulator, the time steps are regular, that is, the interval between any two successive time steps stays constant. If the time interval is too large, the simulator might miss some state changes. On the other hand, if the time interval is too small, the simulator would waste time advancing through time steps during which there are no state changes. Thus, in general, a time-based simulator lacks either accuracy or efficiency, or both.

Event-based simulators [Mac87] get around this problem by advancing the simulation time only to those points where there are state changes. Consequently, the time steps here are irregular. These simulators maintain an event list that is a diary of all unprocessed events. The simulation proceeds by removing from the list the event with the earliest time and modelling the corresponding activities.

In an event-based simulator, the system is modelled as a collection of events. Cod-

ing an event-based simulator is tedious and it is hard to get the code correct. Maintaining and updating the simulator is also tedious and time consuming [BLUL85].

An easier and more natural approach to model a system is to describe the behaviour of its components and the way they interact. Process-based simulators take this approach in which every active component of the system is modelled by a process, so that the actions and interactions of the processes correspond to those of the system's active components. A process could simply be a description of the system component's operation in the simulator's host language. Should the definition of a system component change, the simulator is updated by modifying the corresponding process that models the component. Process-based simulators are modular and thus make the construction and maintenance of large-scale models easy. Genesis is process-based; thus, for every system component, there exists a process in Genesis.

In modelling the system components, it is necessary to specify their static and dynamic structures. The static structure of a system component specifies its physical framework. The dynamic structure, on the other hand, specifies the way the component accomplishes its work. It is the dynamic structure that contributes towards the active nature of a component; thus, components that have no dynamic structure are said to be passive. In general, a system has both active and passive components.

Genesis takes an object-oriented approach to represent the system components and to describe their static and dynamic structures. The hardware components of the system have both static and dynamic structures associated with them; whereas software components have just static structures.

5.1.1 Object Orientation: Objects, Classes and Hierarchies

An object represents an entity and its associated behaviour. Related objects are grouped into classes. Related classes, in turn, can be grouped into further classes, thus resulting in a class hierarchy. An object belonging to a class is said to be an instance of the class. The classes forming the top-levels of the hierarchy are, in general, abstract. No object can be instanced from an abstract class. An abstract class can parent several child classes. A child class inherits most of its parent's properties. In addition, a child can have its own properties. That is, in general, a child is more 'knowledgeable' than its parent. An abstract class expresses a general concept, and a child specializes the concept. Classes in the bottom-levels of the class hierarchy are, in general, concrete classes. Concrete classes express the concepts specifically and completely, and thus they can instance objects.

Object-oriented programming [Str88] is a paradigm that approaches its solution to a programming problem by considering it as a set of objects, their actions and interactions. The object-oriented programming paradigm treats objects as first-class entities. Compare this, for example, with the functional programming paradigm where functions are the first-class entities.

The inheritance and abstraction mechanisms provided by object-oriented programming help in dealing with the development complexities involved in large software systems by enhancing software reusability, extensibility and maintainability. Class inheritance makes the software reusable and extensible. Data abstraction eases software maintenance.

5.1.2 The Modelling Approach

Modelling imposes two requirements: representing the system, and representing the system's work. The system and its work can be viewed in two different ways.

In the first case, the system components are treated as resources that the work can reserve and release. Here work is the active object, and the system stays passive. In other words, the work operates upon the system. For example, a task can reserve a processor for a certain amount of time for its execution. Similarly, a message that needs to be transmitted along a link can reserve the link during the transmission.

The second case treats the system components as the active objects. Work is considered to be passive and is operated upon by the system. For example, a processor can take up a task, execute it and send message packets to the links wired to the processor. Similarly, a link can take an incoming message from its input port and pass it to the output port after a suitable delay. This point of view is more realistic than the first.

Genesis takes the second point of view and treats the system components to be active and the work they do to be passive. Genesis has two main subsystems modelling the hardware and software components of the parallel computer. The hardware components are the active objects; and the software components stay passive.

The software and hardware subsystems are represented by suitably chosen class hierarchies. A software item could be an executable entity such as an instruction or a task; or it could be a non-executable entity such as a datum or a message. Both executable and non-executable entities are named items. Hardware entities, such as processors, memories and connects, are named atoms. An atom functions or operates upon items. The nature of an item – whether or not it is executable – is revealed only by its usage. This is similar to real systems where both instructions and data are represented by strings of bits and differentiated by usage.

5.2 Software Representation: The Software Hierarchy

An Item is the abstract base class that provides the framework for the software hierarchy. It can be either Executable or NonExecutable.

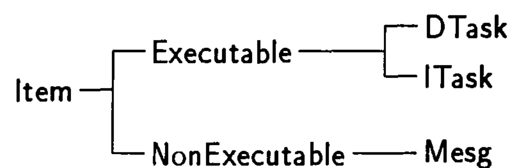


Figure 5–1: The software hierarchy

The class Executable encapsulates the executable objects that are processed by processor atoms. These executable objects are to be extracted from programs.

As has been noted in chapter 2, programs are modelled as task graphs where the vertices represent the tasks, or computation activities, and the edges represent interactions or dependencies between tasks. Depending on what the edges represent, task graphs are broadly classified into dependency graphs and interaction graphs. The execution model of a task under these two graphs differ. A task belonging to a dependency graph can start its execution when all its inputs are ready and finishes only when it has produced all its outputs. On the other hand, a task belonging to an interaction graph iterates infinitely through a sequence of *compute* and *communicate* steps. A task, an Executable object, thus takes two forms: a DTask object realizing a vertex of a dependency graph and an ITask object realizing a vertex of an interaction graph.

The task graph itself is viewed as an object; task graph objects are encapsulated in a class TaskGraph. A TaskGraph object is a collection of Executable objects and

represents an entire program. A dependency graph is a collection of **DTask** objects; and an interaction graph is a collection of **ITask** objects.

The class **Mesg** encapsulates the message objects that are passed from one atom to another. Message objects are constructed and destructed during the course of a simulation run.

5.3 Hardware Representation: The Hardware Hierarchy

An **Atom** is the abstract base class that provides the framework for the hardware hierarchy. An atom is considered to be a black-box that is free to function independently from other atoms. Atoms communicate with each other by passing items. Note that, unlike items, atoms are active entities. Thus, in a process-based simulation, every atom is represented by an independent process.

An atom may be busy simulating real work or simulating a delay. Both simulating work and simulating delays suspend the process that represents the atom and advances the process's simulation clock. An atom may also be busy communicating with other atoms. Communication involves the reception and sending of items. When an atom has no work or delay to simulate, it awaits the reception of items on its input ports. The items it may receive will trigger some work or delay, and may engage the atom in some communication, routing the items to other atoms.

An atom operates on items that represent both executable and non-executable objects. This enables an atom to migrate executable objects with as much ease as migrating a non-executable object.(This becomes useful when analysing the effect of task migration on performance.)

The class **Atom** owns properties essential to specify its operational behaviour. These properties include:

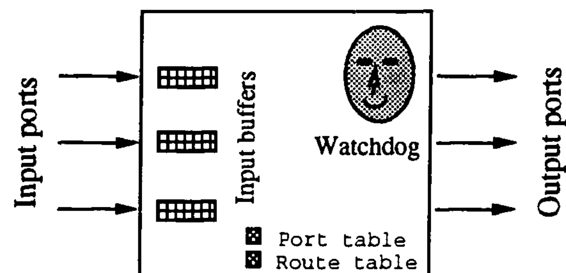


Figure 5–2: Atom: the base class of hardware hierarchy

- Ports

  An atom has several input and output ports that are used for communication with other atoms. An input port of an atom can be wired to an output port of another atom. This sets up the path necessary for communication. Associated with every atom is a port table that maintains details of the atoms directly wired to it.

- Buffers

  Atoms communicate with each other by passing items. Every input port of an atom has a buffer that queues incoming items.

- Communication

  An atom can send an item to a given output port. The item is placed in the buffer of the input port to which the given output port is wired. If the buffer is full, the send operation blocks the execution of the atom.

  Similarly, an atom can receive an item from a given input port. The item is simply removed from the buffer associated with the input port. If the buffer is empty, then the receive operation blocks the execution of the atom.

  An input port is said to be ready when the buffer associated with it is non-empty. A receive operation on this port will not block. An output port is said to be ready if the input port it is connected to has its buffer non-full.

A send operation on this port will not block. Non-blocking send and receive can be realized by testing for the readiness of the respective output and input ports.

- Routing

  An atom maintains a route table that can indicate to which output port an item must be sent in order to reach a given atom.

- Busy-waiting

  Every atom owns a watchdog (Appendix A) that can alert the atom when one or more of its input ports gets an item. The watchdog simulates busy-waiting. If an atom has nothing to do, it can go to sleep after starting up its watchdog.

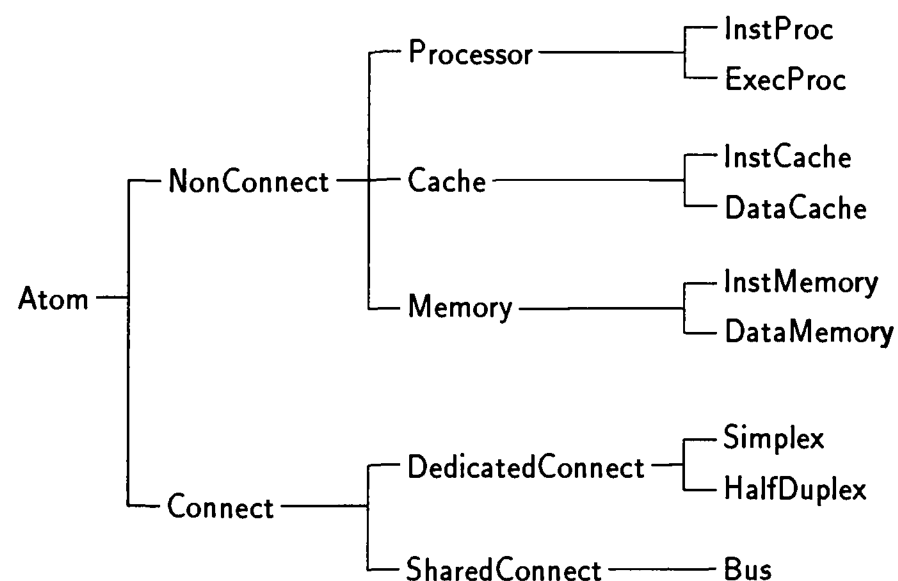


Figure 5–3: The hardware hierarchy

The complete hardware hierarchy is shown in figure 5–3. Note that there is no separate class representing a full-duplex link. A full-duplex link may be constructed by having two simplex links connected in parallel but in opposite directions.

The abstract classes Connect and NonConnect do not possess any extra properties that the class Atom does not. Yet, having them facilitates a better understanding of the structural design.

More classes can be derived from the basic classes as required. As an example, consider the class Bus that has the following static properties: a number of ports and an inter-port communication delay. No dynamic behaviour is associated with Bus since it is an abstract class. Concrete classes IDBus, an interrupt-driven bus, and TDBus, a time-driven bus, can be derived from Bus with suitable definition of dynamic behaviours. For instance, a high-level specification of the dynamic behaviour of a time-driven bus is "wait for something to arrive at one of the ports, send that *something* to the appropriate port after a delay, and go waiting again".

Instruction prefetch and execution can occur concurrently in most of the processors. Such concurrency is modelled elegantly by having instruction and execution processors as separate atoms so that they can execute independently and concurrently. This concurrency in instruction prefetch and execution needs to be modelled for instruction level simulations.

5.3.1 Building a Hardware Model

To build a hardware model requires the following three steps.

1. Deriving new atoms

   The active components of the system to be modelled are identified and grouped into classes according to their dynamic behaviours. These classes are to be derived from an appropriate class of the hardware hierarchy (figure 5–3) and their dynamic behaviours must be defined. However, if the default dynamic behaviour of a given atom is satisfactory, there will not be any need to redefine it. For instance, the default dynamic behaviour of a half-duplex link is sufficiently adequate that it need not be redefined.

2. Instancing new objects and wiring them

   A required number of objects is instanced from the appropriate classes. These objects are then wired together to resemble the physical framework of the system to be modelled.

3. Setting up route tables

   Route tables of most of the objects in the model must be set up. Some classes have a fixed routing scheme. Consider, for instance, a simplex link that has a single input port and a single output port: whatever appears on its input port gets to its output port after a delay. Objects of such classes do not need route tables to be set up.

As an example, assume that we wish to model the network of four P2 processors and two P4 processors shown in figure 5–4. A P2 processor has two ports and a P4 processor has four. Assume that their dynamic behaviours are different.

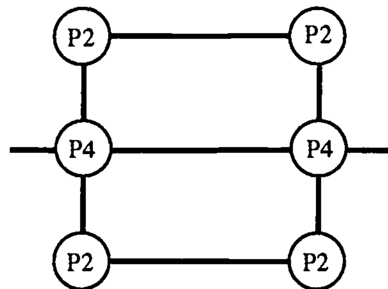


Figure 5–4: An example architectural model

P2 and P4 come under different classes. Thus we derive two classes P2 and P4 from the class Processor and specify their dynamic behaviours. The derivation also will specify the number of ports P2 and P4 have.

Four objects of the class P2 and two objects of the class P4 are instanced. In addition we need some connect objects, half-duplex links, for instance, to connect them. These objects are then wired together to form the network of figure 5–4.

Route tables must then be set up for all the processors. There is no need to set up route tables for the half-duplex links.

5.4 Specifying Assignment and Routing Schemes

Task assignment in Genesis is done during a simulation run. In modelling shared-memory systems, Executable objects are placed in a global task pool accessible by all the processors. In distributed-memory systems they are placed in private task pools belonging to each processor. Assignment schemes decide where each Executable object needs to be placed. In static schemes such decisions are taken before commencing the simulation run; dynamic schemes make decisions during the course of the simulation.

The execution model of an Executable object is determined by its type: DTask or ITask. The dynamic behaviour of the processor needs to specify the execution model. For instance, in a distributed-memory, message-passing system, the dynamic behaviour of a processor that executes DTask objects will typically be the repeated execution of the following steps.

1. If there exists a ready task in the local task pool, remove it from the pool and execute it (by increasing the local simulation clock to the task's finish time and suspending the process corresponding to the processor atom).

2. If there is a task that has just finished execution, send its results to all its successors (by forming message packets and passing it along the appropriate output ports).

3. If there is a message awaiting reception at any of the input ports, receive it. If it is destined to a task that belongs to the processor's local task pool, then

digest it (by updating the task's input record); otherwise route the message along the appropriate output port (since the message is destined to a task residing elsewhere).

It is only required to express these steps using the appropriate methods provided by the class Processor. Note that most of the methods the class Processor provides are properties inherited from the class Atom. Appendix B gives, in terms of these methods, a detailed description of the dynamic behaviour of a processor executing DTask objects.

Task preemption is modelled by suitably defining the dynamic behaviour of the processor. Preemption is particularly needed for the execution of ITask objects which infinitely iterate through a sequence of *compute* and *communicate* steps; fairness, in this case, can be guaranteed through preemptions.

Routing strategies are realized by setting up the route tables. For instance, consider again the distributed-memory, message-passing system. Every processor atom maintains a route table that tells to which output port a given message has to be routed in order to reach a given processor atom. The entries in the route table are thus determined by the routing strategy employed. Whenever there is a message that needs to be sent away to another processor, the route table is consulted to determine the appropriate output port the message has to be sent to. The message can either be one generated locally at the processor (i.e. message generated by a task executing locally) or one that is received at an input port for routing.

Dynamic routing strategies can be realized by updating the route tables at run-time.

5.5 Implementation Notes

Genesis is implemented in C++ [Lip91], an objected-oriented evolution of the C language. C++ extends C with several features including:

- A class construct

  The class construct supports data abstraction and encapsulation. A class is an aggregate of variables of any type and a set of member functions designed to manipulate those variables. The variables can be declared *private* so that they cannot be referenced other than by the member functions of the class. The member functions are often referred to as methods.

  Genesis defines a C++ class corresponding to every entity of the software and hardware hierarchies (figures 5–1 and 5–3). Each class abstracts the static and dynamic structures of the corresponding entity.

- Class derivation

  New classes may be derived from other class definitions, yielding a hierarchy of classes. The old class is the base class, and the new ones are derived classes. The *public* members of the base class become the public members of the derived classes; and the *protected* members of the base class become the private members of the derived classes. A derived class cannot access the private members of the base class.

  Genesis organizes the classes that represent various hardware and software entities in two powerful hierarchies illustrated in figures 5–1 and 5–3. Class derivation permits code sharing – code that is common to the children are owned by the parent so that the children can inherit them without duplicating.

- Virtual functions

  These let a derived class provide definitions of functions named in the base class. This feature allows multiple classes, all derived from the same base class, to provide type-specific implementations of semantically common functions. Yet, a derived class that does not need a special implementation of the virtual function need not provide one. Instead, the function of the base class is used. C++ guarantees that the most specific function is invoked at run-time.

  Most of the methods in the classes of Genesis come with a default implementation. Yet, being virtual, they let the derived classes provide sophisticated implementations, perhaps by improving upon the default.

Genesis uses the simulation engine Awesime [Gru91] that provides the building blocks for constructing process-based discrete event simulations. The Awesime class Thread implements a process. Several other globally known classes manage the set of threads within the simulation. The programmer needs to derive all the active entities in the system from the class Thread. Thread provides a virtual method called `main` that needs to be customized by the derived classes. Execution of a thread is simply the execution of its method `main`.

The class Atom, the base class of the hardware hierarchy, is derived from Thread, permitting itself to be an active entity. Thus all the classes of the hardware hierarchy, being derived from the class Atom, are active entities. Recall that every active entity owns a process that simulates its dynamic behaviour. This process is defined in the method `main` of the class the entity belongs to. As has been noted earlier, execution of an entity is the execution of the corresponding method `main`. The active entities are placed in an internal scheduler, maintained by Awesime, to let them execute. Every active entity is executed in turn, occasionally synchronizing with other active entities using semaphores. When waiting on a semaphore or when simulating a delay, an active entity permits itself to be descheduled by

the internal scheduler, paving way for other active entities to execute. When the semaphore on which the entity is waiting is acquired, or when the delay is simulated (i.e. when the global simulation clock advances enough), the scheduler will reschedule the entity[1].

5.5.1 Definition of the Class Atom

Implementations of most of the methods required by the hardware entities are provided by the class Atom. Thus this section gives a brief definition of Atom and some of its important methods. Figure 5–5 provides a partial definition of Atom.

The method `wait` lets the atom wait for one of its input ports to get ready. It suspends the process corresponding to the atom until an item is received at some input port. The method `inReady` examines if a given input port has an item. The method `outReady` examines if a given output port is ready to accept an item to send it away. It checks if the buffer of the input port it is wired to has room to receive an item.

The method `send` sends an item to a given output port. It blocks if the buffer of the input port the given output port is wired to is full. The method `recv` receives an item from a given input port. If the buffer of the input port has no item to offer, then `recv` blocks.

Methods `add2rtable` and `getRtable` are used in constructing and consulting the route table. The method `wire` wires a given output port to a given atom's input port. The virtual method `main` implements the dynamic behaviour of the atom.

[1]Such descheduling and rescheduling involve saving and retrieving the context. This may introduce a significant overhead to the simulation time if such context switches are many. Thus a process-based simulator may not be suitable for those systems that change state at almost every time step. For such systems, a time-based simulator may be more suitable.

```
class Atom : public Thread {
protected:
    void wait();
    short inReady(const int port_no);
    short outReady(const int port_no);
    int send(const int port_no, Item *const something);
    Item *recv(const int port_no);
    short *getRtable(const int aid) { return rtab.get(aid);
}
public:
    short *add2rtable(const int aid);
    int wire(const int outport, Atom *a, const int port_a);
    virtual void main() = 0;
};
```

Figure 5–5: A partial definition of the class Atom

Since the class Atom is abstract, it does not provide any implementation. Concrete classes, for instance the class HalfDuplex, implement `main`.

5.5.2 An Example: Building a Processor Grid

This section presents an example of constructing a simulation model.

Assume that it is required to build a grid of $a \times b$ processors with distributed memory. The processors are identical, have four ports each and are connected via half-duplex links to their nearest neighbours; no wrap-around connections are assumed at the grid edges.

The processors communicate with one another by message-passing. A simple message routing mechanism is employed. A message is first routed along the row (in which it originated) until it reaches the correct column. Then it is routed along the column until it reaches the right row.

Deriving a processor class. First a class Proc4 encapsulating the processor objects of the grid is derived from the class Processor. Proc4 is defined in the derivation to have four ports. See figure 5–6.

```
class Proc4 : public Processor {
private:
    int row_id, col_id;
protected:
    int getPort(const int i, const int j);
public:
    Proc4() : (4) { } /* Proc4 is a Processor with 4 ports */
    void main();
    /* Dynamic behaviour of Proc4 gets described in main */
};

int
Proc4::getPort(const int i, const int j)
{
    if ( j > col_id ) return 2; /* Go right */
    else if ( j < col_id ) return 0; /* Go left */
    else { /* j == col_id. Move along the column */
        if ( i > row_id ) return 1; /* Go up */
        if ( i < row_id ) return 3; /* Go down */
        else return -1; /* Stay here */
    }
}
```

Figure 5–6: Creating a processor with four ports

In general, it is required to set up route tables at every processor which indicate to which output port a given message must be sent in order to reach a given processor. Since the message routing mechanism in this example is simple, there is no need to maintain route tables. Whenever there is a message requiring routing, the identity of the output port is *computed* using the method `getPort`. Thus a table space of $\Theta(a^2b^2)$ is saved.

The dynamic behaviour of the class is assumed to be defined in the method `main` according to the execution model the class chooses to describe. For instance, if the execution model is to be that of a DTask, then the dynamic behaviour will be similar to the one described in section 5.4; a complete description of dynamic behaviour for this execution model appears in appendix B. Description in `main` will make use of `getPort` to find the appropriate output port to which a given message has to be sent.

Creating the grid of processors. The processors of class Proc4 are used in building the required grid. The function `makeGrid` (see figure 5–7) creates a rectangular grid of Proc4 processors. The processors are connected by identical half-duplex links of a given capacity.

```
Proc4 *
makeGrid(const int a, const int b, const int capacity)
{
    Proc4 *pptr = new Proc4[a*b];
        /* create a × b processors of type Proc4 */
    Proc4 *processor[a][b]; /* processor indices */
    int i, j; /* loop indices */

    /* index the processors according to their position in the grid */
    for ( i = 0; i < a; ++i )
        for ( j = 0; j < b; ++j )
            processor[i][j] = pptr + a*i + j;

    /* wire them across - do not wrap around the edges */
    for ( i = 0; i < a; ++i )
        for ( j = 0; j < b; ++j ) {
            if ( j > 0 )
                wireH(processor[i][j], 0, processor[i][j-1], 2, capacity);
            if ( i > 0 )
                wireH(processor[i][j], 1, processor[i-1][j], 3, capacity);
        }

    return pptr;
}
```

Figure 5–7: Creating a processor grid

The function `wireH` wires the specified ports of two non-connect atoms with a half-duplex link of a given capacity. It is a function that instances a new HalfDuplex

atom and connects the two given non-connect atoms with this new atom. It uses the method `wire` implemented in the class Atom (see figure 5–5).

Firing up the simulation. Figure 5–8 shows a typical code that simulates the execution of a task graph on the processor grid created. The task graph is read and mapped onto the processor grid. A call to the global function `go_simulate` starts the simulation.

```
int main(int argc, char *argv[])
{
    const int a = 64; const int b = 128; const int capacity = 1;

    /* Create an a × b grid of processors */
    Proc4 *grid = makeGrid(a, b, capacity);

    /* Read in the task graph specified in the command line */
    TaskGraph taskGraph(argv[1]);

    /* Map the task graph on the grid */
    map(taskGraph, grid, a, b, capacity);

    /* Start the simulation */
    go_simulate();

    return 0;
}
```

Figure 5–8: Firing up the simulation

The function `go_simulate` passes all the active entities (in this case, the processors and the half-duplex links) on to Awesime's internal scheduler and the scheduler then executes the entities' `main` methods appropriately.

5.6 Comparison with Related Works

Comparable related works on modelling parallel architectures include PARET [NE88], ASIM [Jum90] and OASIS [UBP81]. These are all object-oriented modelling systems – PARET and ASIM are based on C++ and OASIS is based on Simula.

PARET (Parallel Architecture Research and Evaluation Tool) is targeted for non-shared memory, MIMD architectures. A specific computer model comprises three subsystems – the user program, the interconnections and the system functions that each processor must execute. These subsystems share the same model – a directed flow graph where the nodes represent units of action and the arcs represent both data and control flow. PARET provides a graphical user interface that lets the user draw and edit the graphs representing the algorithm and the architecture. Run-time statistics are displayed by user-selected meters.

ASIM is a parallel computer simulator that belongs to a family of discrete event simulators called xSIM. Except a class Processor, all the simulation classes represent software entities. Processors model the sequential processors in a multiprocessor computing system. A process is attached to a processor in order to model the effect of the process executing on that processor. This attachment of processes to processors is under user control and can be changed during simulation. This enables simulations to model process migration. A process can be an actual program written in C. Process synchronization primitives are provided by xSIM.

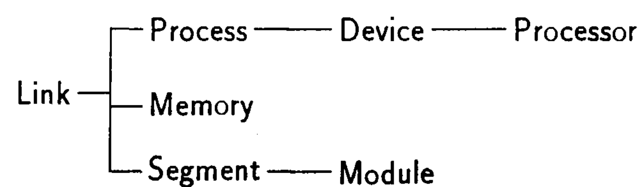


Figure 5–9: OASIS class hierarchy

OASIS is a library of Simula classes. See figure 5–9. The classes Link and Process are provided by Simula. The OASIS classes Device, Segment, Memory, Processor and Module represent an abstract 'computer system device', 'unit of information', 'information storage device', 'information processing element' and an 'executable program' respectively. Each class allows facilities fundamental to the class. For example, class Processor has a facility for simulating the execution of programs, or objects belonging to the class Module. Using the appropriate classes and the facilities they provide, the user creates the simulation model.

In comparison to these works, Genesis is not restricted to any particular category of computers (cf. PARET, ASIM). It can model anything from a simple bus architecture to novel architectures like a dataflow architecture. Classes essential to construct most of the architectures are provided by Genesis. The choice of classes is based on the architectural representation scheme underlying Genesis. Thus the classes are realistic (cf. OASIS).

Hardware components are active objects in Genesis. Every hardware component is modelled by a black-box that is free to function independently. The user has control over the way these black-boxes are connected and the way they function. Software components stay passive. Thus Genesis cannot run a task written in a high-level language. A high-level program must be translated into an intermediate form. This is a drawback of Genesis. Simulation systems such as ASIM support tasks written in high-level languages. Yet, these systems do not have the flexibility that Genesis has in modelling the hardware.

Genesis provides the means to specify and alter routing and assignment schemes. Both static and dynamic schemes can be supported. Since Genesis describes both executable and non-executable objects in a unified framework, it permits migration of executable objects with as much ease as migrating a non-executable object.

Currently there is no graphical user interface to Genesis. Both the hardware and the software must be expressed by some code. A graphical interface similar to that of PARET would be a useful addition to Genesis.

5.7 Summary

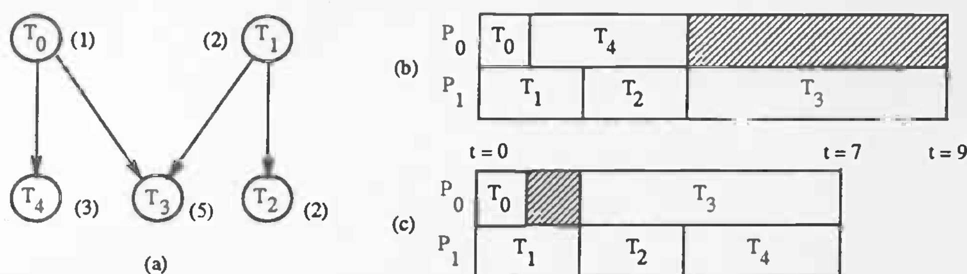
The design of Genesis is described. Genesis presents a unified, object-oriented approach to model parallel systems. The approach is based on the architectural representation scheme developed in chapter 4. Genesis provides the means of describing and modelling the key parameters determining the performance of a parallel system: the architecture, program, assignment method and routing scheme. It is thus a good laboratory for carrying out experiments in performance analysis.

We now turn our attention back to the assignment problem and, using Genesis as a modelling platform, report some experiments on the performance of the assignment schemes described in chapter 3.

Chapter 6

Performance Assessment of Assignment Schemes

Even though work-greedy assignments give guaranteed solutions, they could be worse than a non-work-greedy assignment. The following example demonstrates this. Consider the assignment of the task graph of figure 6–1(a) onto a two processor system $\{P_0, P_1\}$ with zero interprocessor communication delay. Gantt charts of figures 6–1(b) and 6–1(c) show that a work-greedy assignment fairs poorly in comparison to a non-work-greedy one (which, in this case, is an optimal assignment).



(a) Task graph (b) A work-greedy assignment (c) A non-work-greedy assignment (Numerals in parenthesis denote task execution times.)

Figure 6–1: Work-greedy assignments are not always good

However, there is no analytical performance guarantee for a non-work-greedy as-

signment. Thus this chapter presents an experimental performance analysis of the non-work-greedy assignment scheme DFBN.

Comparison of assignment schemes that ignore communication costs have been reported in the literature [ACD74,SWP90]. However, to our knowledge, no comparison of the schemes that consider communication costs has yet been reported. Thus, this chapter reports an extensive set of results comparing these schemes. The comparison experiments use the assignment schemes, ETF, ERT, MH and DFBN. The scheme MCP is not considered here since it is very similar to RMH, the restricted version of MH.

The assignment schemes are tested with random task graphs as well as task graphs obtained from real program routines. For some small task graphs, the makespans of the assignments generated by the heuristic schemes are compared against the makespans of the optimal assignments.

Both DFBN and the work-greedy assignment schemes assume that the task graph parameters – task execution times, volumes of information transfer, etc. – are known *a priori*. However, in practice, it is hard to measure these parameters accurately. One would expect that such inaccuracies would lead to poor assignments. Thus, some experiments are conducted to investigate the impact of measurement inaccuracies on the makespan.

The Experimental Framework. Task dependency graphs are executed on processor graphs (topologies) using partitions dictated by the assignment schemes. Genesis is used for building these processor graphs and simulating the execution. The means of building processor graphs and simulating the execution is discussed in chapter 5. The simulation method uses a fixed shortest path routing for communication; network contention is taken into account by queueing messages. To transfer a message of size v through a link that has an information transfer rate c takes $\lceil v/c \rceil$ units of time. It is assumed that the processors can do only one activity at a time: either computation or communication.

The following points are to be noted.

- The scheme MH uses the number of hops between processors in determining communication costs. This is fine when the processors are connected by identical links, but not otherwise. Thus we use the inter-processor distance (as defined in chapter 3) in determining communication costs under MH.

- MH uses an adaptive shortest path routing scheme while finding the assignments. Even though we use this adaptive routing scheme while determining assignments under MH, we do not use the adaptive scheme when we find the makespans of the MH-generated assignments. We only use a simple shortest path routing scheme.

- DFBN uses a weighted sum of priorities to calculate the overall priority of a task. See chapter 3, equation 3.4.1. Since we do not know what choices of weights will *consistently* give good assignments, we arbitrarily set $w_0 = 10$, $w_i = 1 \ (i = 1 \ldots 5)$ and $\alpha = 1$. This choice gives more weight to the critical tasks than to the others.

6.1 Optimal Assignments

To see how close to optimal a given assignment is, the optimal makespan (i.e. the makespan of the optimal assignment) must be known. The best known method to find an optimal assignment (and the optimal makespan) is exhaustive search. An exhaustive search looks at every possible assignment and chooses the one with the minimum makespan.

If the tasks are independent of each other, then the number of possible assignments that should be looked at is m^n. Now if an arbitrary precedence relation is introduced among the tasks, then task ordering within the partitions of an assignment

must be taken into account in determining the number of possible assignments, N. Task ordering increases N, while precedence relation decreases it. The increase due to task ordering is reflected in the following lemma.

Lemma 6.1. The number of possible assignments a set of n tasks can have on an m processor system is

$$W(n,m) = \frac{(n+m-1)!}{(m-1)!}.$$

ignoring the effect of precedence relation among the tasks, but taking into account the task ordering.

Proof.

The proof is by induction.

Let the number of possible assignments be $W(n,m)$. Since a task can be assigned to any of the m processors, $W(1,m) = m$. Assume that

$$W(q,m) = \frac{(q+m-1)!}{(m-1)!}$$

Now increase the number of tasks from q to $q+1$. The new task can be added to a partition after any of the q tasks; it can also be placed at the head of any of the m partitions. Thus, there are $q+m$ ways of inserting the new task into an existing assignment. That is,

$$W(q+1,m) = (q+m)\ W(q,m) = \frac{(q+m)!}{(m-1)!}$$

Therefore, if the result holds for $n = q$, it will hold for $n = q+1$.

□

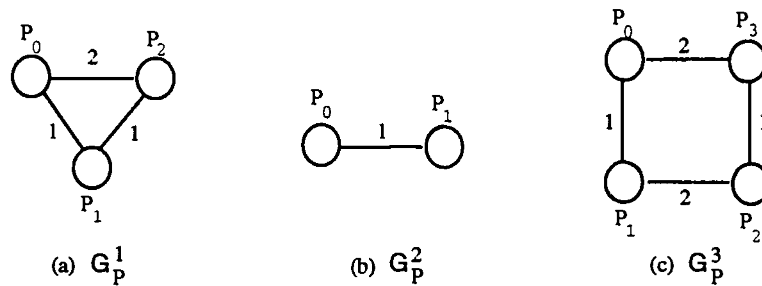
The decrease in N due to precedence relation is hard to determine. This problem is as hard as finding the number of linear extensions in a poset which has been proved

to be #P-complete[1] [BW91]. Given an arbitrary precedence relation among the tasks, we only have an upper bound on the number of possible assignments. That is,

$$N \leq \frac{(n+m-1)!}{(m-1)!}.$$

An exhaustive search through the possible assignments is thus enormously time consuming and is practical only for very small task graphs. Hence, three small-sized task graphs (see figure 6–3) are chosen and their best-case and worst-case makespans (corresponding to the best and worst assignments) on the processor graphs of figure 6–2 are found through an exhaustive search. Table 6–1 lists these best-case and worst-case makespans; the corresponding assignments are tabulated in table C–1 of appendix C.

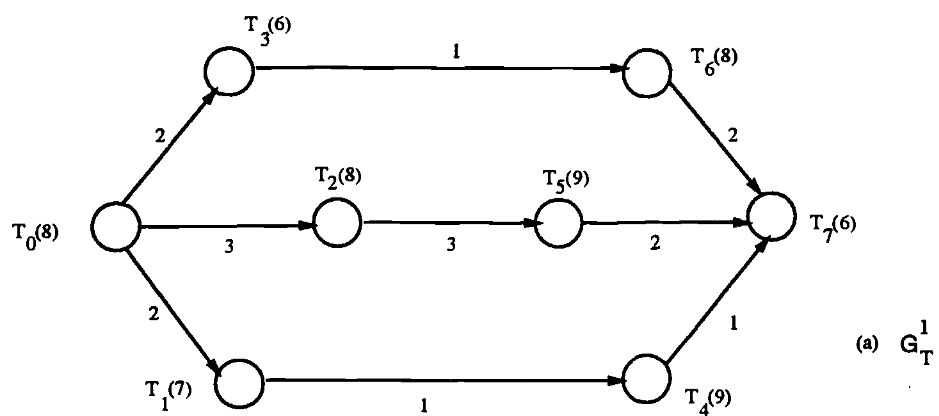
Out of the task graphs of figure 6–3, G_T^3 is taken from Hwang et al. [HCAL89]; G_T^1 and G_T^2 are chosen arbitrarily.

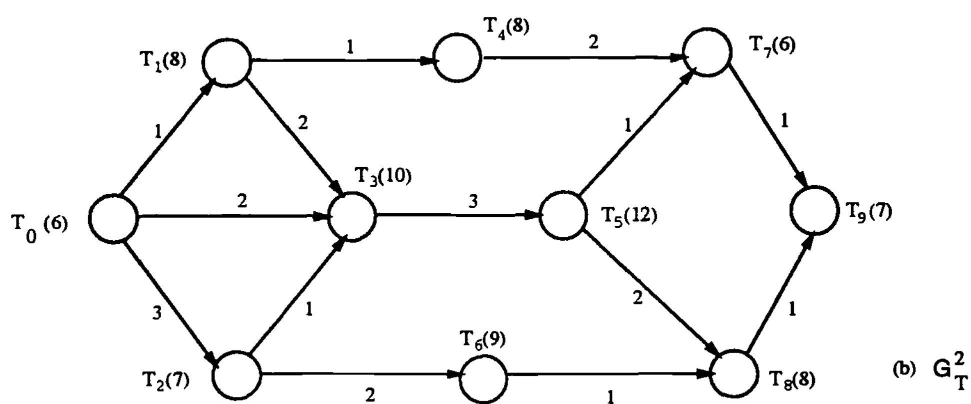


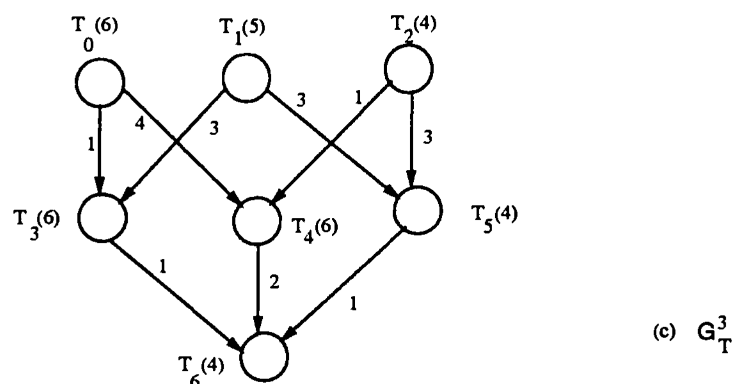
(Information transfer rates are shown beside the edges of the graphs.)

Figure 6–2: Example processor graphs

[1]Any #P-complete problem is as hard as counting the satisfying assignments of a boolean formula [Val79].







(Numerals in parentheses denote task execution times. The volume of information transfer is shown beside each edge.)

Figure 6–3: Example task graphs

G_T	G_P	The best-case makespan	The worst-case makespan
G_T^1	G_P^1	32	65
G_T^1	G_P^2	39	65
G_T^1	G_P^3	32	68
G_T^2	G_P^1	53	92
G_T^2	G_P^2	54	92
G_T^2	G_P^3	53	98
G_T^3	G_P^1	17	39
G_T^3	G_P^2	22	39
G_T^3	G_P^3	17	42

Table 6–1: Best-case and worst-case and makespans

The worst-case execution time is never less than the sequential execution time; in general, it may be greater because of delays due to message latencies.

For each task graph and processor graph combination (figures 6–3 and 6–2), assignments are generated using each of the assignment schemes (see table C–2 of appendix C). The execution of the task graph on the processor graph is then simulated, and the makespan of the task graph on the chosen processor graph is tabulated in table 6–2. As an example, figure 6–4 illustrates the execution of G_T^1 on G_P^1 with a best-case assignment. Note that DFBN generates a best-case (or optimal) assignment in this case.

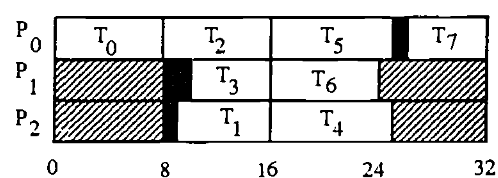


Figure 6–4: Gantt chart showing an execution of G_T^1 on G_P^1

Table 6–2 lists also the total execution times of task graphs under a random assignment. The time complexity of a random assignment is $\Theta(n)$.

G_T	G_P	ETF	ERT	MH/RMH	DFBN	Random
G_T^1	G_P^1	34	34	33	32	47
G_T^1	G_P^2	41	40	42	48	47
G_T^1	G_P^3	34	34	33	32	44
G_T^2	G_P^1	53	55	55	53	58
G_T^2	G_P^2	61	57	55	54	69
G_T^2	G_P^3	54	55	55	53	66
G_T^3	G_P^1	18	18	18	18	23
G_T^3	G_P^2	24	24	25	24	26
G_T^3	G_P^3	17	17	23	19	30

Table 6–2: Makespans of the task graphs

Examination of tables 6–1 and 6–2 leads us to the following observations:

- DFBN generates optimal assignments in five cases; ETF in two cases; and ERT in one case.

- The heuristic assignments, in general, perform better than a random assignment.

Effect of Routing on the Makespan. Consider the assignment of G_T^3 onto G_P^3 under MH. See table C–2 of appendix C for the partitions and figure 6–5 for a Gantt chart showing execution. T_0 and T_2 finish their execution at instances 6 and 4 respectively. To start T_3 on P_2 output message from T_0 on P_0 should reach P_2; and to start T_4 on P_0 output message from T_2 on P_2 should reach P_0. The non-adaptive shortest path routing scheme used in the simulation requires messages to and from P_0 and P_2 to be routed via P_1. Since P_1 is busy executing T_5, these messages do not get routed until T_5 is completed. Therefore, T_3 and T_4 have to start as late as instance 11. Had the messages been routed via P_3, the makespan would have been shorter. (Alternatively, if P_1 could be interrupted to handle message routing, then also the makespan could be shorter.)

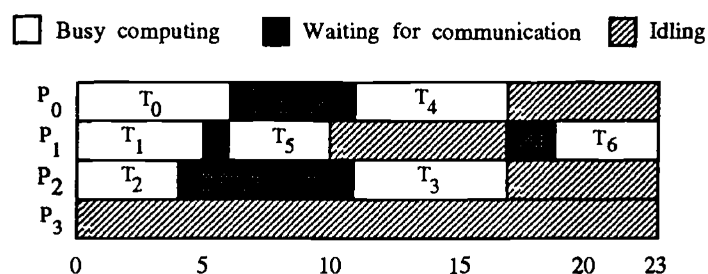


Figure 6–5: Effect of routing on the makespan

It is thus seen that smart routing schemes can be employed to shorten the makespans. However, since our goal is only to study the performance changes due to different assignment techniques, we do not need to use any smart routing scheme; we only have to fix a routing scheme and use it consistently.

6.2 Assigning Random Graphs

This experiment tests the assignment schemes using a number of random task dependency graphs. Random dependency graphs help to predict the performance of assignment schemes without any assumption about specific workloads. Processor graphs of figure 6–2 are used in executing these random task graphs. Random task dependency graphs are generated following the technique outlined in [ACD74]. 111 random graphs[2] with the following characteristics are generated for the test.

[2]The number 111 has no special significance. It is an arbitrary selection.

Range of number of vertices	8 – 35
Average number of vertices	21
Range of number of edges	15 – 44
Average number of edges	27
Range of mean execution time of a task	13 – 42
Average mean execution time of a task	28
Range of mean volume of information transfer	4 – 18
Average mean volume of information transfer	11

Task execution times and volumes of information transfer have negative exponential distributions with the mean values given above. Thus the execution times and volumes of information transfer tend to vary widely within a task graph.

G_P	ETF	ERT	MH	RMH	DFBN
G_P^1	278	274	274	274	265
G_P^2	362	352	361	361	346
G_P^3	269	264	275	275	255

Table 6–3: Average makespan of the random graphs

Table 6–3 shows the average makespan of the assignments generated by different assignment schemes. The average is taken over the 111 random graphs.

Some graphs may favour a particular heuristic. For example, consider the task graph of figure 2–3(a). The best-case work-greedy assignment of this graph is the one shown in the Gantt chart of figure 2–3(c) whereas the best-case non-work-greedy assignment is the one in figure 2–3(a). The best-case work-greedy assignment is worse than the best-case non-work-greedy assignment. Now consider the same task graph with all its precedence relations inverted. The best-case assignments under both work-greedy and non-work-greedy schemes will then be the assignment shown in figure 2–3(c) (but with the chart reversed). That is, by inverting the precedence relation of the task graph we see that both schemes fair equally well.

To investigate the possibility of the graph generation process favouring a particular heuristics, we repeat the experiment with the same random graphs but with all the precedence relationships inverted. Table 6–4 shows the average makespan of the precedence-inverted random graphs under different assignment schemes. Tables 6–3 and 6–4 show no evidence that the random graph generation is biased.

G_P	ETF	ERT	MH	RMH	DFBN
G_P^1	262	254	261	261	250
G_P^2	351	341	345	345	336
G_P^3	244	242	250	248	236

Table 6–4: Average makespan of the precedence-inverted random graphs

Examination of tables 6–3 and 6–4 reveals that

- DFBN does consistently better than the rest of the assignment schemes. Performance of DFBN is up to 8% better than the rest of the schemes considered here.
- MH does not exhibit any significant performance improvement over RMH. In fact, there are cases where RMH does better than MH. This may be due to the fact that the simulator that executes the task graphs uses a fixed shortest path routing scheme rather than the adaptive routing embedded in MH.

Note that the random graphs used in obtaining tables 6–3 and 6–4 are sparse. Sparse random graphs are used because they closely model real programs.

6.2.1 Assessing the Effect of Estimation Errors

Due to many non-deterministic factors affecting program execution, it is not always possible to determine the exact values of the task graph parameters. Compile time

analysis cannot predict the task execution times when there are run-time dependencies. Even in the case of instruction scheduling, where the compiler knows the instruction execution times, load and store instructions may take longer to execute than predicted, because of cache misses[3]. Run-time estimation of parameters may also be inaccurate. For instance, execution time of a task in a multiprogramming system may not be deterministic due to the interference by other tasks executing in the system.

This experiment examines the effect of estimation error on the makespans of random task graphs. The sparse random graphs of the previous experiment are re-used.

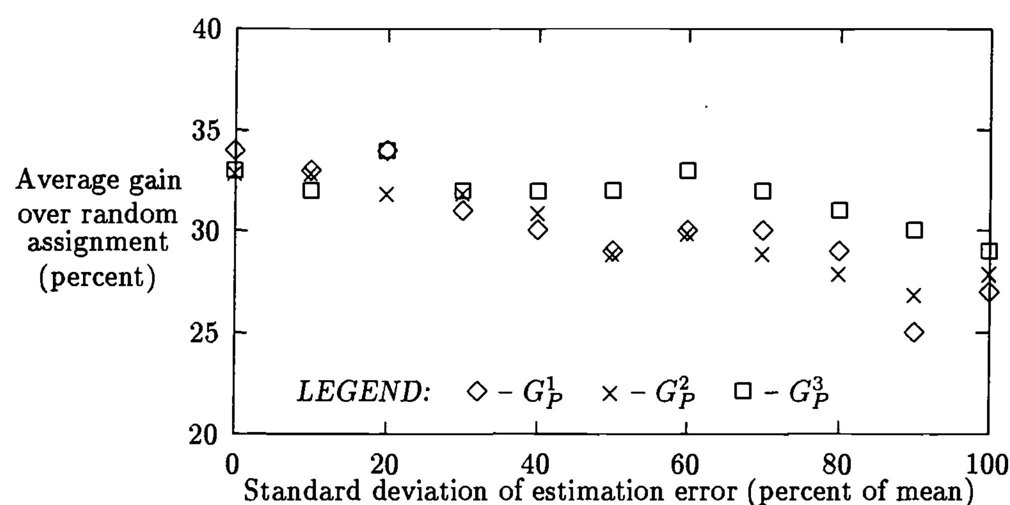


Figure 6–6: Effect of estimation error on the makespan under ETF

Assignments under the schemes ETF, ERT, RMH, MH and DFBN are determined with random errors added to both the task execution times and the volumes of information transfer. A random error is assumed to be normally-distributed. Makespans of the task graphs on the processor graphs of figure 6–2 under these

[3]Normally, in such cases, the entire instruction pipeline is frozen to maintain the state of the schedule.

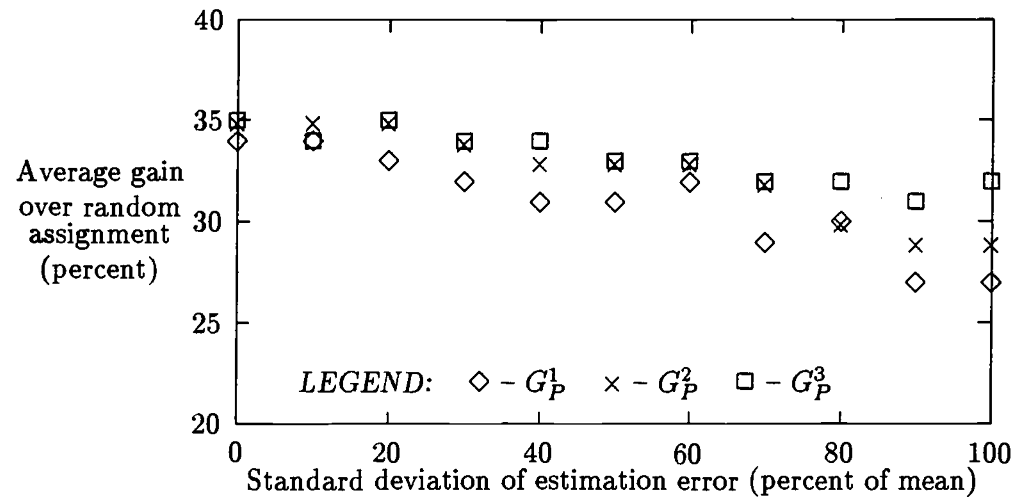


Figure 6–7: Effect of estimation error on the makespan under ERT

assignments are found through simulation. The simulation uses the correct task graph parameters in executing the task graphs (i.e. it does not add any random error to the task graph parameters).

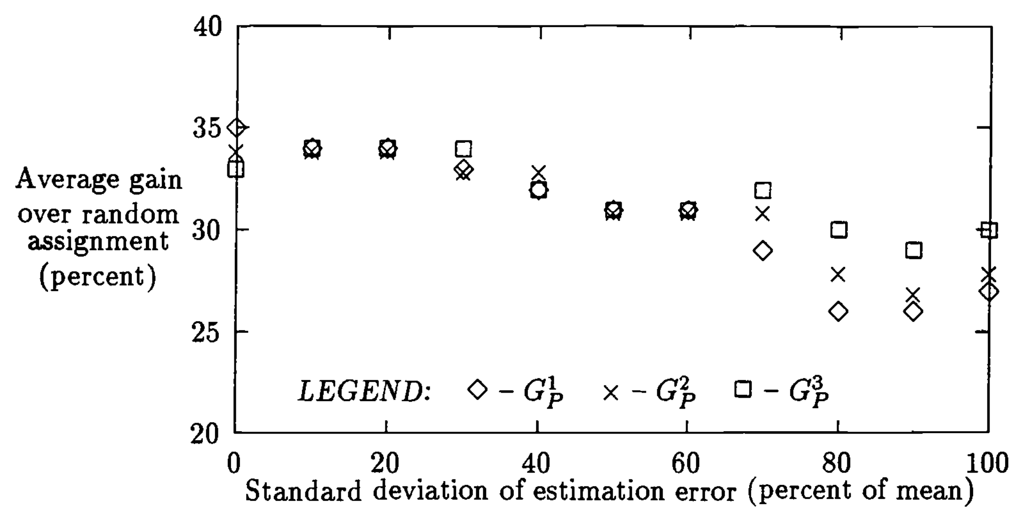


Figure 6–8: Effect of estimation error on the makespan under RMH

The graphs of figures 6–6 to 6–9 illustrate the effect of estimation errors on the makespan. The average gain in the makespan of a heuristic assignment is computed with respect to the makespan of a random assignment and is plotted against the standard deviation of estimation error. The gain in the makespan is defined

as

$$\frac{\omega_{ran} - \omega}{\omega_{ran}} \times 100$$

where ω_{ran} is the makespan of the random assignment and ω is the makespan of the heuristic assignment. The average is taken over the 111 random graphs used in the previous experiment.

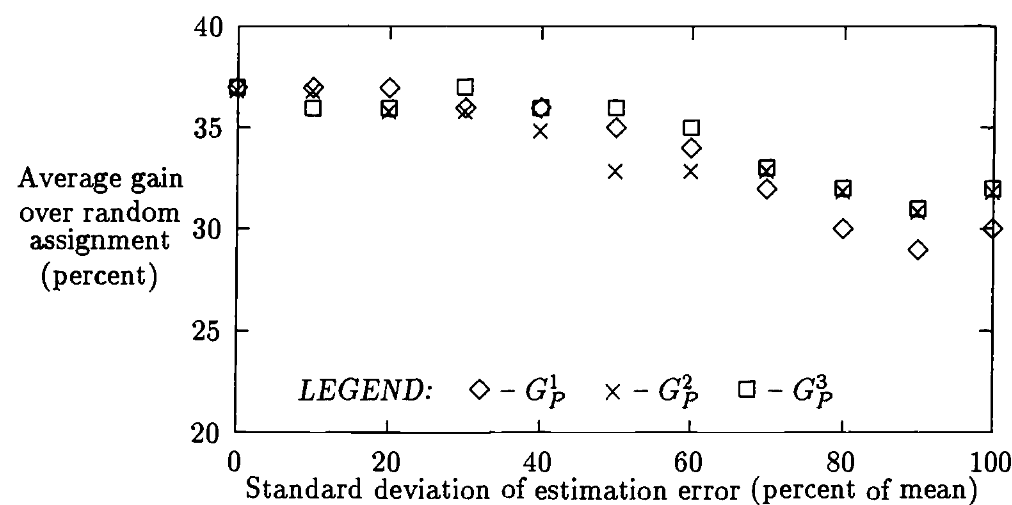


Figure 6–9: Effect of estimation error on the makespan under DFBN

Since the makespans produced by RMH and MH are very close to each other, only the results from RMH are reported here.

These graphs show that the estimation errors have very little impact on the makespan. A 70% standard deviation of estimation error results in no more than 6% variation in the average gain. The prime reason for this low sensitivity for estimation errors is that the assignment schemes depend more on the structure (or the layout) of the task graph than on the parameters of the graph.

This important result implies that accurate estimation of task graph parameters is not necessary to produce reasonably good assignments. As long as the structure of the task graph and *some* estimate of the task graph parameters are determined at compile time, the assignment schemes will generate good assignments. This

means that, if the task graph structure is known at compile time, one need not postpone the assignment until run time.

Observe also that DFBN has the maximum average gain over the random assignment.

6.3 Dependency Graphs from Programs

Synthetic random graphs often lack the complex embedded correlations that real task graphs contain. It is thus desirable to validate the results obtained using synthetic random graphs by experiments conducted on real task graphs. This section thus repeats the experiments carried out in the last section using dependency graphs extracted from real program routines.

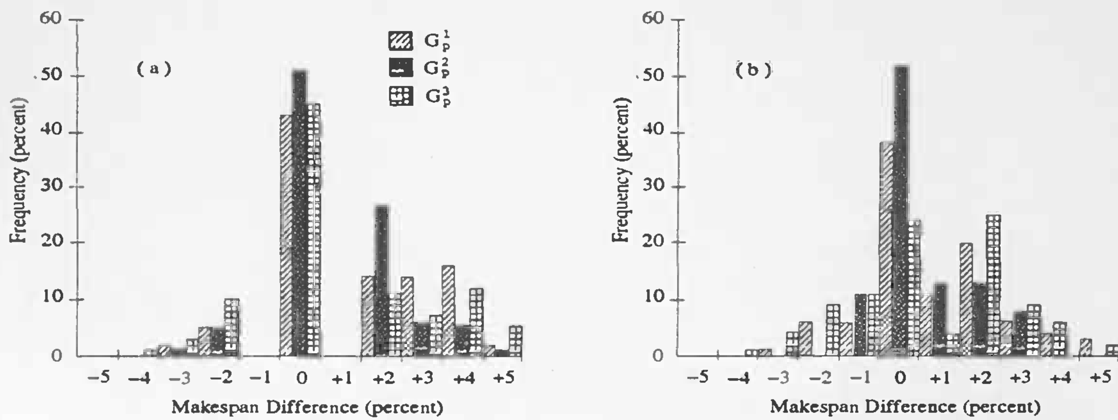
More than 500 dependency graphs obtained from an extensive dependency analysis of Fortran subroutines are used in this section to test the assignment schemes. The subroutines are part of one of the Perfect Club Benchmarks [B+89,CKPK90]. The benchmark implements a QCD (Quantum Chromodynamics) simulation using the Cabbibo-Marinari pseudo heat-bath algorithm [CM82]. The QCD simulation is required in high energy Physics for understanding subnuclear processes and is extremely computationally demanding.

The dependency graphs correspond to the basic blocks (i.e. the maximal blocks of instruction sequences containing no branches) of the source subroutines. Tasks are thus fine-grained, representing individual instructions. They have execution times of 1 – 8 clock cycles. Each dependency is taken to mean a unit information transfer between the dependent instructions. The biggest graphs have 63 instructions. No graph having less than 8 instructions is used. These graphs are grouped into two: a set of small graphs each with 8–15 instructions; and a set of large graphs each with 16–63 instructions.

The processor graphs of figure 6–2 are used in executing these dependency graphs under different assignment schemes. Makespans of the assignments under ETF, ERT, MH and RMH are compared to the makespan of the DFBN generated assignment. The performance gain of DFBN over the other assignment schemes is expressed by the makespan difference

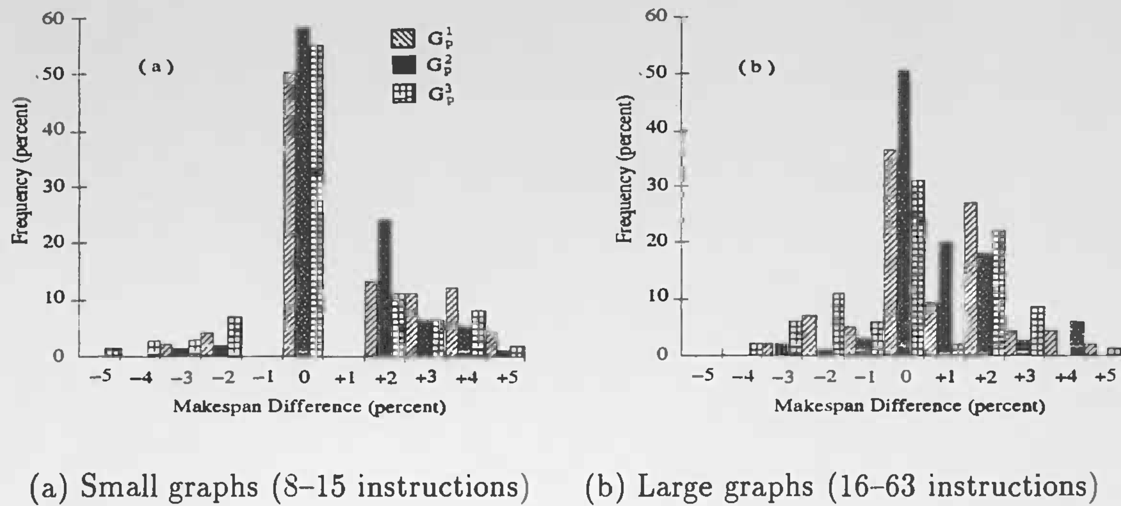
$$\Delta_X = \frac{\omega_X - \omega_{DFBN}}{\omega_{DFBN}} \times 100 \qquad \text{where } X \in \{\text{ETF, ERT, MH, RMH}\}.$$

Since MH produced makespans very close to those of RMH, results from MH are not reported here.



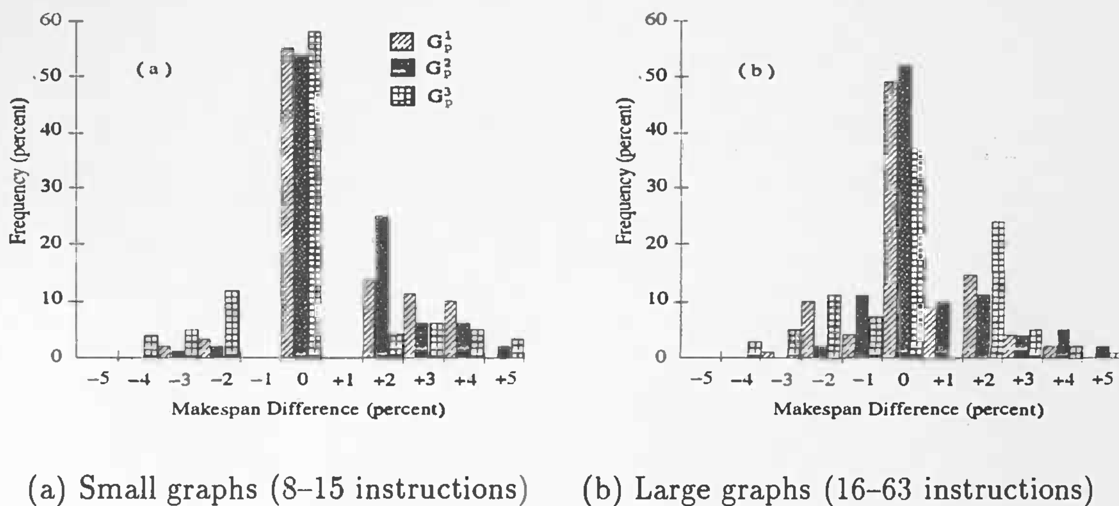
(a) Small graphs (8–15 instructions) (b) Large graphs (16–63 instructions)

Figure 6–10: Frequency of makespan difference between ETF and DFBN



(a) Small graphs (8–15 instructions) (b) Large graphs (16–63 instructions)

Figure 6–11: Frequency of makespan difference between ERT and DFBN



(a) Small graphs (8–15 instructions) (b) Large graphs (16–63 instructions)

Figure 6–12: Frequency of makespan difference between RMH and DFBN

The histograms of figures 6–10 to 6–12 show the frequency of Δ_X, where X is either ETF, ERT or RMH. The positive values of Δ_X imply an advantage for DFBN over the assignment scheme X. Figures 6–10 to 6–12 show a definite advantage of using DFBN over the other schemes. Table 6–5 summarizes the advantage of DFBN over these schemes. It shows the percentage of cases where DFBN performs better than or equal to the assignment scheme under comparison. The figures in the table are averages over the small and large instruction graphs and the three processor graphs.

DFBN advantage	
82%	over ETF
83%	over ERT
81%	over RMH

Table 6–5: The DFBN advantage

6.3.1 Effect of Estimation Errors

Section 6.2.1 reported experimental evidence suggesting that the assignment schemes considered here are fairly insensitive to estimation errors. The experiment used random graphs. Here we repeat those experiments with the instruction dependency graphs extracted from QCD subroutines.

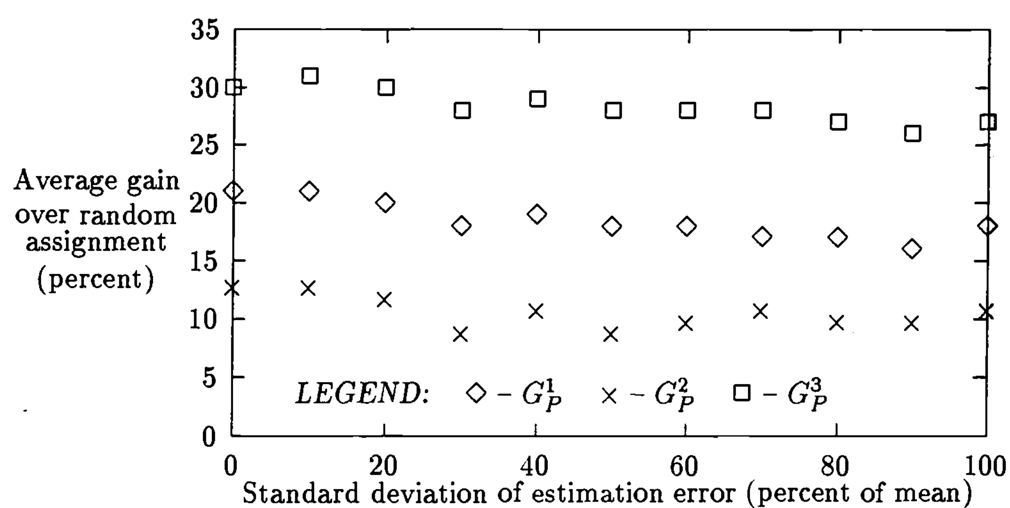


Figure 6–13: Effect of estimation error on the makespan under ETF

Assignments of the instruction graphs on the processor graphs of figure 6–2 are determined using the heuristic schemes ETF, ERT, RMH and DFBN. Normally-distributed random errors are added to the instruction execution times while determining the heuristic assignments. As before, the makespans of these assignments are found through simulations using the error-free instruction execution times. The average gain in the makespan of heuristic assignments is calculated with re-

spect to the makespan of a random assignment. See section 6.2.1 for a definition of makespan gain.

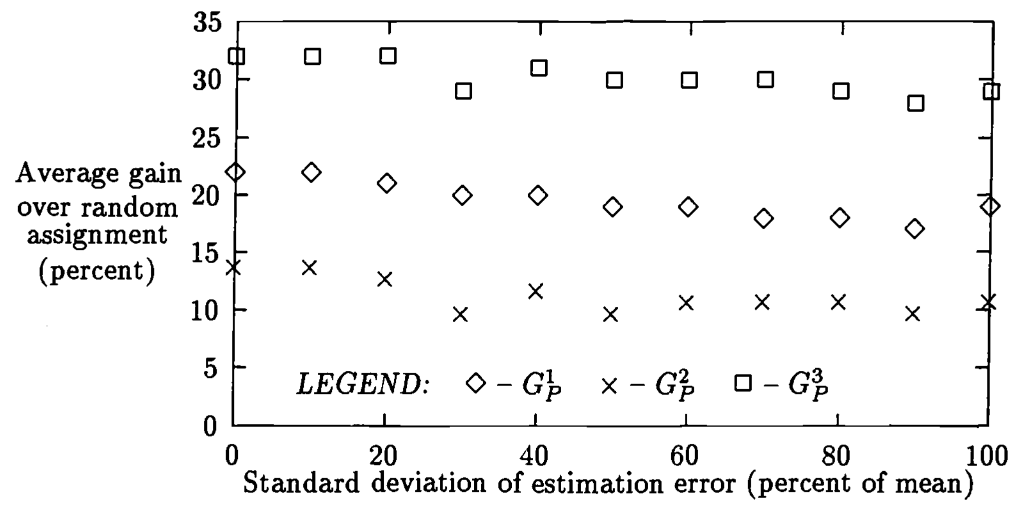


Figure 6–14: Effect of estimation error on the makespan under ERT

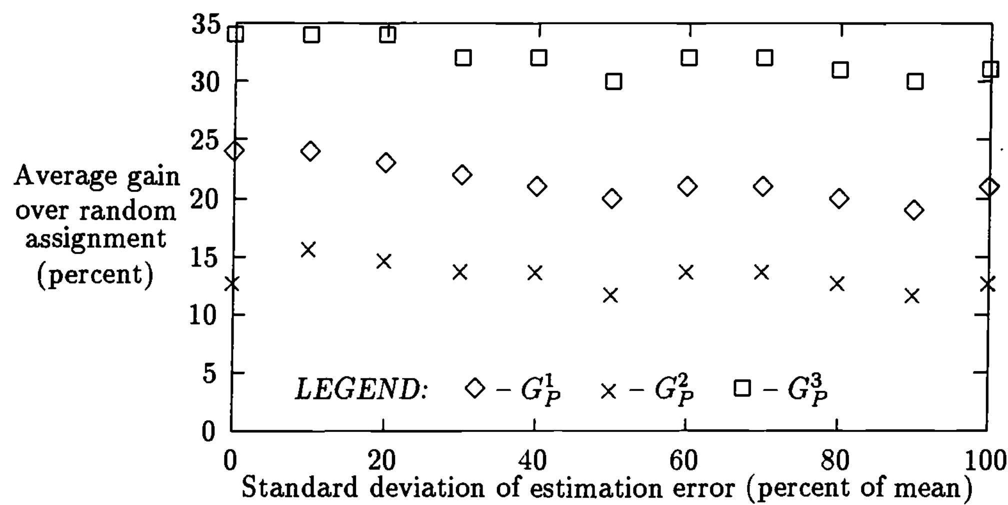


Figure 6–15: Effect of estimation error on the makespan under RMH

Figures 6–13 to 6–16 show the effect the maximum estimation error has on the average makespan. Maximum estimation errors up to 100% result in no more than 6% variation in the average makespan.

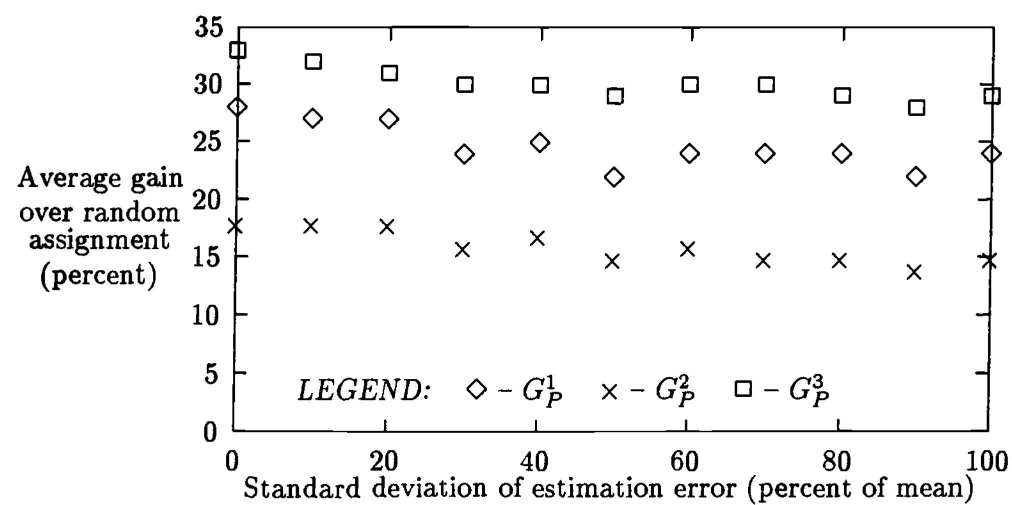


Figure 6–16: Effect of estimation error on the makespan under DFBN

Out of the assignment schemes tested, DFBN has the maximum average gain over the random assignment. One exception to this is the assignment onto G_P^3; RMH performs slightly better than DFBN in this case.

To summarize, all the assignment schemes exhibit a good deal of insensitivity to estimation errors. This has two important implications:

1. Inaccuracies in the estimation of task graph parameters can be mostly tolerated.

2. An instruction schedule generated for a particular architecture can be executed on a slightly different architecture, where some instructions have different execution times, without incurring a large penalty.

6.3.2 On the Assignment of Loops

The dependency analysis carried out to extract dependency graphs from the QCD subroutines does not utilize the parallelism that may be present across the loops – it just extracts the dependency graph corresponding to the loop body. However,

it is possible to extract more parallelism (than is visible in the body) from a loop by unrolling it several times.

Consider the loop of figure 6–17(a). The task T_b is dependent on T_a during any iteration; and the task T_a during iteration I is dependent on T_a during iteration $I-1$.

```
DO I = 1,2*N
    Ta : X(I) = X(I-1) + 10
    Tb : Y(I) = X(I) + Y(I)
END DO
```

```
DO I = 1,2*N,2
    Ta0 : X(I) = X(I-1) + 10
    Tb0 : Y(I) = X(I) + Y(I)
    Ta1 : X(I+1) = X(I) + 10
    Tb1 : Y(I+1) = X(I+1) + Y(I+1)
END DO
```

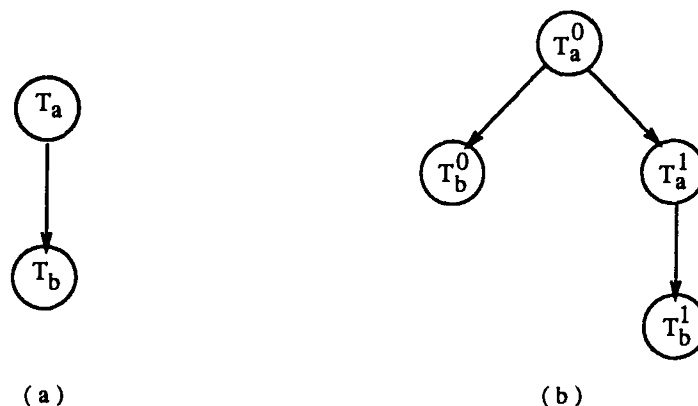
Figure 6–17: (a) An example loop. (b) Loop (a) unrolled once.

Data dependence in loops are classified as follows: (a) *loop-carried dependence* that arises when data are passed between different iterations; and (b) *loop-independent dependence* that arises when data are passed from one task to another within the same iteration. See [ERL91]. The example loop of figure 6–17(a) has a loop-carried dependence from T_a to itself, and there is a loop-independent dependency from T_a to T_b. Both loop-independent and loop-carried dependencies make a loop less parallelizable.

Taking into account the dependencies within a loop body, a dependency graph corresponding to the loop body can be constructed. This dependency graph can then be assigned onto a parallel execution system using one of the assignment heuristics discussed thus far.

The dependency graph corresponding to the loop body of figure 6–17(a) is depicted in figure 6–18(a). Assuming T_a and T_b take unit time for their execution, the makespan of the entire loop is $4N$.

Now consider unrolling the loop. When a single loop is unrolled u times, the loop



(a) Task graph corresponding to the loop body of figure 6–17(a)

(b) Task graph corresponding to the loop body of figure 6–17(b)

Figure 6–18: Task graphs for the example loops

body is replicated u times, the loop control variable is adjusted for each copy and the step value of the loop is multiplied by $u + 1$. Figure 6–17(b) shows the loop of figure 6–17(a) unrolled once.

The dependency graph corresponding to the unrolled loop body of figure 6–17(b) is depicted in figure 6–18(b). Ignoring communication delays, the makespan of this dependency graph on a system with more than one processor is 3. The makespan of the entire loop is thus $3N$. As before, tasks T_a and T_b are assumed to take unit time for their execution. A speed-up of 4/3 has been achieved by unrolling the loop once.

In general, the makespan of an entire loop depends on u (the number of times the loop is unrolled) and the assignment heuristics employed in assigning the loop body. Good assignment heuristics would result in shorter makespans. A discussion on the influence of u on the makespan and the treatment of nested loops are found in [ERL91].

Loop unrolling, however, has one main disadvantage. It produces significant code expansion, increasing the pressure on an instruction cache. Thus an overlapped loop execution (that pipelines the execution of tasks within and across the loop

body) has been proposed [DHB89]. However, such overlapped execution requires complex hardware support in addition to the compiler efforts. When the code size is not of great concern, loop unrolling is a good technique to expose the parallelism in the loops. Good assignment schemes find applicability in assigning the unrolled loops.

6.4 Varying the Size of the Processor Graphs

This experiment examines the effect the size of the processor graph has on the assignment. It uses two-dimensional processor grids of different sizes. These processor grids are constructed using the methodology outlined in section 5.5.2. The grids, however, have wrap-around connections in order to exploit all the processor links. The wrap-around connections enable faster routing.

The experiment uses the assignment schemes ETF, ERT, RMH and DFBN to find assignments of task graphs onto these processor grids and simulates the execution to find the makespans.

MH has a time-complexity cubic in the number of processors. When the processor graphs are large, MH spends a lot of time updating the route tables used in its embedded adaptive routing scheme. Since the earlier experiments have shown that RMH (the restricted version of MH) is equally good, this experiment uses RMH instead of MH and avoids the extra time-complexity of MH.

The average makespans of the task graphs of the QCD benchmark are found to be the same for all assignment schemes on all processor grids. We thus report results from experiments that use the sparse random graphs of section 6.2. Note that the number of tasks (n) in the random graphs varies from 8 to 35 with an average of 21. Table 6–6 summarizes the average makespans of the assignments of these task graphs on different processor grids.

Grid size (m)	ETF	ERT	RMH	DFBN
1×1	623	623	623	623
2×2	248	249	250	238
2×4	218	218	218	224
4×4	212	212	212	228
8×4	213	212	213	230

Table 6–6: Average makespans for different grid sizes

When the grid size is small, DFBN has the minimum average makespan. For large grid sizes, average makespans of the work-greedy assignments are smaller than those of DFBN generated assignments.

The deficiency of DFBN for large grid sizes arises from the fact that it does not make use of the topology of the processor graphs in the same way as work-greedy assignment schemes. DFBN simply uses the 'distance' of processors from the 'most capable' processor in determining processor priorities. In contrast, work-greedy assignments choose a processor according to how quickly the processor can start executing a given task.

However, choosing processors in the way work-greedy assignment schemes do requires a time-complexity higher than that of DFBN. Work-greedy schemes have a time-complexity of $\Theta(n^2m)$ whereas the time-complexity of DFBN is just $\Theta((n + m). \log m + e)$. The improvement the work-greedy schemes achieve for large grid sizes does not thus match the effort and time spent. Judging the schemes by their time-complexity and the results they produce, DFBN is certainly a very promising assignment scheme even for large grid sizes.

It is possible to show a performance guarantee for DFBN for grid sizes larger than the task graph widths (as in the lower rows of table 6–6) if the communication costs are small compared to computation costs. As has been pointed out in section 3.4.5, the makespan generated by DFBN in this case is within a factor of two of the optimal makespan. This same guarantee holds for the work-greedy schemes too.

Thus the makespan of a DFBN-generated assignment is within a factor of two of the makespan of a work-greedy assignment. It will be interesting to investigate if one could prove a similar guarantee when communication costs are comparable to the computation costs.

6.5 Task Assignment on Meiko

This section discusses the viability of testing the assignment schemes for dependency graphs on Meiko [Mei89], the Edinburgh Concurrent Supercomputer. Meiko is a multi-transputer machine consisting of T800 transputers. The set of transputers is divided into domains that a user can reserve for use. Parallel programming is supported through CS Tools [CST], a development toolset for the Meiko. User programs are written in a high-level language, C for example. The programmer needs to think in parallel and code a number of sequential tasks. These tasks operate concurrently, passing messages amongst themselves when necessary. CS Tools provides the programmer with a set of library routines that let explicit message-passing.

Within each domain, tasks communicate using the same library calls regardless of whether or not a direct physical link can be made between the transputers. CS Tools provides all the message re-transmission, multiplexing and buffering that are necessary. The programmer sees the hardware as an idealized, fully connected and homogeneous system. The mechanism that underlies the communication and routing services of CS Tools is called CSN (Computing Surface Network). CSN takes the form of a background process that resides on every transputer of the domain. User tasks do not directly interact with one another; they have to interact via CSN.

In addition to the application program, the user must prepare a configuration file that states which tasks are involved in the application and where they are to run.

Typically, a configuration file consists of task name and transputer id pairs. In the configuration file the user can specify how the transputers are to be connected. If no such specification is given, CS Tools connects the transputers in a line and uses up the spare links for random cross connections.

There are several points about CS Tools worth noting.

- Task assignment is left to the user.

- Processor interconnection is also left to the user. If the user fails to specify the interconnection then an interconnection pattern, that in no way corresponds to the communication structure within the program, is chosen.

- Message-passing is strictly via CSN even if the two communicating tasks reside in the same processor. This is very desirable, but the efficiency depends solely on the way CSN uses the locality in communication to minimize the communication time.

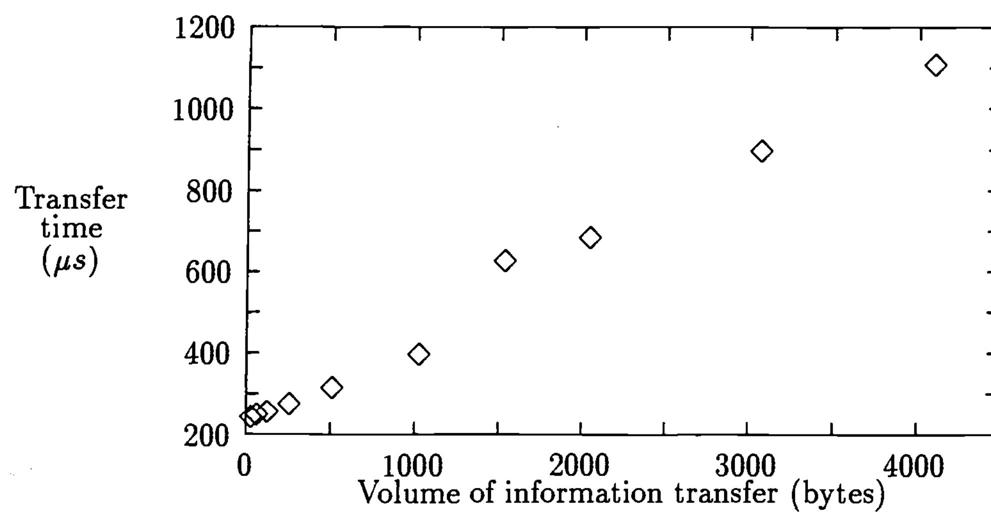


Figure 6–19: Average intraprocessor information transfer time

A simple experiment conducted to measure the time for interprocessor and intraprocessor message-passing shows that the time to pass a message between two

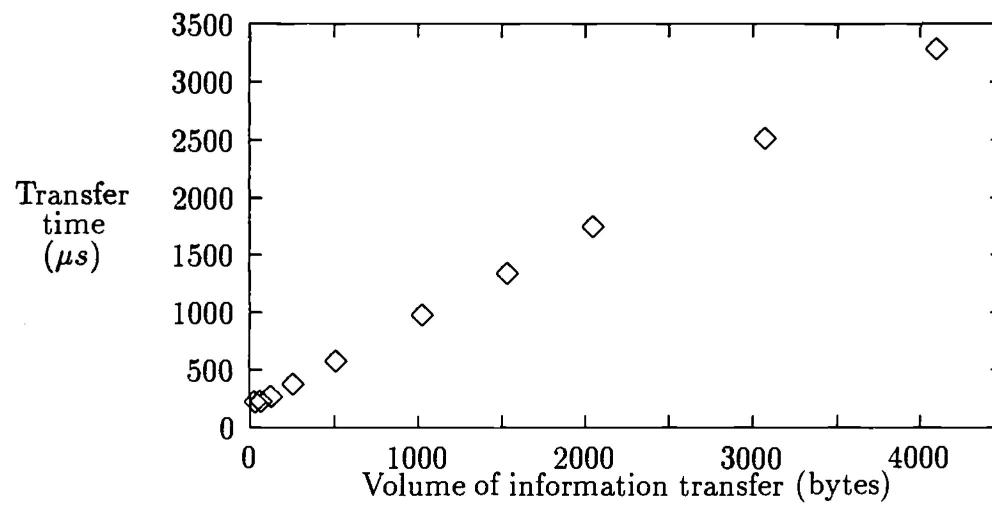


Figure 6–20: Average interprocessor information transfer time

tasks residing in the same processor is comparable to the time to pass the same message between tasks residing in two adjacent processors[4]. Figures 6–19 and 6–20 show that the time t (in microseconds) to transfer a message of volume v (in bytes) is given by

$t \approx 210 + 0.25v$ if the tasks are in the same processor, and

$t \approx 210 + 0.75v$ if the tasks are in adjacent processors.

Intra-processor communication time is more than one third of the inter-processor communication time. This implies that CS Tools fails to exploit the locality to minimize communication times.

Now let us see how viable it is to use the automated assignment schemes for dependency graphs with CS Tools.

In CS Tools, the tasks and their communication and are modelled by an interaction graph. It is, however, possible to restrict the communication so that the tasks

[4]The experiment transfers back and forth a message of size v bytes between two tasks and finds the average time for a single transfer.

would receive messages when they start and send messages when they terminate. By imposing such a restriction one can create a dependency graph model. The tasks forming the dependency graph can then be assigned using one of the automated assignment schemes for dependency graphs, thus relieving the user from having to create a configuration file.

Since CS Tools does not efficiently exploit the locality in communication, the intraprocessor communication cost is comparable to that of the communication cost between any two processors. Automated assignment schemes, however, assume that the intraprocessor communication cost is negligible when compared to the interprocessor communication. Assessing the performance of these assignment schemes on Meiko under CS Tools is thus not appropriate.

In order to have an efficient automated assignment, one needs to take a global view of the application program. There should be a compile-time analysis of the program, and the compiler, rather than the user, must determine the program decomposition. The decomposition may make use of user directives in determining the potential parallelism. This is an exercise in its own merit and we do not deal with it in this thesis. Once such a decomposition is made, the compiler can make use of an appropriate assignment scheme to generate an assignment automatically. In such a case, intraprocessor message-passing can be achieved by passing pointers; and it will be reasonable to assume that the intraprocessor communication cost is negligible, an assumption that does not hold with CS Tools.

6.6 Summary

Comparison of assignment schemes that consider communication costs has not been reported in the literature. This chapter thus presented an extensive set of experimental results comparing these schemes.

Task graphs, either generated randomly or extracted from real program routines, are executed on processor topologies under different assignments and the resulting makespans are compared. For some small task graphs makespans under heuristic assignments are compared against optimal makespans.

The results indicate that DFBN is a promising alternative to work-greedy schemes. It has a time-complexity less than those of work-greedy schemes and achieves a performance better than, or comparable to, that of work-greedy schemes.

These assignment schemes operate under a common assumption: the task graph parameters – the task execution times, volumes of information transfers, etc. – are known prior to the assignment. Accurate estimation of these parameters is often hard due to run-time dependencies, interference from other programs, etc. It is generally thought that such inaccuracies will result in poor assignments. However, the error-sensitivity experiments reported in this chapter suggest that estimation errors have very little impact on the quality of the assignments.

Chapter 7

Summary and Conclusions

This thesis showed some new analytical and experimental results relating to the assignment problem and proposed a new scheme for assignment. It also reported the development of a generic simulation environment for parallel architectures and used this environment to compare the performance of a number of assignment schemes.

Chapter 2 proposed a hierarchical taxonomy for automated assignment schemes. The taxonomy was based on program models. It broadly classified assignment schemes into schemes dealing with interaction graphs and those dealing with dependency graphs. Desirable properties for efficient assignments under different program models were discussed.

As opposed to the assignment of an interaction graph, an assignment of a dependency graph, in general, can be proved to be close to the optimal assignment. Moreover, the explicit temporal information made available by dependency graphs helps in establishing better assignment heuristics.

Chapter 3 thus chose to examine in detail the assignment of dependency graphs. The impact of task ordering on the makespan was established. It was shown that an assignment with a poor task ordering can perform m times worse than an equivalent assignment with a good task ordering, m being the number of processors.

Factors that must determine the task ordering were discussed. Tasks with long execution times, task involving large communication times, tasks with large numbers of successors, tasks with long-length successors and tasks with large memory requirements were identified to be those that need high priority in a task ordering.

Most of the assignment schemes for dependency graphs are work-greedy. Their heuristics is based on satisfying the following rule of thumb: keeping the processors busy will lead to a 'good' assignment. These schemes do not let a processor idle if there is a task the processor could execute. Many of these work-greedy schemes assume that the communication costs are negligible compared to the computation costs. With such an assumption, *any* work-greedy assignment can be proved close to the optimal by no more than a factor of two. Chapter 3 proved such performance guarantees for the work-greedy assignments of:

- independent tasks, and
- dependency graphs with zero communication costs.

The performance guarantees depend on the degree of average software parallelism and the hardware parallelism (i.e. the number of processors available).

Recent assignment schemes extend the work-greedy heuristics to take communication costs into account. However, when the communication costs are taken into account, they lose two important characteristics of zero-communication work-greedy assignment schemes. That is, with arbitrary communication costs, it was shown that

- there is no guarantee that a processor will not idle when there is a task it could execute, and
- a work-greedy assignment can be worse than the optimal assignment by a

large factor (determined by the communication costs along some path in the dependency graph).

There was thus a case for examining an assignment scheme that moves away from the work-greedy heuristics. Chapter 3 proposed such an assignment scheme. It is based on satisfying two desirable properties put forward in chapter 2:

DP1. Assignment of independent tasks to different processors.

DP2. Assignment of dependent tasks to the same processor.

This new scheme, called DFBN (depth-first breadth-next), uses a combination of the familiar depth-first and breadth-first search algorithms to arrive at an assignment.

DFBN does not primarily aim to keep the processors busy. It does not provide any analytical performance guarantee as do the work-greedy schemes. However, it has a time-complexity lower than that of the work-greedy schemes. The time-complexity is linear in the number of tasks and task graph edges.

Comparisons of assignment schemes when the communication costs are zero have been reported in the literature. However, since most of the assignment schemes that consider communication are recent, no comparison of these schemes has yet been published. This thesis thus reported an extensive set of experimental results comparing these recent assignment schemes including DFBN.

An object-oriented simulation platform for parallel systems was developed in order to carry out simulations comparing the performance of assignment schemes. Chapter 5 discussed the design and significant implementation issues involved in the development of this simulation platform. The platform, called Genesis, is generic, in the sense that it can model the key parameters that describe a parallel system: the architecture, the program, the assignment scheme and the routing strategy.

Genesis uses as its basis a sound architectural representation scheme reported in chapter 4.

A number of experiments on the performance of assignment schemes was carried out using Genesis. Chapter 6 reported the results of these experiments. Task graphs, either generated randomly or extracted from real program routines, are executed on processor topologies under different assignment schemes. For some small synthetic task graphs, makespans under the heuristic assignment schemes are compared against the optimal makespans. Genesis was used in constructing the simulation models. Real task graphs were extracted from the subroutines of a Perfect Club benchmark.

The comparison results indicated that DFBN is a promising alternative for work-greedy schemes. It has a time-complexity less than those of the work-greedy schemes and achieves an average performance better than, or comparable to, that of work-greedy schemes. The linear time-complexity of DFBN will make it a suitable scheme for the assignment of large task graphs.

All these assignment schemes assume that the task graph parameters – the task execution times, volumes of information transfer, etc. – are known *a priori*. However, due to many non-deterministic factors, these parameters cannot always be estimated correctly. It is generally thought that such inaccuracies will result in poor assignments. Experiments were conducted to investigate this; the effect of estimation errors on the performance of different assignment schemes were studied. Chapter 6 reported results of these experiments. The results indicated that estimation errors have very little impact on the makespan. They showed that all the assignment schemes exhibit a good deal of insensitivity to estimation errors. Two important implications of these results were pointed out:

1. Inaccuracies in the estimation of task graph parameters can be mostly tolerated. Therefore, an accurate estimation of task graph parameters is not

necessary (nor, in many cases, is possible) to produce reasonably good assignments.

2. An instruction schedule generated for an architecture can be executed on a slightly different architecture, where some instructions have different execution times, without incurring a large penalty.

7.1 Future Directions

This section outlines some of the possible directions in future research.

Chapter 2 noted that the two-step non-work-greedy assignment schemes are complex. These schemes, in the first step, assign the task graph onto an unbounded number of virtual processors that are completely connected and have equal interprocessor communication times. In the second step, they map the virtual processors onto physical processors. Table 7–1 shows the time-complexities of the known two-step non-work-greedy assignment schemes.

Scheme	1st step	2nd step
Kim [Kim88]	$\Theta(ne^3)$	$\Theta(\hat{n}^3 m)$
Sarkar [Sar89]	$\Theta(ne + e^2)$	$\Theta(\hat{n}nm + e)$

Table 7–1: Time-complexities of two-step non-work-greedy schemes

Recall that n is the number of tasks and m is the number of processors. The number of virtual processors that have been actually used is $\hat{n}$ ($\leq n$). The time-complexity of DFBN is much lower than the time-complexities of these two-step non-work-greedy schemes. Since these two-step schemes are complex and require large time-complexities, the comparison experiments reported in chapter 6 did not take them into account. However, it will be interesting to compare the performance of these two-step non-work-greedy schemes in a framework similar to the one reported in chapter 6.

Chapter 3 showed the impact task ordering has on the makespan. Poor task orderings can result in long makespans. The factors that determine the task ordering were pointed out. Tasks with long execution times, task involving large communication times, tasks with large numbers of successors, tasks with long-length successors and tasks with large memory requirements were identified to be those that need high priority in a task ordering. But a task graph may contain a mixture of such tasks. How should one determine a unique task ordering in a task graph like this? DFBN, when determining a task ordering, took a weighted sum of various priorities. Is there be a better way of doing this? We need more exploration – either analytical or experimental – to provide an answer. An interesting direction would be to explore the relative significance of the task priorities.

Another direction for future work is to find analytical performance guarantees for DFBN. This may be hard, since DFBN does not predict *when* to execute a task; it only finds *where* to execute it. With no start and finish times of the tasks predicted, it may be hard to quantify the makespan in terms of the task and processor graph parameters.

If two concurrently-executable tasks communicate heavily with a common successor task, it may be advantageous to assign these two concurrently-executable tasks to the same processor (that will then be assigned the common successor). Such an assignment produces a *communication-based clustering*. DFBN and the other work-greedy assignment schemes always assign these concurrently-executable tasks to different processors, if there are sufficient processors. That is, they do not produce a communication-based clustering. When the tasks graphs have a small computation to communication ratio, there is a case for extending an assignment heuristics to produce communication-based clusterings.

The thesis did not take into account the possibility of task replication. There is a definite performance gain by executing some tasks on more than one processor when the average communication to computation ratio is more than one [KL87]. However, the time-complexity of the assignment scheme will then be large. For

instance, the scheme proposed by Kruatrachue and Lewis [KL87] has a time-complexity of $\Theta(n^4m)$. One future direction is to compare the assignment schemes reported in chapter 3 to that of DSH. Besides, it will be interesting to extend theorem 3.5 of chapter 3 taking task replication into account.

On a more general direction, it will be interesting to explore the suitability of assignment schemes to programming languages. Automated assignment schemes are well suited to dataflow languages [Sar89]. Other languages need to be translated into dependency graphs first. For the translation exercise, tools that perform dependency analysis of programs and generate dependency graphs will be useful. If the language does not support explicit parallelism, then such tools should also be able to automatically detect and extract parallelism embedded in the programs. Tool kits such as Sigma II [Sig92] take this direction.

Bibliography

[ACD74] Thomas L Adam, K M Chandy, and J R Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[Alm85] G S Almasi. Overview of parallel processing. *Parallel Computing*, 2:191–203, 1985.

[AM90] Mayez A Al-Mouhamed. Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Transactions on Software Engineering*, 16(12):1390–1401, December 1990.

[AP91] Silvano Antonelli and Susanna Pelagatti. On the complexity of the mapping problem for massively parallel architectures. Technical Report TR-5/91, Dipartimento di Informatica, Università di Pisa, March 1991.

[B+89] M Berry et al. The perfect club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3:5–40, 1989.

[BDW86] Jacek Blàzewicz, Mieczyslaw Drabowski, and Jan Weglarz. Scheduling multiprocessor tasks to minimize schedule length. *IEEE Transactions on Computers*, C-35(5):389–393, May 1986.

[BLUL85] Graham Birtwistle, Greg Lomow, Brian Unger, and Paul Lucker. Process style packages for discrete event modelling: Experience from the transaction, activity and event approaches. *Transactions of the Society for Computer Simulation*, 2(1):27–56, May 1985.

[BMRS88] F Warren Burton, G P McKeown, and V J Rayward-Smith. On process assignment in parallel computing. *Information Processing Letters*, 29:31–34, 1988.

[Bok81] Shahid H Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):207–214, March 1981.

[Bok88] Shahid H Bokhari. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Transactions on Computers*, 37(1):48–57, Jan 1988.

[BPTS91] Peter L Bird, Uwe F Pleban, Nigel P Topham, and Henrik Scheuer. Semantics driven computer architectures. In *Proceedings of Parallel Computing*, September 1991.

[BW91] Graham Brightwell and Peter Winker. Counting linear extensions is #P-complete. In *Proceedings 23rd STOC*, pages 175–181, 1991.

[CA82] Timothy C K Chou and Jacob A Abraham. Load balancing in distributed systems. *IEEE Transactions on Software Engineering*, SE-8(4):401–412, July 1982.

[CK88] Thomas L Casavant and Jon G Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, Feb 1988.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer performance evaluation and the perfect benchmarks. In *Proceedings of the International Conference on Supercomputing*, pages 254–266, Amsterdam, The Netherlands, June 11-15, 1990. ACM Press.

[CM82] N Cabbibo and E Marinari. A new method of updating SU(N) matrices in computer simulations of gauge theories. *Physics Letters*, 119B(387), 1982.

[Cof76] E G Coffman, editor. *Computer and Job Shop Scheduling Theory.* John Wiley and Sons, 1976.

[Col89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[CST] CS Tools: A technical overview. Technical Report S0205-09S, Meiko Ltd.

[Cve87] Z Cvetanovic. The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems. *IEEE Transactions on Computers*, C-36(4):421–432, April 1987.

[Dad91] Luigi Dadda. The evolution of computer architectures. In *Proceedings of the 5th Annual European Computer Conference: CompEuro '91*, pages 9–16, Bologna, Italy, May 13-16, 1991. IEEE Computer Society Press.

[Das90] Subrata Dasgupta. A hierarchical taxonomic system for computer architectures. *Computer*, pages 64–74, March 1990.

[DHB89] James C Dehnert, Peter Y-T Hsu, and Joseph P Bratt. Overlapped loop support in cydra 5. In *Proceedings ASPLOS-III*, pages 26–38. ACM Press, 1989.

[DSS88] J G Donnett, M Starkey, and D B Skillicorn. Effective algorithms for partitioning distributed programs. In *Proceedings 7th Annual International Conference on Computers and Communications*, Scotsdale AR, March 1988.

[Dun90] Ralph Duncan. A survey of parallel computer architectures. *Computer*, pages 5–16, February 1990.

[Efe82] Kemal Efe. Heuristic models of task assignment scheduling in distributed systems. *Computer*, pages 50–56, June 1982.

[ERL90] Hesham El-Rewini and T G Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138–153, 1990.

[ERL91] Hesham El-Rewini and Ted Lewis. Loop scheduling on distributed-memory parallel processors. Technical Report 91-60-1, Computer Science Department, Oregon State University, Corvallis OR, 1991.

[FB73] Eduardo B Fernandez and Bertram Bussell. Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Transactions on Computers*, C-22(8):745–751, August 1973.

[Fly72] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21:948–960, 1972.

[GJ79] Michael R Garey and David S Johnson. *Computers and Intractability - A Guide to the Theory of NP-Completeness.* W H Freeman and Company, 1979.

[GKS87] Michael Granski, Israel Koren, and Gabriel M Silberman. The effect of operation scheduling on the performance of a data flow computer. *IEEE Transactions on Computers*, C-36(9):1019–1029, Sept 1987.

[GLLK79] R L Graham, E L Lawler, J K Lenstra, and A H G Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 3:287–326, 1979.

[GPC88] J L Gaudiot, J I Pi, and M L Campbell. Program graph allocation in distributed multicomputers. *Parallel Computing*, 7:227–247, 1988.

[Gra69] R L Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, March 1969.

[Gra76] R L Graham. Bounds on the performance of scheduling algorithms. In E G Coffman, editor, *Computer and Job Shop Scheduling Theory*, pages 165–227. John Wiley and Sons, 1976.

[Gru91] Dirk C Grunwald. A users guide to AWESIME: An object oriented parallel programming and simulation system. Technical Report CU-CS-552-91, Department of Computer Science, University of Colorado, November 1991.

[GVY90] Apostolos Gerasoulis, Sesh Venugopal, and Tao Yang. Clustering task graphs for message passing architectures. In *Proceedings of the International Conference on Supercomputing*, pages 447–456, Amsterdam, The Netherlands, June 11-15, 1990. ACM Press.

[HCAL89] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal of Computing*, 18(2):244–257, April 1989.

[HMS] G Haring, W Mullner, and K Sharma. The application of simulated annealing for optimal module placement in a multiprocessor system. Preliminary Draft.

[Hoa78] C A R Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hoc85] Roger W Hockney. MIMD computing in the USA – 1984. *Parallel Computing*, 2:119–136, 1985.

[Hoc87] Roger W Hockney. Classification and evaluation of parallel computer systems. In *Lecture Notes in Computer Science*, volume 295, pages 13–25. Springer-Verlag, 1987.

[HS78] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures*. Computer Science Press, 1978.

[Jaf80] Jeffrey M Jaffe. Bounds on the scheduling of typed task systems. *SIAM Journal of Computing*, 9(3):541–551, August 1980.

[Jum90] J Robert Jump. *xSIM User's Manual*. Dept of Electrical and Computer Engineering, Rice University, July 1990.

[K+83] S Kirkpatrick et al. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[Kal87] L V Kalé. Comparing the performance of two dynamic load distribution methods. Technical Report UIUCDC-R-87-1776, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1987.

[Kim88] Sung Jo Kim. A general approach to multiprocessor scheduling. Technical Report TR-88-04, Department of Computer Science, University of Texas, Austin, Texas, February 1988.

[KL87] Boontee Kruatrachue and Ted Lewis. Duplication scheduling heuristics, a new precedence task scheduler for parallel systems. Technical Report 87-60-3, Computer Science Department, Oregon State University, Corvallis OR, 1987.

[KN84] Hironori Kasahara and Seinosuke Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33(11):1023–1029, Nov 1984.

[Kri90] Sanjay M Krishnamurthy. A brief survey of papers on scheduling for pipelined processors. *SIGPLAN Notices*, 25(7):97–106, July 1990.

[LA87] S Lee and J K Aggarwal. A mapping strategy for parallel processing. *IEEE Transactions on Computers*, C-36(4):433–442, April 1987.

[LHCA88] Chung-Yee Lee, Jing-Jang Hwang, Yuan-Chieh Chow, and Frank D Anger. Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, 7(3):141–145, June 1988.

[Lip91] Stanley B Lippman. *C++ Primer*. Addison-Wesley Publishing Company, 1991.

[LK87] Frank C H Lin and Robert M Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, SE-13(1):32–38, January 1987.

[LL78] Jane W S Liu and C L Liu. Performance analysis of multiprocessor systems containing functionally dedicated processors. *Acta Informatica*, 10:95–104, 1978.

[Mac87] M H MacDougall. *Simulating Computer Systems: Techniques and Tools.* MIT Press, 1987.

[Man91] Sathiamoorthy Manoharan. A taxonomy for assignment in parallel processor systems. In *Proceedings of the 5th Annual European Computer Conference: CompEuro '91*, pages 143–147, Bologna, Italy, May 13-16, 1991. IEEE Computer Society Press.

[Man92] Sathiamoorthy Manoharan. Genesis: A generic simulation subsystem for parallel architectures. In *Proceedings of the 6th Annual European Computer Conference: CompEuro '92*, The Hague, The Netherlands, May 4-8, 1992. IEEE Computer Society Press.

[McN59] R McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6, October 1959.

[ME67] David Martin and Gerald Estrin. Models of computational systems – cyclic to acyclic graph transformation. *IEEE Transactions on Electronic Computers*, EC-16(1):70–79, Feb 1967.

[Mei89] *Computing Surface Hardware Reference Manual.* Meiko Ltd, 1989.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall International, 1989.

[MT90] Sathiamoorthy Manoharan and Nigel P Topham. A general bound on schedule length for independent tasks. *Parallel Computing*, 16(1):69–73, November 1990.

[MT91] Sathiamoorthy Manoharan and Peter Thanisch. Assigning dependency graphs onto processor networks. *Parallel Computing*, 17(1):63–73, April 1991.

[NE88] Kathleen M Nichols and John T Edmark. Modeling multicomputer systems with PARET. *Computer*, pages 39–48, May 1988.

[NW88] David M Nicol and Frank H Willard. Problem size, parallel architecture, and optimal speedup. *Journal of Parallel and Distributed Computers*, 5:404–420, 1988.

[PB87] C D Polychronopoulos and U Banerjee. Processor allocation for horizontal and vertical parallelism and related speedup bounds. *IEEE Transactions on Computers*, C-36(4):410–420, April 1987.

[RAP87] D A Reed, L M Adams, and M L Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Transactions on Computers*, C-36(7):845–858, July 1987.

[RF87] D A Reed and R M Fujimoto. *Multicomputer Networks - Messaged based Parallel Processing.* MIT Press, 1987.

[RG69] C V Ramamoorthy and M J Gonzalez. A survey of techniques for recognizing parallel processable streams in computer programs. In *Proceedings AFIPS Fall Joint Computer Conference*, pages 1–17, 1969.

[RS87] V J Rayward-Smith. UET scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.

[Sah84] Sartaj Sahni. Scheduling multipipeline and multiprocessor computers. *IEEE Transactions on Computers*, C-33(7):637–645, July 1984.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Cambridge MA, 1989.

[SE87] Ponnuswamy Sadayappan and Fikret Ercal. Nearest-neighbour mapping of finite element graphs onto processor meshes. *IEEE Transactions on Computers*, C-36(12):1408–1424, Dec 1987.

[Sig92] *Sigma II Documentation.* University of Illinois, Urbana-Champaign, 1992.

[Ski88] David B Skillicorn. A taxonomy for computer architectures. *Computer*, pages 46–57, November 1988.

[Squ90] Mark S Squillante. Issues in shared-memory multiprocessor scheduling: A performance evaluation. Technical Report 90-10-04, Dept of Computer Science, University of Washington, October 1990.

[ST85] Chien-Chung Shen and Wen-Hsiang Tsai. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, C-34(3):197–203, March 1985.

[Str88] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, pages 10–20, May 1988.

[SWP90] Behrooz Shirazi, Mingfang Wang, and Girish Pathak. Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributed Computing*, 10:222–232, 1990.

[Tow86] Don Towsley. Allocating programs containing branches and loops within a multiprocessor system. *IEEE Transactions on Software Engineering*, SE-12(10):1018–1024, Oct 1986.

[TSS88] Charles P Thacker, Lawrence C Stewart, and Edwin H Satterthwaite. Firefly: A multiprocessor workstation. *IEEE Transactions on Computers*, 37(8):909–920, August 1988.

[UBP81] Brian Unger, Don Bidulock, and Jim Parker. OASIS 3.0 reference manual. Technical Report 81/58/10, Dept of Computer Science, University of Calgary, 1981.

[Ull76] J D Ullman. Complexity of sequencing problems. In E G Coffman, editor, *Computer and Job Shop Scheduling Theory*, pages 139–164. John Wiley and Sons, 1976.

[V+85] D Vrsalovic et al. The influence of parallel decomposition strategies on the performance of multiprocessor systems. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 396–405, 1985.

[V+88] D Vrsalovic et al. Performance prediction and calibration for a class of multiprocessors. *IEEE Transactions on Computers*, 37(11):1353–1365, Nov 1988.

[Val79] L G Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8:189–201, 1979.

[VLL90] B Veltman, B J Lageweg, and J K Lenstra. Multiprocessor scheduling with communication delays. *Parallel Computing*, 16:173–182, 1990.

[WG88] Min-You Wu and Daniel D Gajski. A programming aid for hypercube architectures. *The Journal of Supercomputing*, 2(3):349–372, November 1988.

[Win84] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley, 1984.

[WM85] Yung-Terng Wang and Robert J T Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, C-34(3):204–217, March 1985.

[YG91] Tao Yang and Apostolos Gerasoulis. A fast scheduling algorithm for DAGs on an unbounded number of processors. In *Proceedings of Supercomputing*, pages 633–642, Albuquerque, November 18-22, 1991. IEEE Press.

Appendix A

Definition of the Watchdog

Each atom of Genesis owns a watchdog in order not to busy-wait for the arrival of items on the atom's input ports. The definition of the watchdog is presented here.

Watchdog is a boolean variable w which can be either **true** or **false**. The only permissible operations on w are *wait(w)* and *notify(w)*. These two are indivisible or atomic operations:

wait(w) : **if** ($!w$) { set w; go to sleep; }
else error;

notify(w) : **if** (w) { wake up the process waiting on w; reset w; }
else do nothing;

Two processes cannot wait on the same watchdog. That is, only one process has the right to set w; however, any process can reset w.

Another example of the use of watchdogs is in interrupt-handlers. An interrupt-handler is a process that needs to be waken up only when one or more other

processes needs service. Hence, an interrupt-handler can own a watchdog and wait on it. The processes needing service may notify the watchdog, thus waking up the interrupt-handler.

Appendix B

Dynamic Behaviour of a Processor Executing a Dependency Graph

The following C++ code describes the dynamic behaviour of a processor executing a dependency graph (consisting of DTask objects). The code would appear in the member function `main` of the processor class derived from Processor. The routing mechanism in the code assumes that the processors are connected to form a grid. See section 5.5.2.

```
for ( ; ; ) { /* repeat for ever */
    /* execute the ready task in the task pool */
    /* and convey their outputs to their successors. */
    while ( ( task = pop_pool() ) ≠ 0 ) {
        ::hold(task→exec_time() / speed()); /* execute task */
        /* for all successors of task do the following */
            destination_task = task→succ();
            destination_processor = destination_task→where();
            if ( destination_processor == this ) { /* successor in this processor */
                destination_task→input(); /* the successor gets an input */
                /* if the succ is ready after getting the input, */
                /* add it to the task pool. */
```

```
            if ( destination_task→ready() )
                add2pool(destination_task, destination_task→priority());
        }
        else { /* successor elsewhere */
            /* form a new message packet */
            /* and send it to the destination processor */
            send(get_port(destination_processor→xid(),
                destination_processor→yid()), msg);
        }
    /* end for */
}

/* nothing to do now. wait for some input port to get a message */
wait();

/* there is message. */
/* if it is for this processor, input it to the task it is destined to. */
/* otherwise route the message. */
for ( int p = 0; p < inports(); ++p )
    if ( in_ready(p) ) { /* input port p has a message */
        msg = (Mesg *) recv(p); /* receive the message */
        if ( msg→where() == this ) {
            /* task t to which the message is destined is in this processor */
            Task *t = (Task *)msg→destination();
            /* t gets an input */
            t→input();
            /* if t is ready, add it to the task pool */
            if ( t→ready() )
                add2pool(t, t→priority()):
        }
        else { /* message destined elsewhere */
            /* route the message */
```

```
send(get_port((msg→where())→xid(),
    (msg→where())→yid()), msg);
```

Appendix C

Tables

G_P^1, G_P^2 and G_P^3 refer to the processor graphs of figure 6–2. G_T^1, G_T^2 and G_T^3 refer to the task graphs of figure 6–3.

G_T	G_P	*A* best-case assignment	Makespan	*A* worst-case assignment	Makespan
G_T^1	G_P^1	P_0: 0, 2, 5, 7 P_1: 6, 3 P_2: 1, 4	32	P_0: 0, 7 P_1: 1, 2, 3, 4, 5, 6 P_2: —	65
G_T^1	G_P^2	P_0: 3, 5, 6, 7 P_1: 0, 2, 1, 4	39	P_0: 0, 7 P_1: 1, 2, 3, 4, 5, 6	65
G_T^1	G_P^3	P_0: 0, 2, 5, 7 P_1: 3, 6 P_2: — P_3: 1, 4	32	P_0: — P_1: 5, 1, 6, 2, 7 P_2: 3 P_3: 0, 4	68
G_T^2	G_P^1	P_0: 0, 2, 3, 5, 8, 9 P_1: 4, 6, 7 P_2: 1	53	P_0: 0, 5, 9 P_1: 1, 2, 4, 6, 3, 7, 8 P_2: —	92
G_T^2	G_P^2	P_0: 0, 2, 3, 5, 8, 9 P_1: 1, 4, 6, 7	54	P_0: 0, 5, 9 P_1: 1, 2, 4, 6, 3, 7, 8	92
G_T^2	G_P^3	P_0: 0, 2, 3, 5, 8, 9 P_1: 4, 6, 7 P_2: — P_3: 1	53	P_0: 0, 5, 9 P_1: — P_2: 1, 2, 4, 6, 3, 7, 8 P_3: —	98
G_T^3	G_P^1	P_0: 1, 3, 6 P_1: 2, 5 P_2: 0, 4	17	P_0: 0, 1, 2, 6 P_1: 3, 4, 5 P_2: —	39
G_T^3	G_P^2	P_0: 1, 3, 5, 6 P_1: 0, 2, 4	22	P_0: 0, 1, 2, 6 P_1: 3, 4, 5	39
G_T^3	G_P^3	P_0: 1, 3, 6 P_1: 2, 5 P_2: — P_3: 0, 4	17	P_0: 6 P_1: 4, 5 P_2: 3 P_3: 0, 1, 2	42

Table C–1: Best-case and worst-case assignments and makespans

G_T	G_P	ETF	ERT	MH/RMH	DFBN
G_T^1	G_P^1	P_0: 3, 6 P_1: 2, 5, 7 P_2: 0, 1, 4	P_0: 3, 6 P_1: 2, 5, 7 P_2: 0, 1, 4	P_0: 0, 2, 5, 7 P_1: 1, 4 P_2: 3, 6	P_0: 0, 2, 5, 7 P_1: 6, 3 P_2: 1, 4
G_T^1	G_P^2	P_0: 3, 5, 6, 7 P_1: 0, 1, 2, 4	P_0: 2, 3, 5, 7 P_1: 0, 1, 4, 6	P_0: 0, 2, 3, 4 P_1: 1, 5, 6, 7	P_0: 0, 2, 5, 7 P_1: 1, 4, 6, 3
G_T^1	G_P^3	P_0: 3, 6 P_1: — P_2: 2, 5, 7 P_3: 0, 1, 4	P_0: 3, 6 P_1: — P_2: 2, 5, 7 P_3: 0, 1, 4	P_0: 0, 2, 5, 7 P_1: 1, 4 P_2: — P_3: 3, 6	P_0: 0, 2, 5, 7 P_1: 6, 3 P_2: — P_3: 1, 4
G_T^2	G_P^1	P_0: 2, 6, 7 P_1: 3, 5, 8, 9 P_2: 0, 1, 4	P_0: 2, 6 P_1: 3, 5, 7 P_2: 0, 1, 4, 8, 9	P_0: 0, 1, 4, 7 P_1: 2, 3, 5, 8, 9 P_2: 6	P_0: 0, 2, 5, 3, 8, 9 P_1: 6 P_2: 1, 7, 4
G_T^2	G_P^2	P_0: 2, 6, 8, 9 P_1: 0, 1, 3, 4, 5, 7	P_0: 2, 3, 5, 7 P_1: 0, 1, 4, 6, 8, 9	P_0: 0, 1, 6, 4, 7 P_1: 2, 3, 5, 8, 9	P_0: 0, 2, 5, 3, 8, 9 P_1: 1, 7, 6, 4
G_T^2	G_P^3	P_0: 2, 6 P_1: 7 P_2: 3, 5, 8, 9 P_3: 0, 1, 4	P_0: 2, 6 P_1: 8, 9 P_2: 3, 5, 7 P_3: 0, 1, 4	P_0: 0, 1, 4, 7 P_1: 2, 3, 5, 8, 9 P_2: 6 P_3: —	P_0: 0, 2, 5, 3, 8, 9 P_1: 6 P_2: — P_3: 1, 7, 4
G_T^3	G_P^1	P_0: 0, 4 P_1: 1, 3, 6 P_2: 2, 5	P_0: 0, 4 P_1: 1, 3 P_2: 2, 5, 6	P_0: 0, 4 P_1: 1, 5 P_2: 2, 3, 6	P_0: 0, 4, 6 P_1: 2, 5 P_2: 1, 3
G_T^3	G_P^2	P_0: 1, 4, 5, 6 P_1: 0, 2, 3	P_0: 1, 4, 5, 6 P_1: 0, 2, 3	P_0: 0, 3, 5, 6 P_1: 1, 2, 4	P_0: 0, 4, 6 P_1: 1, 3, 2, 5
G_T^3	G_P^3	P_0: — P_1: 0, 4 P_2: 1, 3, 6 P_3: 2, 5	P_0: — P_1: 0, 4 P_2: 1, 3, 6 P_3: 2, 5	P_0: 0, 4 P_1: 1, 5, 6 P_2: 2, 3 P_3: —	P_0: 0, 4, 6 P_1: 2, 5 P_2: — P_3: 1, 3

Table C–2: Task partitions under various assignment schemes