Models of Machines and Modules for Mapping to Minimise Makespan in Multicomputers

Michael G. Norman
Edinburgh Parallel Computing Centre
University of Edinburgh
Edinburgh, Scotland
EH9 3JZ

Peter Thanisch
Department of Computer Science
University of Edinburgh
Edinburgh, Scotland
EH9 3JZ

Abstract

It is now more than a quarter of a century since researchers started publishing papers on mapping strategies for distributing computation across the computation resource of multiprocessor systems. There exists a large body of literature on the subject, but there is no commonly-accepted framework whereby results in the field can be compared. Nor is it always easy to assess the relevance of a new result to a particular problem. Furthermore, changes in parallel computing technology have made some of the earlier work of less relevance to current multiprocessor systems.

Versions of the mapping problem are classified, and research in the field is considered in terms of its relevance to the problem of programming currently available hardware in the form of a distributed memory multiple instruction stream multiple data stream computer: a *multicomputer*.

Categories and Subject Descriptors: C.1.2 [**Multiple Data Stream Architectures**]:Multiprocessors, Parallel Processors; C.2.1 [**Network Architecture and Design**]; F.1.2 [**Models of Computation**]: Parallelism and Concurrency.

General Terms: Multicomputers.

Additional Key Words and Phrases: Partitioning, Mapping, Scheduling, Multicomputer Load Balancing.

Symbol	Usage
A, B	An instance of a model
F	A set of module to processor mapping functions
L	A set of interprocessor links
M	Makespan
P	A set of processors
Q	An undirected graph of processors
R_m	The total cost of a mapping g_m
S_m	The set of valid schedules for a mapping function g_m
T_{Calc}	Processing time spent doing useful calculation
T_{Comm}	Processing time associated with message transfer
T_{Fin}	Time processors spend waiting for others to finish
T_{House}	Processing time spent on "housekeeping"
T_{Idle}	Time processors spend idle
T_{Init}	Processing time spent initialising message transfer
T_{Inter}	Processing time associated with "interference" between modules
T_{Recalc}	Processing time spent on repeated calculation
T_{Route}	Processing time spent through-routing messages
T_{Sched}	Processing time spent on scheduling-related computation
T_{Term}	Processing time spent receiving messages
T_{Wait_D}	Time processors spend waiting for messages that are in transit
T_{Wait_S}	Time processors spend waiting for messages that have not yet been sent
T_{Wait}	Time processors spend waiting for messages
U_m	The computation cost of a mapping g_m
V_m	The communication cost of a mapping g_m
$Wait(\gamma)$	Idle time before executing task γ
$Wait_D(\gamma)$	Idle time spent waiting for messages in transit before executing task γ
$Wait_S(\gamma)$	Idle time spent waiting for unsent messages before executing task γ
Δ	A set of directed arcs between tasks
Γ	A set of tasks
Λ	A task dag
γ, δ	Tasks
τ	An integer communication delay
a, b, c, d	Parameterisations of quiet network message latency
f	Cost functions (possibly subscripted)
g_m	A function returning the set of processors to which a processor is mapped
h_m	The inverse function of g_m
l	The cardinality of a set of modules
m	The index of a mapping function
n	The cardinality of a set of processors
o	An ordered set of tasks
p, q	Processors
s	A scheduling function
t	Task deadline
v	Interprocessor links travelled by a message
w	Length of a message
x	The cardinality of the set of modules mapped to a processor

Contents

1 Introduction

One of the key problems in the area of parallel computing is that of mapping computation across processors. There has been a huge number of papers published dealing with this problem, over more than twenty years, and it is difficult to understand how they relate to each other in the context of programming a particular architecture. This review concentrates on the distributed memory multiple instruction stream multiple data stream computer which we shall refer to as the multicomputer.

The class of parallel computer systems that we have in mind has the following characteristics. Each processor has its own memory. There is no global memory. Processors communicate by passing messages. We allow the possibility that processors may execute asynchronously. Informally, the processors need not all be on the same chip, but should be in the same box, that is we are not interested in distributed systems. With regard to the software, we consider a "space shared machine" where programs do not compete for resources. We are interested in the time to completion of a program rather than the throughput of jobs.

1.1 Scope of Literature Under Review

With the current interest in the multicomputer architecture—one can buy multicomputers off the shelf from a number of manufacturers—we believe it is is timely to consider the way in which work in related fields can be usefully applied to the multicomputer mapping problem. By restricting our discussion to multicomputers we are excluding related areas in scheduling for VLIW architectures and hardware data-flow architectures (eg. McDowell and Appelbe [1986], and the complexity results of Fellows and Langston [1988]); we are not directly interested in the issues in scheduling pipelined or vector processors, as reviewed recently by Krishnamurthy [1990]; and we are not interested in issues of language design for multicomputers (see Bal *et al.* [1989]). We ignore the interactions of multicomputer programs with operating system facilities, such as the file system, such as those discussed by Chu and Lan [1987]. We shall be drawing upon work on analysis of operating systems and of distributed systems, and on work in scheduling theory. Having restricted ourselves to just this small part of the literature, it is still clear that the possible confusion between results comes from the differences in the models that are being used in the various papers. In fact the main thrust of the review is to describe the various models at a level of abstraction that enables the multicomputer programmer to ascertain the relevance of a mapping strategy to their particular programming environment and application.

In order to further limit the range of our discussion, although we consider models of parallel computers, we frame our discussion from the point of view of their ability to effectively model multicomputers rather than the efficiency of a multicomputer in emulating any given architectural model. Thus we make no discussion of the universality of models over the architecture such as that proposed by Valiant [1990] for his XPRAM, and indeed our discussion of PRAMs is limited to considering the availability of PRAM algorithms for mapping to our architectural models which are not framed in terms of the PRAM.

The review is restricted to work published in international journals (except where important results published elsewhere have not been included in such journals) and only covers work published up to and including the calendar year 1990. Our intended audience comprises not only the users of multicomputers but also those intending to do research on the mapping problem for multicomputers. Mapping and scheduling are hard computational problems so it is not surprising that a large proportion of the review is devoted to the published complexity results

on the mapping problem, ie. the "negative" complexity results[1] and the "positive" results describing heuristics, approximations and the special cases that avoid NP-hardness. Unlike Casavant and Kuhl [1988] whose taxonomy for describing mapping strategies is determined by the types of algorithms being used[2], we are interested primarily in the way the problem is being modelled. That is, we attempt to make explicit the assumptions of the models of parallel computation that are implicit in the aforementioned results. In doing so, we hope to make it easier for the reader to assess the relevance of a result to a particular concrete mapping problem.

1.2 An Informal Discussion of the Problems

In general, a multicomputer is harder to use than the so called *von neumann*, ie. sequential, computer. A fact easily overlooked by computer scientists is that, by and large, people only use a multicomputer if they wish their software to run faster. In order to introduce our terminology and notation, we look at the problem of programming a multicomputer from the point of view of a programmer who is developing some software for a multicomputer and wishes to optimise its performance.

There are several different ways to formulate this "optimisation" problem. For example: for software such as operating systems, it is useful to construe the optimisation in terms of maximising the throughput of jobs. In a real time system design problem, the designer may be interested in the minimum number of processors that can guarantee a particular level of performance. Alternatively, the number of processors in the multicomputer may be fixed, and the performance of the software may be optimised with respect to a single program.

The throughput based formulation has been used by a number of authors eg. Baccelli and Liu [1990] and Bokhari [1981]; the problem of finding a minimum number of processors has been addressed by others such as Fernández and Bussel [1973], Al-Mouhammed [1990] and Houstis [1990]; but in this paper we formulate the problem for a fixed number of processors and a single program and therefore consider the minimisation of *makespan*, which is the elapsed wall-clock time between the moment when the multicomputer starts to execute the program to the moment at which the result is presented.

The work that we shall be reviewing makes the assumption that the designer of the software, be it human or machine, has some abstract model of his, her or its program. As with any modelling exercise, this abstraction will emphasise some features of the program's behaviour at the expense of, or even to the exclusion of, some other features. In order to identify the problems we shall begin by making a common assumption: that the program can be represented as a set of tasks which communicate their results to other tasks only on termination, and that the structure of the computation can be represented as a task *graph* in which a directed arc connects a pair of distinct tasks if and only if the task at the head of the arc requires the results of computation from the task at the tail of the arc.

The task digraph represents a partial order for an agenda of activities. It may be, for example, that at two or more separate stages in the computation it is necessary to perform a given computation—for example a sort operation on two different sets of data. This would be represented by two separate tasks/nodes in the task digraph. Consequently it is clear that in a valid task graph, there cannot exist a cycle in which a given task requires the results of a task to which it, in turn (either directly or indirectly) supplies results. The task graph is always a

[1] We assume that readers are familiar with the concepts of computational complexity as outlined, for example, by Garey and Johnson [1979].

[2] In their terms, our discussion is restricted, for the most part, to the *static* mapping problem.

directed acyclic graph or dag.

Let us imagine that the program's designer can model the computation as a set of tasks with dependency arcs. Most researchers in the area of the mapping problem would expect the model to be labelled: the task nodes are labelled with the execution times of the tasks, and the communications arcs are labelled with the volume of the communication that is required to flow from the tail task to the head task.

Assuming that such a labelled task graph can be created—no mean feat in itself—the mapping problem becomes the problem of mapping tasks to processors and giving the processors local schedules for their tasks, subject to two constraints: no processor is executing more than one task at a time; and the tasks that are defined to precede a given task have finished executing before that task is started. We can consider two ways of doing this: we could decide upon the mapping to processors before we start the processing, or we could map tasks to processors on the fly during the computation. The former is referred to as the static mapping problem the latter as the dynamic mapping problem. In this pedagogical section of the review we shall consider only the former, although dynamic mapping is discussed to some extent in later sections.

Let us consider an example task graph (shown in Figure 1), one which has a single sink node corresponding to the task that presents the final result to the user, and a single source node corresponding to the task that inputs the user's request to start the computation.

A common assumption made by those considering the mapping problem is that inter-task communications delay is zero if both the tasks involved in the communication are assigned to the same processor. In this case, we may wish to identify the *critical path* in the task graph, ie. the path from source to sink such that the sum of the node weights is maximised. The significance of the critical path is that regardless of the number of processors, the sum of the execution times labelling the nodes on the critical path represents the minimum achievable makespan for this task graph (assuming the program's designer got the labels correct).

This path has been identified on Figure 2, and corresponds to the shaded nodes. We can consider mapping the tasks to two processors (there is no point in using more, since the maximum *width* of the dag is two). We might generate a schedule which can be represented by the Gantt chart shown in Figure 3. Here we show the activity of the processors, through time, and also show communication events as arrows, and the buffering of messages that are received. The time to completion of the schedule we have used is the makespan. Since to one of the two processors we have mapped the critical path, and only the critical path, the makespan of the schedule is minimal. If this minimum makespan is too long, the programmer has three courses of action.

1. To buy a faster processor for handling the critical path.
2. To reduce the sum of the labels of the critical path tasks by finding a more efficient algorithm to implement the computation.
3. To change the task graph of the program by finding more parallelism, ie. by dividing a task on the critical path into a number of independent parallel tasks, the sum of whose computation might well be greater than the computation associated with the whole task[3].

In general the sum of the node weights along the critical path will underestimate the makespan. This can happen for a variety of reasons, none of which is identified by the simple analysis of the problem by identification of the critical path. First, the critical path processor might be idling waiting for an input from a task residing on another processor. (This does not contradict our

[3]Note that if this is the case then this course of action reduces makespan by increasing total execution time.

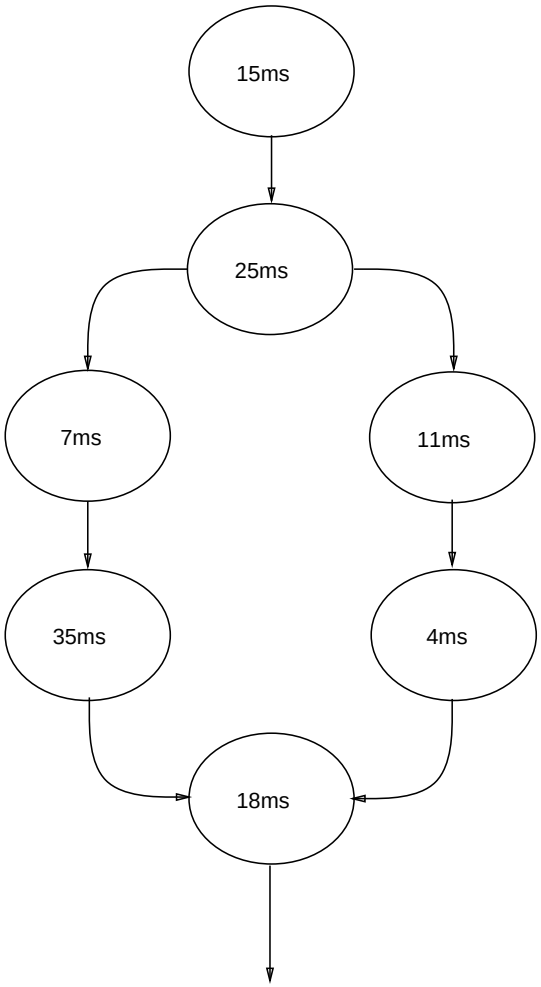

Figure 1: The Weighted Dag

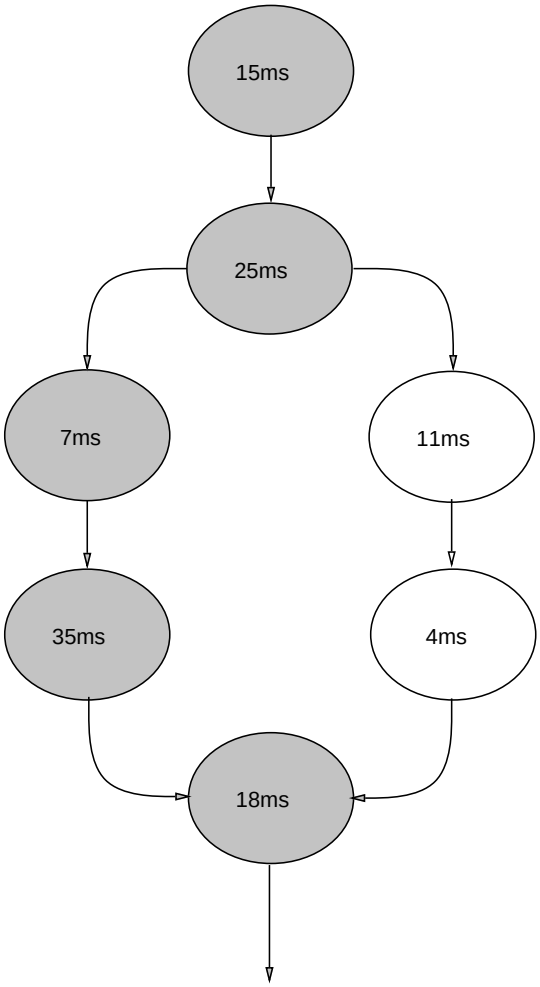

Figure 2: Dag with Shaded Critical Path

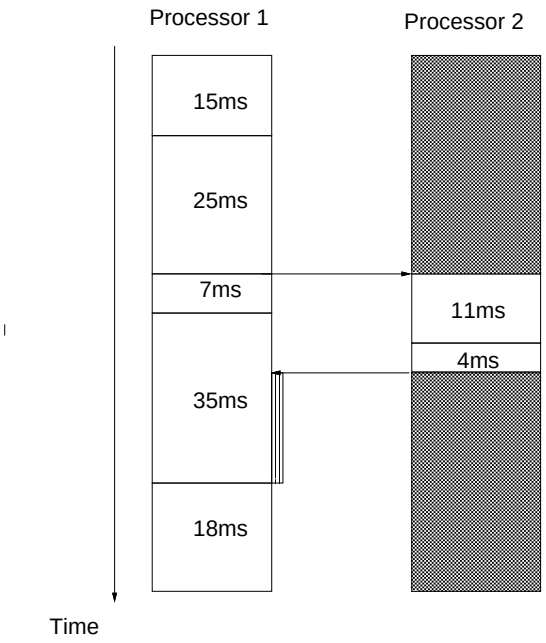

Figure 3: A Schedule for the Dag

definition of the critical path, since the delay may be caused by, for example, message latency in interprocessor communication.) Second, the critical path processor may be constantly busy, but may have to spend some of its time on communications-related computations, rather than on processing the critical path tasks.

The above problems lead us to consider the other labels that the program designer has been asked to supply to the mapping algorithm: the estimates of the volume of communication between tasks. We shall ignore, for the purposes of this section, the extra *computation* associated with communication events, and concentrate on the delay that may be introduced between the point in time at which the results of a task are known at the processor on which they were computed, and the point in time at which another processor becomes aware of them, perhaps making it possible to commence a task which has been mapped to it and which requires the result. This problem is illustrated in Figure 4, where the communication between processors can be seen to incur a delay which causes the processor to which the critical task has been mapped to be idle waiting for a message to arrive. Indeed the execution time for the schedule is more than the sum of the execution times of all tasks.

If we use a more sophisticated mapping algorithm which takes into account communication delays, we can still find a critical path. Informally, such an algorithm will tend to assign a pair of tasks to the same processor if one task produces a large volume of data that the other requires as input. The volume of communication can be used, along with some property of the interprocessor communications system, to calculate the delay associated with the message transfer. This is, however, dependent upon assuming that all the results of the task will be communicated at the tasks termination.

1.3 The Structure of the Review

The rest of this review is structured in the following way:

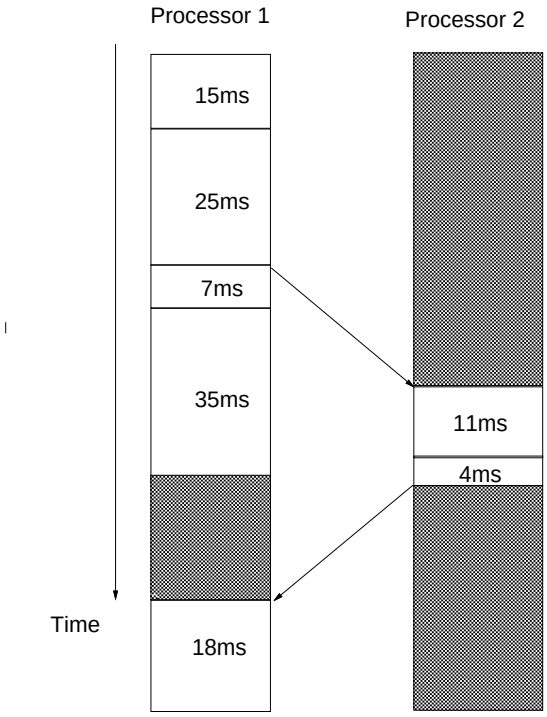

Figure 4: The Effect of Delays on the Schedule

First (Section 2) we give an overview of the way in which models of the mapping problem are usually formulated. Next (Sections 3 and 4) we outline a framework for understanding models of multicomputer architectures which we shall use when discussing the simplifying assumptions of the models used in the papers that we review in later sections.

There follow four sections, each dealing with a class of models of the mapping problem. Section 5 considers the simplest models of parallel processing where tasks are independent and do not communicate. This has been the subject of previous reviews and so we outline only major results and the most recent work. Section 6 considers models of parallel processing where communication between tasks occurs, according to a set of precedence relations on the tasks, but communication is assumed to be instantaneous and cost-free. Again this has been subject to previous reviews. In Section 7 we move on to some recent work where tasks have a precedence relationship and communication *delays* are taken into consideration. This contrasts with Section 8 where communication between tasks, according to the precedence relation, is modelled as a *cost*. In each of these sections we relate the different models to the framework.

Section 9 of the review is concerned with how parameters of models may be derived. We conclude, in Section 10, by summarising the way in which results and heuristics may be of use to the multicomputer programmer, and by outlining promising areas of research.

2 An Introduction to Modelling the Mapping Problem

The mapping problem consists of allocating various elements of a computation, which we shall refer to as modules, to various components of a parallel computer, which we shall refer to as

processors[4]. In broad terms we can consider three aspects to the mapping problem which we require to model: the processors and their communications facilities, the modules and their communications patterns to be mapped, and the function which is used to determine the *cost* of a mapping.

2.1 Modelling Costs

In this paper we wish to equate costs with the expected makespan—or time to completion—of the program. Thus we consider the "mapping problem" to consist of finding a mapping which minimises the expected makespan of a given program on a given multicomputer. We discuss a number of papers which do not share our understanding of cost, and which attempt to find a mapping that minimises a different property of the computation. As a result, in the following subsections the word "cost" is used to represent quantities other than the expected makespan—in particular we describe the cost of a communication, something that may relate rather nebulously to any real overhead, and which is discussed in Section 8.

2.2 Modelling Processors

We shall refer to a set P of n processors. It is common (eg. Graham *et al.* [1979]) to consider three ways in which the processors making up a parallel computer can vary in processing speed. They may be *identical*, that is every processor processes all modules at the same speed as every other. They may be *uniform*, that is the time of any given processor to process any module is a constant integer multiple of a unit speed. Alternatively they may be *unrelated*, for example a processor p could be *faster* than processor q at computing module γ, but *slower* than q at computing some other module δ. Since we are considering the *multicomputer* we shall mainly be interested in *identical* processors. The more general case of heterogeneous multicomputers corresponds to models of *unrelated* processors.

In order to model communications facilities between processors we introduce a dependence of the *cost* of communication between modules upon the processors to which tasks are mapped.

There are a number of options:

- The cost of communication between modules is independent of the processors to which modules are mapped.
- The cost of communication between modules depends upon the processors to which they have been mapped but in a way which is not based upon any property of the parallel computer.
- The cost of communication between modules depends only upon whether or not they have been assigned to the same processor.
- Processors are considered to be connected in an undirected graph $Q = (P, L)$ in which the nodes are the processors and an edge, $\langle p, q \rangle$, exists in L if and only if processors p and q are physically directly connected to each other; the cost of a communication between modules mapped to a given pair of processors depends upon some properties of this graph.

[4]Some authors refer to them as *processing elements* since they may consist of more than just a simple processing unit.

2.3 Modelling Modules

We shall refer to a set Γ of l modules that make up the computation. A module is a unit of computation that is executed sequentially. Modules can be executed preemptively or non-preemptively: that is they may or may not be allowed to be suspended. Modules are often algorithmic units – perhaps functions in a functional decomposition. Alternatively in numerical applications they may correspond to the computation associated with divisions of a data space.

There are three basic types of models, two of which were briefly mentioned in Section 1.2. First there are models where no communication occurs between modules. Here we shall refer to modules as *tasks* or *processes* interchangeably. Where communication is allowed there are two different classes of computational models which we shall refer to as *task* based models and *process* based models. Task-based models consist of tasks arranged in *directed* acyclic graphs where an arc between a pair of tasks corresponds to both a precedence relationship and an associated communication event. Process-based models consist of processes arranged in undirected graphs where an arc corresponds to a volume of communication between tasks. The directed graph models tend to be used by researchers interested in scheduling problems whereas the undirected graph models are used by those interested in modelling communications costs.

2.4 Mapping *vs* Partitioning

The correspondence between task-based and process-based models may be illustrated by considering the problem with using a task-based formulation in the context of programming multicomputers. We can imagine searching through a program, expressed in a language such as FORTRAN or c for tasks which can be identified as independent. The natural level of modularity for finding such tasks might be the function or subroutine level. This will be an appropriate level if functions are constrained to access external variables only when passed as parameters. Otherwise it will be necessary to drop to a lower level of modularity such as the compiler's abstract machine instruction level.

It is clearly rather fanciful to expect a graph of abstract machine instructions to be of a fixed topology since all that is required to make the topology variable is a non-predeterminable loop termination condition. For this and other reasons, the mapping problem is often viewed (for example by Efe [1982], Baxter and Patel [1989] and Sarkar [1989]) as the second step of a two phase process for allocating computation to processors. The first phase, *partitioning*, consists of merging tasks to *grains*, and then *mapping* consists of allocating grains to processors. Since a grain corresponds to a larger volume of computation, for example to the computation associated with a subspace of a data space, it is more appropriate to consider a static grain graph than a static task graph, but it is now less appropriate to make the modelling assumptions that grains only communicate on termination, and to require that all their constituent tasks be ready to compute before any of them are started—not least because certain partitions would then have cyclic dependencies.

In fact the distinction between partitioning and mapping is rather arbitrary, and this touches on a central issue in the mapping problem. Sometimes it is useful to consider computation as a set of atomic tasks, communicating on termination, sometimes as a set of processes sending and receiving a number of communications during their computation. Neither model can adequately capture all of the aspects of the mapping problem.

3 A Framework for Discussing Models of a Multicomputer

In this section we present a framework for discussing the activity of the processors of a multicomputer. We consider the case of a calculation – that is a set of algorithms and a set of data upon which the algorithms are to be applied – which is partitioned into a set of modules which are executed on a set P of n *identical*[5] processors. The framework bears some resemblance to the models of parallel computers described by Fox *et al.* [1988], or by Reed and Fujimoto [1987]. It differs in that it considers the state of the processor rather than the time for communication or calculation events to complete.

3.1 The Basic Four States

It is assumed that at all times during a given execution of a parallel program any processor $p \in P$ can be uniquely identified as being in one of four states

- Performing the computation required by the calculation
- Performing computation associated with message transfer
- Performing computation associated with housekeeping operations
- Idle

We shall refer to the time that a processor p spends in these states as $T_{Calc}(p)$, $T_{Comm}(p)$, $T_{House}(p)$ and $T_{Idle}(p)$ respectively. We also define:

$$T_{Calc} = \sum_{p \in P} T_{Calc}(p)$$

and similarly for all other values which are subscripts in T.

3.2 Time Spent Calculating

We can define T_{Calc} to be a property of P and the calculation being performed, and completely independent of the partitioning of the calculation into modules and the mapping of modules to processors. Where the set of modules includes some re-calculation this will appear as a component of T_{House}.

3.3 Time Spent Communicating

We can subdivide the term $T_{Comm}(p)$ as follows

$$T_{Comm}(p) = T_{Init}(p) + T_{Term}(p) + T_{Route}(p)$$

Where $T_{Init}(p)$, $T_{Term}(p)$ and $T_{Route}(p)$ are the amounts of time that a processor p spends performing the computation associated with message initiation, termination and through-routing respectively. Where processor p sends a message to processor q, the processing associated with communication contributes to $T_{Init}(p)$ and $T_{Term}(q)$, and if, for example, through-routing of the message causes costs to be incurred at processors other than p and q then it appears in T_{Route}

[5] Recall our definition in Section 2.2.

on those processors. It should be noted that these quantities have nothing to do with the time that a message may take to arrive[6]

Two of the message passing overheads, T_{Term} and T_{Init}, are defined to be a property of the decomposition of the algorithm into modules, and independent of the mapping of modules. They include all the computation which results from the partitioned address space: determining whether a message transfer is required; generating a packet; function call overhead associated with transfer and receipt of the packet and with generating any associated protocol; copying into and out of local address space; and unpacketing. As far as these two values are concerned, it is assumed that the computation associated with sending a message is independent of the localisation of the pair of modules. If the computation associated with sending a message to a module on the same processor is less than that for sending it outwith the processor, then this will show up as a negative interference cost (see below).

T_{Route}, the computation associated with intermediate node transfer of messages in a multicomputer, is seen only on such machines as first generation hypercubes and Transputer based multicomputers running message-passing systems. It is envisaged that this overhead will disappear as the computation is taken over by dedicated hardware. This is not to say that the influence of the underlying processor network disappears, since it continues to show up in T_{House} and T_{Idle}.

3.4 Time Spent Housekeeping

$T_{House}(p)$ is partitioned into three overheads according to

$$T_{House}(p) = T_{Sched}(p) + T_{Inter}(p) + T_{Recalc}(p)$$

$T_{Sched}(p)$ contains the local overheads of local dynamic scheduling of computation where processor p is assigned more than one module. It thus contains the cost of time-slicing between modules ie. the context switch, and of any algorithm being used to determine context-switching[7]. Unfortunately $T_{Sched}(p)$ is not simply a function of the set of modules that has been assigned to a given processor since it may be assumed that local scheduling will be dependent upon the timing of communication events, and this is dependent upon the global mapping and scheduling of modules.

$T_{Inter}(p)$ is another term which is dependent upon the global mapping of modules. It contains all the overheads which would not be there had processor p been assigned a single module, and which are not related to module scheduling. An example would be the buffering of arriving messages for a module while another module is being processed, given that, had the first module not been scheduled, the message would be passed to the second module without buffering.

$T_{Recalc}(p)$ is the overhead associated with recomputation of parts of the calculation on processor p (which may be performed in an attempt to minimise makespan). The computation associated with the calculation may appear only once in T_{Calc}. Recomputation, either of a whole module or of part of a module, must appear in T_{Recalc}. If calulation is performed more than once it is useful to consider the first computation to be *calculation*, the others *recalculation*. If it is initiated simultaneously on more than one processor then some arbitrary assignment of the computation events to T_{Calc} and T_{Recalc} must be made.

[6] The above definition applies for multicast messages if they are considered to be multiple message transfers.

[7] If, for example, receipt of communication causes a context switch to allow associated computation then this appears as a communication cost not a housekeeping cost.

3.5 Time Spent Idle

$T_{Idle}(p)$, the time processor p spends idle, is also partitioned:

$$T_{Idle}(p) = T_{Wait}(p) + T_{Fin}(p)$$

T_{Fin} is the time processors spend idle, having completed all their modules. It can be thought of as a place-filler. There is no cost associated with the extension of processing into T_{Fin}.

$T_{Wait}(p)$, the time that processor p spends idle with none of its modules executing, before it has performed all the computation associated with its modules, is a property of the global mapping of modules, and can be regarded as the overhead associated with load imbalance. Again there is no cost directly associated with extension of processing into $T_{Wait}(p)$ and so, for example, $T_{Recalc}(p)$ can be extended at the expense of $T_{Wait}(p)$, corresponding to re-computation of values so as to minimise overall makespan. It is often useful for $T_{Wait}(p)$ to be considered to be the sum of two components:

$$T_{Wait}(p) = T_{Wait_S}(p) + T_{Wait_D}(p)$$

Where $T_{Wait_S}(p)$ is the time processor p spends waiting for messages before they are sent, and $T_{Wait_D}(p)$ is the time that processor p spends waiting for messages that are in transit. If p's next scheduled task is awaiting more than one message from tasks on other processors then the time clocks up against $T_{Wait_D}(p)$ only if *all* outstanding messages are in transit. Although this explanation assumes an asynchronous or datagram service (see Tanenbaum [1989] for an introduction to this area) of message passing between modules, there is a corresponding explanation for synchronous transfers since, in all the models we deal with in later sections we can assume that no processor is ever waiting for another processor to read a message.

3.6 Makespan in Terms of the Model

Having developed our model, and stated that at any given time the processor can be in one and only one of the aforementioned states, for any given execution of a parallel program we can define $T(p)$, the time a processor spends processing, as:

$$T(p) = (T_{Comm}(p) + T_{Calc}(p) + T_{House}(p) + T_{Wait}(p))$$

Note that T_{Fin} is not included in the above, which allows us to define M the makespan of the program as

$$M = \max_{p \in P} T(p).$$

4 The Influence of the Communications Network

The properties of the interprocessor communications network have only briefly been alluded to in the above analysis, since they affect only indirectly the makespan of the program. Apart from the contribution to T through T_{Route}, it is only possible for the network to extend the makespan of the program if it causes an extension of T_{Wait_D} in the critical path. Unfortunately the times of arrival of messages may change the critical path. As a general rule, it is not necessary to expedite a message simply so that it can sit in a buffer at destination, whereas if a message is being awaited by an idle processor it should be sent as speedily as possible. This said, the way in which the network and the application interact to determine T_{Wait_D} depends critically upon the way in which tasks send and receive messages.

The interprocessor network is often described as having quiet and busy network performance characteristics. The former are often well understood, the latter are more complex.

4.1 Quiet Network Performance

Recall our definition of a processor graph $Q = (P, L)$ in Section 1. Clarke [1990] (see also Wallace [1991]) has shown that the performance of a number of interprocessor networks can be characterised in terms of a simple model. The time to completion of a message transfer, τ is given by

$$\tau = a + cw + bv + dvw$$

where v is the number of edges in Q being traversed, w is the number of words being transferred in the message, a, b, c and d are hardware specific constants constants.

In the case of packet switched networks, we can identify that the quantity $a + cw$, comprising the components of message transfer time which are independent of the number of edges being traversed, is associated with T_{Init} and T_{Term}, the processor overheads of sending and receiving the message. We can also identify bv, a fixed overhead associated with every step the message takes, as being partially attributable to T_{Route}. Finally, dvw is simply a property of the communications network. See Seitz [1990] and Dally [1990] for more discussion of message latencies in packet switched and circuit switched networks.

It should be stressed that there is no strong causal link between the components of T_{Comm} and the components of τ described above. The latter are the combination of pure network properties and a subset of what shows up in the former. For example the components of T_{Comm} which are associated with packeting and unpacketing of messages do not appear in τ.

4.2 Busy Network Performance

It is useful to consider the above quiet network analysis as a lower bound on the value of τ for busy networks. Busy network performance may be considered as being governed by contention for network resources. As in a conventional computer network, performance may be governed by one or more of switching mode (eg. packet switching, circuit switching or wormholing), routing strategies, queueing disciplines, the availability of queueing resource, flow control protocols and error correction protocols.

It is important for an analysis of communicatons performance in a busy network to be performed, especially for those programs which are close to saturating network bandwidth, either locally or globally, but detailed analysis of application specific performance of multicomputer routing systems is rarely made and contention is rarely analysed in models of parallel processing except in the context of PRAM emulation, which we have excluded from our discussion. Kruskal and Snir [1989] give an analysis of processor interconnection networks in terms of utilisation, but their analysis is framed form the viewpoint of processor network design rather than mapping to a fixed network.

5 No Task Precedence

We can identify a number of papers which consider the mapping of *independent* tasks across processors. The time that a processor spends computing its tasks is independent of the order in which it performs them, and of the mapping that has been made to other tasks.

Model 1: No Precedence

An instance A of the model is a 3-Tuple (P, Γ, f_c), where P is a set of n processors, Γ is a set of l tasks and $f_c : \Gamma \rightarrow Z_0^+$ is a function such that $f_c(\gamma)$ returns the time taken to compute task γ.

Let F_A denote the set of all surjective mappings from Γ to the collection of singleton subsets of P. This may be thought of as the set of possible task mapping functions. For each $g_m \in F_A$ we let $h_m : P \rightarrow 2^\Gamma$ denote the function which returns the set of tasks mapped to a given processor by mapping function g_m.

We define M_m, the makespan associated with the mapping function g_m to be:

$$M_m = \max_{p \in P}\left(\sum_{\gamma \in h_m(p)} f_c(\gamma)\right)$$

Model 1 is extremely limited. When an instance of the model is mapped with a function g_m, we can consider the times processors spend in the states defined in our framework as shown in Table 1.

Clearly $T_{Comm} = 0$, since there is no communication between tasks. Furthermore, $T_{Wait} = 0$; there is no waiting since a processor can simply start another task as soon as it has completed a previous one. We have no housekeeping costs: $T_{Recalc} = 0$, since there is no recomputation; $T_{Sched} = 0$ since scheduling is non-preemptive or instantaneous; $T_{Inter} = 0$ since tasks are assumed not to interfere with each other. $T_{Calc}(p)$ is merely the sum of the execution times of the tasks allocated to processor p, and the time a processor spends having finished its work appears as $T_{Fin}(p)$.

The model corresponds to the execution of independent programs, say for example by a parallel batch server, in a way that multicomputer programmers often refer to as *embarassingly parallel* (eg. Fox and Furmanski [1988]) or *event parallel* (eg. Pritchard *et al.* [1987]).

This model, and Model 6 outlined below, correspond to the formulations that are the basis of scheduling theory, which developed from the 1950's onwards and still is of current interest (eg. Ramamithram *et al.* [1990]). Although they underly much of what is discussed in the later sections of this review, it is not our purpose to discuss the results in detail, merely to put them in the context of multicomputer programming. As a result we refer interested readers to a number of previous reviews and books (Graham *et al.* [1979], Conway *et al.* [1967], Coffman Jr. [1976] (including the chapter by Sethi [1976]), Krishnamurthy [1990], Gonzalez [1977]), and summarise a number of relevant results.

We can pose the following decision problem:

$T_{Calc}(p)$		$\sum_{\gamma \in h_m(p)} f_c(\gamma)$
T_{Comm}		0
T_{House}		0
T_{Idle}	T_{Wait}	0
	$T_{Fin}(p)$	$\max_{q \in P} T_{Calc}(q) - T_{Calc}(p)$

Table 1: Model 1 in Terms of our Framework

Decision Problem 1 *Given an instance A of the above model and a positive integer k does there exist a function $g_m \in F_A$ such that $M_m < k$?*

Decision Problem 1 is NP-complete (Bruno *et al.* [1974]) in the case of two or more identical processors. It is related to the bin packing problem (See Coffman Jr. *et al.* [1984] and Garey and Johnson [1982]). As a result, it is natural to consider polynomial algorithms which, although they are not guaranteed to find the best solution, are guaranteed to find solutions which are close to the optimum. Such algorithms are known as approximation algorithms, or in the case where their solutions are guaranteed to be at most $1 + \epsilon$ times the optimal solution, as ϵ-approximation algorithms. An early partial review of this area can be found ion Garey *et al.* [1977].

Graham's Longest Processing Time (LPT) algorithm[Graham, 1969] guarantees to find M such that

$$M \leq (4/3 - 1/3n)M_{opt}$$

where M_{opt} is the optimum makespan. Coffmann *et al.* [1978] give an algorithm based on techniques from bin packing and improve this to $1.22M_{opt}$. Sahni [1976] produced a family of approximation algorithms for any guaranteed performance ϵ, whose running time was polynomial in l but exponential in n. More recently, Hochbaum and Shmoys [1988a, 1988b] have developed a family of ϵ-approximation algorithms which is polynomial in both l and n. Nevertheless the problem is still the focus for algorithm development, for example, Hellerstrom and Kanal [1990] consider an interesting approach to the problem, embedding it in a mean-field thermodynamic neural network, similar to that used by Hopfield and Tank for the travelling salesperson problem [1985].

As an extension to Graham's work, Coffman *et al.* [1978] consider the expected makespan for LPT scheduling under the assumption that tasks' execution times are independent, identically-distributed, random variables. They show that LPT makespans converge stochastically to optimal makespans for a range of distributions of task execution times. It is interesting to compare this approach to an analysis of the common multicomputer programming technique of scattered decomposition by Nicol and Saltz [1990], which considers the effect of correlations between tasks (associated with processing a dataspace), on the performance of a mapping.

Other authors (eg. Kafura and Shen [1977] and Garey and Johnson [1975]) have extended Model 1 to allow constraints upon allocations of tasks so that they compete for resources, thereby modelling the situation in multicomputers which do not allow virtual memory and have localised software interfaces and special purpose hardware.

Another generalisation of the model, by Błażewicz *et al.* [1984,1986] allows tasks to require more than one processor and corresponds closely to the subcube allocation problem for hypercubes studied by Chen and Lai [1988] and Chen and Shin [1987]. Błażewicz *et al.* [1986] show that the problem is NP-complete if tasks can require arbitrary numbers of processors, but give linear time complexity algorithms for an exact solution in the case of each task either requiring one or k processors. Du and Leung [1989] show that the problem is *strongly* NP-complete only if the number of processors on which the task graph is to be scheduled is greater than or equal to five. The fact that tasks require many processors task can be generalised to a requirement for arbitrary sets of resources, and so made to model access to the operating stystem. See Zhao [1987] for an approach to this.

The above results for the non-preemptive scheduling of tasks are in contrast to a more positive set of results for preemptive scheduling. McNaughton [1959] produced a simple polynomial time algorithm and Martel [1988] shows that this version of the problem is in the complexity

class NC defined by Pippenger [1979]. In the case of preemptively scheduled tasks requiring more than one processor, Du and Leung [1989] show that the problem is ordinarily NP-hard if processes require either one or k processors, and strongly NP-hard if they are allowed to require an arbitrary number of processors.

6 Tasks with Precedence

Multicomputer programs with inter-task communications are better modelled by an alternative formulation. Below we describe a model of non-preemptive scheduling where tasks show dependencies, and the dependency is satisfied at the termination of the precedent task. In terms of our multicomputer this corresponds to tasks performing and completing all their communication instantaneously at the point in time at which they finish executing.

Model 2: Precedence With No Cost

An instance A of the model is 3-tuple (P, Λ, f_c), where P is a set of n processors; $\Lambda = (\Gamma, \Delta)$ is a directed acyclic graph where Γ is a set of l tasks and Δ represents a partial order on the tasks; $f_c : \Gamma \rightarrow Z_0^+$ is a function such that $f_c(\gamma)$ returns the time taken to compute task γ.

Let F_A denote the set of all surjective mappings from Γ to the collection of singleton subsets of P. This may be thought of as the set of possible task mapping functions. For each $g_m \in F_A$ we let $h_m : P \rightarrow 2^\Gamma$ denote the function which returns the set of tasks mapped to a given processor by mapping function g_m.

For each $g_m \in F_A$ let S_m denote the set of all functions with domain Γ and range Z_0^+ such that for any $s \in S_m$

- if $\langle \gamma, \delta \rangle \in \Delta$ then $s(\gamma) + f_c(\gamma) \leq s(\delta)$
- $\forall p \in P, \forall \gamma, \delta \in h_m(p)$ if $\gamma \neq \delta$ and $s(\delta) \geq s(\gamma)$ then $s(\delta) \geq s(\gamma) + f_c(\gamma)$

S_m is the set of valid schedules for a given mapping g_m.

Now the makespan of a schedule s is given simply by

$$M_s = \max_{\gamma \in \Gamma}(s(\gamma) + f_c(\gamma))$$

Given an instance A of Model 2, a mapping $g_m \in F_A$ and a schedule $s \in S_m$ we can consider the activity of processors according to our framework as shown in Table 2.

$T_{Calc}(p)$, the time that a processor p spends computing, is again just the sum of the computation time of the tasks assigned to it. We assume no communications costs so $T_{Comm} = 0$. Again there is assumed to be no recomputation, no preemption costs, and no interference costs so $T_{House} = 0$. Since we assume there is no delay associated with communciation, processors are never idle waiting for messages in transit, ie. $T_{Wait_D} = 0$,. They can, however, be idle waiting for a precedence relation to be satisfied, and this contributes to T_{Wait_S}. Finally, the time they spend idle after all their tasks have been computed shows up as T_{Fin}.

Complexity results for this model were presented in 1975 by Ullman [1975].

[7] Martin and Estrin [1967] give methods for transforming directed cyclic graph based models of computation to directed acyclic graph based models.

$T_{Calc}(p)$		$\sum_{\gamma \in h_m(p)} f_c(\gamma)$
T_{Comm}		0
T_{House}		0
T_{Idle}	$T_{Wait_S}(p)$	$\max_{\gamma \in h_m(p)}(s(\gamma) + f_c(\gamma)) - T_{Calc}(p)$
	T_{Wait_D}	0
	$T_{Fin}(p)$	$\max_{\gamma \in \Gamma}(s(\gamma) + f_c(\gamma)) - \max_{\gamma \in h_m(p)}(s(\gamma) + f_c(\gamma))$

Table 2: Model 2 in Terms of our Framework

Decision Problem 2 *Given an instance A of Model 2 and an integer k, does there exist a mapping function $g_m \in F_A$ such that there exists a schedule $s \in S_m$ such that $M_s < k$?*

Decision Problem 2 is *NP*-complete for general n, even if the range of f_c is $\{1\}$, or in the case of $n = 2$ if the range of f_c is $\{1, 2\}$. Garey and Johnson [1977] show that the problem is NP-complete even if $n = 2$ and the range of f_c is $\{1\}$ if for all $\gamma \in \Gamma$, $s(\gamma)$ is required to be earlier than some deadline t_γ

More recently, Vazirani and Vazirani [1989] have shown that if $n = 2$, Decision Problem 2 is in Random NC, and a stronger result by Hembold and Mayr [1987] shows that it is in NC. Due to differing delays in the reviewing process, this stronger result was published first. Showing the two-processor scheduling problem is in NC implies that it can be solved on a concurrent read concurrent write PRAM with a number of processors polynomial in the size of the problem in time which is a polylog of the problem size. In the case of Vazirani and Vazirani the component of their algorithm with highest order parallel complexity requires $O(log^2 l)$ on l^3 processors. Remember it is only *scheduling* two.

Du and Leung [1989] give complexity results for the scheduling of precedence constrained tasks which require more than one processor. This is NP-complete even if the precedence constraints consist of a set of chains, and there are only two processors.

In a similar way to the serial complexity results there are results for the parallel complexity of special classes of task graphs. The problem is in NC if the precedence constraints are represented by a collection of outtrees (Dolev *et al.* [1986]). However, as a complementary result shown in the same paper, it is unlikely that it is in NC if the precedence constraint is either a collection of outtrees and intrees or if the number of processors varies with time.

Algorithms for solving or approximately solving the mapping problem as formulated above have been appearing in the literature with remarkable frequency. In the cases where the problem is not NP-complete there are a number of exact polynomial time algorithms. Otherwise, heuristic approaches have often been used. This version of the mapping problem has been the subject of previous reviews (see Chen and Liu [1975] for a discussion of various similar heuristic approaches). Indeed it is often considered in the same reviews as Model 1, and as a result we only describe a selection of the results and heuristics.

Hu [1961] showed that where the task graph is a tree, and the range of f_c is $\{1\}$, a schedule based upon sequential processing of layers of the tree is optimal. Kaufmann [1974] extended Hu's algorithm to the case of non-unit length tasks, and showed bounds on its performance which allowed him to consider it "almost optimal".

This work contrasts with Graham's List scheduling approaches [Graham 1966, Graham 1969]. In 1972 Coffman Jr. and Graham [1972] showed an algorithm, for any task dag, which generates optimal schedules for $n = 2$ and f_c with range $\{1\}$. Their algorithm may be generalised to arbitrary n. Lam and Sethi [1977] showed that if M_o is the optimal makespan of the

graph, then Coffman and Graham's algorithm will generate a schedule of makespan M, where $M/M_o \leq 2 - 2/n$. Gabow [1988] showed a linear time algorithm for scheduling on two *uniform* processors (recall our definition in Section 2.2), again with unit length tasks, which generates optimal results with certain fixed ratios of processor speeds and nearly optimal results otherwise. Cho and Sahni [1980] give bounds for list schedules on general uniform processor systems. Cole and Vishkin [1988] show logarithmic time parallel implementations of list scheduling on an EREW PRAM. There are also a number of results for preemptive scheduling of precedence constrained jobs (eg. Muntz and Coffman Jr. [1969] for 2-processor systems), and in this context we refer the reader to the reviews outlined in Section 5 and to Lawler [1982].

Relatively few authors consider applying their algorithms to real problems. An exception is Kasahara and Narita [1984] who give a branch and bound based approach which they demonstrate on a number of task graphs. In this context it is interesting to compare Martin and Estrin [1967] with Shirazi *et al.* [1990]. The former use a heuristic optimisation method to refine an initial heuristic mapping of tasks to processors. They show simulation results for a number of program-derived task graphs in a model which allows probabilistic branching. The latter analyse the worst case performance of three different heuristic based approaches to mapping task dags, and show simulation results for random task graphs, and for other program-derived task graphs. In the intervening twenty three years, no consensus has evolved over a meaningful way of comparing heuristics, either for their performance on a given task graph, or, as Martin and Estrin point out, for their robustness to variations in the topology and labelling of task graphs. See Adam *et al.* [1974] for some discussion of this latter issue.

7 Task Precedence and Communication Delays

The model described in Section 6 captures the essence of interprocessor communication in terms of the implied precedence, but fails to capture any of the overheads associated with message transfer. This and the following sections explain extensions to the previous model which attempt to characterise the overheads of communication in different ways. This section describes an approach where messages are simply delayed. The area has been the subject of a recent short review (Veltman *et al.* [1990]).

In the simplest form of the delay based models there is assumed to be a uniform communication delay between the result of a computation being generated and it being known to all processors. In the case described by Papadimitriou and Yannakakis [1990] tasks may be assigned to more than one processor, that is they may be re-computed if it is more efficient to do so than to wait for the results of the computation to arrive from elsewhere. Again the model refers to non-preemptive scheduling.

> **Model 3: Precedence with Communication Delay**
>
> We define an instance A of the model to be a 4-tuple, (P, Λ, f_c, τ) where P is a set of n processors; $\Lambda = (\Gamma, \Delta)$ is a directed acyclic graph such that Γ is a set of l tasks and Δ represents a partial order on the tasks; $f_c : \Gamma \rightarrow Z_0^+$ is a function returning the time for execution of a task; and τ is an integer communication delay.
>
> Let F_A denote the set of all mapping functions, g_m, such that
>
> $$g_m : \Gamma \rightarrow 2^P$$

T_{Calc}		$\sum_{\gamma \in \Gamma} f_c(\gamma)$
T_{Comm}		0
T_{House}	T_{Recalc}	$\sum_{p \in P} \sum_{\gamma \in h_m(p)} f_c(\gamma) - T_{Calc}$
	T_{Sched}	0
	T_{Inter}	0
T_{Idle}	$T_{Wait}(p)$	$\max_{\gamma \in h_m(p)}(s(\gamma,p) + f_c(\gamma)) - \sum_{\gamma \in h_m(p)} f_c(\gamma)$
	$T_{Fin}(p)$	$M_s - \max_{\gamma \in h_m(p)}(s(\gamma,p) + f_c(\gamma))$

Table 3: Model 3 in Terms of our Framework

Function $g_m \in F_A$ returns the set of processors on which a task is executed in the mapping defined by g_m. Task replication is permitted in this model, but by defining that g_m returns a *set* of processors we have also defined that the mapping ensures that no task is executed more than once on the same processor. (Under the terms of the model this can only ever increase makespan.)

For each g_m we define a corresponding function:

$$h_m : P \rightarrow 2^{\Gamma}$$

which, given a processor p returns the set of tasks $\{\gamma \in \Gamma | g_m(\gamma) = p\}$.

For each g_m we define the set S_m of allowable schedules s such that

$$s : \Gamma \times P \rightarrow Z_0^+$$

where, for any given $s \in S_m$, if $\gamma \notin h_m(p)$, then $s(\gamma, p)$ is undefined, otherwise $s(\gamma, p)$ is the time at which task γ is executed on processor p. The value of $s(\gamma, p)$ is constrained such that.

- $\forall \langle \gamma, \delta \rangle \in \Delta \, \forall q \in g_m(\delta)$
  - if $q \in g_m(\gamma)$ then $s(\gamma, q) + f_c(\gamma) \leq s(\delta, q)$
  - otherwise $\exists p \in g_m(\gamma)$ such that $s(\gamma, p) + f_c(\gamma) + \tau \leq s(\delta, q)$
- $\forall p \in P \, \forall \gamma, \delta \in h_m(p)$ if $\gamma \neq \delta$ and $s(\delta, p) \geq s(\gamma, p)$ then $s(\delta, p) \geq s(\gamma, p) + f_c(\gamma)$

Given a mapping function $g_m \in F_A$, for any given $s \in S_m$,

$$M_s = \max_{p \in P} \max_{\gamma \in h_m(p)} (s(\gamma, p) + f_c(\gamma))$$

Again we can consider the times processors spend in the states referred to in our framework. For any given schedule s of any given mapping g_m of any given instance A of Model 3. This is shown in Table 3.

Here we are defining T_{Calc} to be the sum of the time it would take to execute each of the tasks once. Any other time spent executing tasks is assigned to T_{Recalc}. We define the wait time of each processor to be the time to it finishing its last task less the time it spends calculating or recalculating, and the time it spends finished to be the time between it finishing its last task

and the last processor finishing its last task. It is also possible to partition the time in $T_{Wait}(p)$ to $T_{Wait_D}(p)$ and $T_{Wait_S}(p)$ in the following way.

Let $x = |h_m(p)|$. We define the ordered set $o(p) = \langle \gamma_1, \gamma_2, \ldots, \gamma_x \rangle$ as the ordering of the tasks in $h_m(p)$ in the order in which they are executed. In other words, $o(p)$ satisfies the inequalities

$$s(\gamma_1, p) < s(\gamma_2, p) < ... < s(\gamma_{x-1}, p) < s(\gamma_x, p)$$

For $i = 1, 2, ..., x$, we define $Wait(\gamma_i, p)$ to be the idle time on processor p before γ_i can start to execute on p. For γ_1,

$$Wait(\gamma_1, p) = s(\gamma_1, p) - 1.$$

For $2 \leq i \leq x$,

$$Wait(\gamma_i, p) = s(\gamma_i, p) - [s(\gamma_{i-1}, p) + f_c(\gamma_{i-1})].$$

We can distinguish between idle time waiting for messages in transit and idle time waiting for messages that have not been sent when p becomes idle. These two quantities are denoted $Wait_D(\gamma_i, p)$ and $Wait_S(\gamma_i, p)$.

$$Wait_D(\gamma_i, p) = \min(Wait(\gamma_i, p), \tau)$$

and

$$Wait_S(\gamma_i, p) = Wait(\gamma_i, p) - Wait_D(\gamma_i, p).$$

Now we can define

$$T_{Wait_D}(p) = \sum_{\gamma \in h_m(p)} Wait_D(\gamma, p)$$

and

$$T_{Wait_S}(p) = \sum_{\gamma \in h_m(p)} Wait_S(\gamma, p).$$

Decision Problem 3 *Given an integer k and a 3-tuple $B = (\Lambda, f_c, \tau)$, where the range of f_c is restricted to $\{1\}$, does there exist an instance $A = (P, \Lambda, f_c, \tau)$ of Model 3 for which there is a mapping function $g_m \in F_A$, and an associated scheduling function $s \in S_m$ such that $M_s < k$?*

Papadimitriou and Yannakakis [1990] show Decision Problem 3 is NP-complete. Their proof implies it is NP-complete even if recomputation is forbidden. Rayward-Smith [1987a] shows the problem is NP complete for unit length tasks with $\tau = 1$. Rayward-Smith [1987b] shows the preemptive version of the problem is NP-complete for $\tau > 1$, and gives a polynomial algorithm for the preemptive version with $\tau = 1$. Chrétienne [1989] gives polynomial algorithms for tree-like precedence contraints without replication.

Following work deriving algorithms for special task graphs as described in Papadimitriou and Ullman [1987] which have a cost based communications model (see Section 8), Papadimitriou and Yannakakis show a polynomial-time approximation algorithm with worst case ratio 2 for the delay based problem with general task graphs, possible task replication and no preemption. This is based upon computing a function e on the depth of a task which is the time before which it cannot be computed. For each processor, their algorithm then computes the τ highest in e value ancestors of any task that it has been assigned and receives the rest from other processors. Another positive result is shown by Jung *et al.*[1989]. Although Decision Problem 3 is NP-complete, they show algorithms that are $O(n^{\tau+1})$. That is they are polynomial in the problem size once τ is fixed. A variable number of processors is used by these algorithms. In

the case of Jung *et al.* there must be as many processors as tasks. In the case of Papadimitriou and Yanakakis, the algorithm devises a schedule, and not a mapping and it is not clear how do derive a *processor efficient* mapping (ie. one that uses few processors) from the schedule. In both cases it may be possible to use Brent's scheduling principle [Brent 1974] when the number of physical processors available is less than the number of tasks that may be simultaneously processed.

Papadimitriou and Yanakakis give a generalisation of their algorithm to general f_c which allows τ to be a property of the task to which the communications arc is outgoing. One can rationalise this as tasks outputting an equal number of unit length messages to all their postcedents, and it may be that the work of Valiant and collaborators [Valiant and Brebner 1981], [Valiant 1982], Upfal [1984] and others provides a framework within which it is possible to consider the time for a message to be delivered to be independent of the other communications going on in the processor network.

Lee *et al.* [1988] and Hwang *et al.* [1989] deal with a similar model, but τ is dependent upon the precedence relation that is being satisfied and the processors to which the communicating tasks are mapped. Moreover, they deal with non-unit length tasks and a fixed number of processors and so the problem can be seen to be a direct extension of the problem, outlined in section 6, which is NP-complete (Ullman [1975]). Lee *et al.* [1988] show an algorithm which is similar in flavour to Graham's List scheduling algorithm [1969] which they refer to as Earliest Ready Task (ERT) where tasks are scheduled at a processor in the order at which they become ready to be computed. This time is clearly the time of arrival of the last message which is required to service the precedence relations of the task. The time of arrival is clearly dependent upon properties of the message itself and the properties of the communication network. They do not, however, model contention. Their algorithm is shown to satisfy

$$M \leq (2 - 1/l)M_{opt} + K$$

where M_{opt} is the optimal makespan, and K is a constant for a particular task graph. Although it is not necessary to compute K to perform the scheduling, its value is derived from the length of the longest chain in the task graph and they give a non-polynomial algorithm to compute it.

Baxter and Patel [1989] give a heuristic called LAST for scheduling graphs, however they appear to be unaware of other work on Model 3 (most of their references use models similar to Model 4), and give no comparative performance. The first such heuristic approach to scheduling precedence constrained graphs with delays appears to be Wiliams [1983].

El-Rewini and Lewis [1990] have extended the above model by analysing contention in the processor network. That is, τ is dependent upon the existence of other communications in the processor network, and communications claim communications resource for the duration of their message transfer. This model is built upon a graph based model of interprocessor communications similar to that outlined in Section 8. El-Rewini and Lewis present heuristics, but no analytical results for their effectiveness, and simulated performance results for a set of task graphs.

8 Cost Based Models

Section 7 has explained an approach to modelling communications overheads where there is a delay associated with the results of a module becoming known. This section describes another, historically older, approach whereby intermodule communications incur costs.

Feature	Distributed Systems	Multicomputer Systems
Work Profile	Multiple Job	Single Job
Processor Type	Homogenous	Heterogenous
Fault Tolerance	Important	Ignored
Optimisation	Maximise Throughput by Minimising System Resource Consumption	Minimise Makespan
Ratio of Message Latency to Instruction Cycle Time	Relatively High	Relatively Low
Individual Job Execution	Can be Required to be Sequential	Parallel

Table 4: Relevant Differences between models of distributed systems and models of multicomputer systems

In this section we review work on mapping that is of more relevance to distributed systems than it is to multicomputer systems. We include it in our review, firstly because there is a grey area between the two types of system and, secondly, because several papers can be viewed as attempts to adapt these research results to mapping problems for multicomputers. Unfortunately, there are significant differences in the abstract models of computation being used by researchers in the two fields; these are summarised in Table 4. We are not saying that all researchers make these assumptions, rather we are conveying an impression of the differences in emphasis between the two fields.

The basis of many of the models reviewed in this section is that there is a computation cost associated with each module and a communication cost associated with each inter-module message. The total cost of a mapping is the sum of all the computation costs and communication costs. The majority of these models deal with inhomogenous processing systems, where the computation cost of a task depends upon the processor upon which it is executed. Modules are assumed to be persistent processes, and if they are multiprocessed, it must be assumed that they can be preempted, and that the costs of preemption can be modelled. The models, however, lack a precedence constraint on processes, and thus the strategy by which multiprocessing is controlled is not part of the model.

We can formulate a model such as that used by Stone [1977].

Model 4: Communication Costs and Computation Costs

An instance A of the model is a 4-tuple (P, Λ, f_d, f_e). P is a set of n processors, $\Lambda = (\Gamma, \Delta)$ is an undirected graph, where Γ is a set of l processes, and Δ is a set of undirected edges corresponding to communication between processes; $f_d : \Gamma \times P \rightarrow Z_0^+$ is a function such that $f_d(\gamma, p)$ returns the time required to compute task γ on processor p; $f_e : \Delta \rightarrow Z_0^+$ is a function returning the cost associated with communication between processes if they are mapped to different processors.

Given A, we can consider the set F_A of functions which map from Γ onto P. This may be thought of as the set of possible task mapping functions. For each $g_m \in F_A$ we can define a corresponding function

$$h_m : P \rightarrow 2^{\Gamma}$$

which returns the set of tasks mapped to a given processor by mapping

function g_m.

Now the global *cost* of computation, U_m associated with mapping function g_m, is given by:

$$U_m = \sum_{\gamma \in \Gamma} f_d(\gamma, g_m(\gamma))$$

and the global cost of communication associated with g_m is:

$$V_m = \sum_{\{\gamma,\delta\} \in \Delta | g_m(\gamma) \neq g_m(\delta)} f_e(\langle \gamma, \delta \rangle)$$

And the total cost R_m of a mapping is given by:

$$R_m = U_m + V_m$$

For the model we can define the following decision problem:

Decision Problem 4 *Given an instance A of Model 4 and an integer k: does there exist a function g_m in F_A such that $R_m < k$?*

There are a number of results for this model which relate to two-processor systems. These stem from work by Stone [1977, 1978] based upon the use of network flow diagrams. Stone shows an optimal algorithm for model 4 with $n = 2$. An extension of Stone's work to three processors was performed by Stone himself [1977] and Bokhari [1981] cites an unpublished result of Gursky that the four or more processor versions are NP-complete.

A substantial nail in the coffin of this model came in Fernández-Baca's result [Fernández-Baca 1989] that Decision Problem 4 is NP-complete if all of the following restrictions hold:

- the range of f_d is $\{0\}$,
- the range of f_e is $\{1\}$,
- $n = 3$,
- Λ is both planar and biparite.

Furthermore, Fernández-Baca [1989] showed that there can exist no ϵ approximation algorithms for the problem constrained in the above fashion and no exact local search algorithm that takes polynomial time per iteration.

There were, however some positive aspects to Fernández-Baca's paper in that he presented an extension of some work by Bokhari [1981] on task graphs that were trees and by Towsley [1986] who applied a dynamic programming approach to the problem with an arbitrary number of processors but special types of task graphs. Bokhari had shown an algorithm that, if the task graph is a tree, is guaranteed to find an exact solution in $O(ln^2)$ time. Towsley had considered series-parallel graphs, and shown an algorithm with time complexity $O(ln^3)$. Fernández-Baca extended the results to other tree-like graphs. Rao *et al.* [1979] consider the case of Stone's original model but where one of the processors is constrained in memory, and show two techniques which can reduce the complexity of the problem in some cases but not in general. Gusfield [1983] solves the problem with a parametric computing technique, for the costs of mapping tasks to the two processors varying as a function of two independent parameters.

8.1 Applying Stone's model

We can rationalise Stone's model in terms of our framework as outlined in Table 5.

$T_{Calc}(p)$		$\sum_{\gamma \in h_m(p)} f_d(\gamma, p)$
T_{Comm}		0
T_{House}		0
T_{Idle}	T_{Wait_D}	V_m
	T_{Wait_S}	$(n-1)R_m$

Table 5: Model 4 in Terms of our Framework

The above rationalisation stems from the fact that Stone's work considers only the case of sequentially executing tasks. That is modules are executing on one of n processors, and the other processors are idle for the duration of its execution. The cost of communication between modules corresponds to a delay between a module terminating and another module starting. The *sum* of the costs only corresponds to the makespan if the costs are incurred sequentially.

There is another problem with using Stone's model in the context of multicomputers: in the case of multicomputers with identical or uniform (ie. not unrelated) processors, the minimal cost mapping will assign all modules to the same processor. (Interestingly Stone suggests making the assumption of uniform processors in his proposed extention to dynamic mapping).

8.2 Interference Between Processes

Given the problem of NP-completeness, a number of heuristic approaches to the problem of determining optimal mappings have been proposed. Many of these attempt to allow constraints upon the solution, for example memory resource constraints. Chu *et al.* [1980] consider integer programming approaches in the presence of constraints. Gylys and Edwards [1976] describe module clustering algorithms which satisfy constraints. Efe [1982] describes an approach where modules are mapped by a clustering algorithm, and then re-allocated to satisfy constraints. Ma *et al.* [1982] use a branch and bound technique to solve a model which includes constraints, including a *redundancy* constraint: certain modules must be allocated to more than one processor.

Other authors have used constraints and costs to attempt to encourage the potential of parallelism between processes in the resulting mapping. Lo [1988] considers an alternative to Model 4 where if γ is mapped to processor p, $f_c(\gamma, p)$ is dependent upon the elements of $h_m(p)$: that is there is a cost associated with interference between tasks. Houstis [1990] (in the context of real-time systems) adds an explicit parallel processing constraint to the model: if two processes *can* be executed in parallel then they *must* be executed on different processors – it is not clear what the justification for this is. Gaudiot *et al.* [1988] use a less extreme version of this constraint. Where two potentially parallel modules are assigned to overlapping sets of processors (they allow tasks to require many processors) a cost is incurred which is included in the cost of the mapping.

8.3 Processor Graphs

Chu *et al.* [1980], in what may be considered a partial review of the field, describe a version of the above where the cost of communication between tasks is dependent upon the processors to which they have been mapped. See also Chu [1969]. Sinclair [1987] applies a branch and bound algorithm to the problem with general task graphs and mapping dependent communications costs. He claims that it gives good results but can give no guarantees of its time complexity.

Houstis [1990] and Cvetanovic [1987] consider models of contention for communication resource, where the communications medium is considered to be a saturable bus, whose performance for any communication is determined by its rate of utilisation. Although the inclusion of contention makes the model more complete, this particular model of contention is not appropriate for multicomputers. More relevant to the multicomputer programmer, there is a set of models where the communication cost is dependent upon some property of the underlying processor network. The problem becomes that of mapping an undirected graph of processes into an undirected graph of processors so as to minimise the communication overhead. In the case of a multicomputer, the overhead is often defined as some mis-match between the links of the process graph and the links of the processor graph. The algorithms can either assume as many processors as processes [Bokhari 1981], or can assume multiprocessing [Berman and Snyder 1987]. Udiavar and Stiles [1990] use simulated annealing to solve a mapping problem in which communications costs[8] between processors are variable – dependent upon the distances between processors in a graph the topology of which is a parameter in the optimisation.

We can describe a model similar to that used by Bokhari [1987]

Model 5: Communication Costs Only

An instance A of the model consists of a 4-tuple (Q, Λ, f_s, c) where $Q = (P, L)$ is a graph of n processors, (the edges correspond to interprocessor links); $\Lambda = (\Gamma, \Delta)$ is a process graph where Γ is a set of n processes (ie. there are as many processes as there are processors) and an edge $\{\gamma, \delta\} \in \Delta$ implies γ communicates with δ; c is the time required to execute any $\gamma \in \Gamma$; $f_s : \Delta \rightarrow Z_0^+$ is a function such that $f_s(\{\gamma, \delta\})$ returns an integer corresponding to the amount of communication that γ performs with δ (bidirectionally) during its execution.

For A we define a communications cost function

$$f_A : P \times P \rightarrow Z_0^+$$

(where $f_A(p, q)$ returns the cost associated with sending a message between processors p and q) in the following way. A *route* in Q between p and q is a set of vertices

$$\{r_1,\ r_2,\ \ldots,\ r_v\} \subseteq P\ -\ \{p, q\}$$

such that

$$\{p, r_1\},\ \{r_1, r_2\},\ \ldots,\ \{r_{v-1}, r_v\},\ \{r_v, q\}\ \in\ L.$$

We define the *length* of this route between p and q to be v. Now we define $f_d(p, q)$ to be the length of the shortest route between p and q in Q.

For our instance A, we can consider the set F_A of functions which map from Γ onto a member of P. This may be thought of as the set of possible process mapping functions. For each $g_m \in F_A$ we can define a corresponding function

$$h_m : P \rightarrow \Gamma$$

which returns the process mapped to a given processor by mapping function g_m.

R_m, the cost of the mapping associated with any $g_m \in F_A$ is given by:

[8] It is not clear from the paper whether their model associates delays with messge transfers as in Model 3 or simply costs.

$$R_m = nc + \sum_{\{\gamma,\delta\}\in\Delta} (f_s(\{\gamma,\delta\}) \times f_A(g_m(\gamma), g_m(\delta)))$$

We define two decision problems for the above model.

Decision Problem 5 *Given an integer k and an instance A of Model 5 where the range of f_A is restricted to $\{0,1\}$ and $c = 0$, does there exist a function $g_m \in F_A$ such that $R_m \leq k$?*

Decision Problem 6 *Given integers k, j and a 3-tuple $B = (\Lambda, f_s, 0)$, does there exist a corresponding instance of the above model $A = (Q, \Lambda, f_s, 0)$, where Q is a graph with degree at most j and there exists a mapping $g_m \in F_A$ such that $R_m \leq k$?*

Decision Problem 5 corresponds to the question: can I allocate processes to processors in a given processor topology so as to minimise through-routing? Decision Problem 6 corresponds to the question: can I wire up my processors so that through-routing is minimised?

Bokhari [1981] points out that Decision Problem 5 is a notational variant of the graph isomorphism problem. The incorrect notion that graph isomorphism is NP-complete is endemic in literature on the mapping problem; see, for example, Krämer and Mühlenbein [1989] and Pountain [1989]. The complexity status of graph isomorphism is still an open problem (see Garey and Johnson [1979]).

Thanisch and Norman [1990] point out that in the case of processor graphs that are chains, and non-zero k, Decision Problem 5 is a notational variant of the NP-complete *simple optimal linear arrangement* problem [Garey *et al.* 1976]. They also show that Decision Problem 6 is NP-complete, although they require an unrestricted range of f_s. Pinter and Wolfstahl [1987] show that it is an NP-complete problem to determine the minimum number of edges that must be added to a linear graph of processors so as to allow a g_m such that $R_m = 0$.

There are some obvious extensions to the above model. First it is possible to consider contention in the processor network, assuming that messages are routed between processors according to some deterministic routing strategy. The overall cost of a set of message transfers is the time to completion which is the maximum of any time to completion allowing for contention. This approach is taken by Lee and Aggarwal [1987] and Berman and Snyder [1987]. The second extension is to consider the problem when there are more processes than processors. Berman and Snyder [1987] extend the analysis to consider multiprocessing, having contracted the process graph to have the same number of processors as the processor graph.

8.4 Introducing Precedence

It is instructive to consider Model 5 in terms of our framework. If we assume that all processes start waiting for communications at exactly the time they are sent, then the cost of the message, which is dependent upon interprocessor distance, makes sense as a communications latency, which that process must wait upon, and therefore shows up in T_{Idle} in our framework as outlined in Table 6.

This analysis does not, however, bear up to close scrutiny since the time at which a process comes into a waiting state will depend upon the other messages that it receives.

Another way of reconciling the cost of communication in terms of our framework is to consider it to be contributing to T_{Route}. Here, we can assume that processors are always potentially busy, that is they never wait for messages, and messages delay computation by taking up

$T_{Calc}(p)$		c
T_{Comm}		0
T_{House}		0
T_{Idle}	T_{Wait_D}	$R_m - nc$
	T_{Wait_S}	0

Table 6: An Attempt to Rationalise Model 5 in Terms of our Framework

the processors' time doing through-routing. Unfortunately, this too does not bear up to close scrutiny since the through-routing costs are only additive on a per-processor basis—we are interested in the bottleneck processor(s) not the total amount of through-routing. The function f_A does not apportion through-routing computation to a particular processor along the shortest route(s), and so the relationship between the through-routing costs and the time to completion of the program is undefinable.

In general, the cost functions in models 4 and 5, can be made more relevant by considering the maximum cost across processors rather than the total, but there are problems in apportioning costs to individual processors. Indurkhya *et al.* [1986], who considering randomly generated programs, simply add all communications costs to the maximum of the processors execution costs. See Nicol [1989] for the limitations of the results in this paper. Shen and Tsai [1985] consider a function which assigns communication costs to both the sending and receiving processor in a way that it is difficult to justify in the case of a multicomputer[9].

Bokhari [1988] describes polynomial time algorithms for solving the above mini-max problem in the case of chains of tasks and chains of processors with the constraint that communicating tasks must be mapped to adjacent processors, and also in various other constrained task formulations for host-satellite processor systems.

Shen and Tsai justify their claim by stating that they are dealing with a model where "little or no" precedence relations exist between modules. They vary the cost of sending a message according to its destination processor. A paper which attempts to address these issues head on is Chu and Lan [1987] where for each processor, the costs of computation $T_{Calc}(p)$, message receipt $T_{Term}(p)$ and message sends $T_{Init}(p)$ were identified, and the cost of a mapping was considered to be its makespan which was the maximum of sum of these values for each processor. Chu and Lan went on in the same paper to consider the effect of the precedence relation upon the makespan and concluded that, where two tasks are connected by a precedence relation, if the execution of the second module is much larger than that of the first module, then they should be mapped to the same processor, whereas if the second is much larger than the first, they should be mapped to different processors. They use this heuristic, and one which tends to group heavily communicating modules, to form grains which are mapped to the same processor, and then map the grains to processors by an exhaustive search which minimises makespan ignoring precedence. They make no claims as to the guarantee of effectiveness of the algorithm, but show simulation results for its use in an example task graph. By using a mini-max criterion rather than a sum to optimise a mapping, both Shen and Tsai [1987] and Chu and Lan [1985] allow the possibility of re-computation of tasks on processors so as to minimise the overall

[9]One can argue that the approach of setting communication costs to zero for communications on the same processor and k for non-local communications models the overhead associated with packeting and unpacketing messages, in that local communications do not need to invoke the message passing system and therefore T_{Init} and T_{Term} are zero. Alternatively we can consider the reduced costs of local mapping to be negative interference costs. However this approach is equivalent to merging tasks mapped to the same processor into grains, and is an effective re-partitioning of the computation. Partitioning is largely outwith the scope of this review, but discussed briefly in section 10.

computation.

Another approach is to move back from the dag to a probabilistic and possibly cyclic graph of module dependencies and branching probabilities. Queuing theory and markov decision theory can be applied to estimated time to execution of the program. This approach is taken by Kapelnikov *et al.* [1989] and Chou and Abraham [1982] and, for the special case of series-parallel task graphs, by Mak and Lundstrom [1990]. Chu and Lan [1987] and Chu *et al.* [1984] use a variation of this where the probabilities of branching are allowed to vary between intervals during the program's execution. This latter modelling approach may prove a useful alternative to dynamic allocation of modules in some programs.

9 Finding the parameters of the model

So far during this review a few obvious questions have been avoided. These relate to the problem of determining, for a given program, or more specifically for an execution of a given program on a given architecture, the corresponding instance of a given model. That is, the costs associated with computation of each task, the costs associated with communication between tasks, the precedence relations between tasks, and even the number of tasks. In order for an algorithm to map and/or schedule the computation it requires these values as input. They need to be derived from the program. In general it is difficult to derive them without actually performing the computation, and they need to be re-derived for any run of the program with diferent parameters.

9.1 Parallelisation of Sequential Code

As discussed in section 2.4, we can view the allocation of computation to processors as a two phase process consisting of *partitioning* followed by *mapping*. Sarkar [1989] considers these two phases to be preceded by something he refers to as an *identification of parallelism*, which is an operation performed upon a program and which may be thought of as a compilation. The *vanilla* view of moving from a program to a parallel implementation is thus seen as a compilation phase, which generates an atomic task graph, which is partitioned (by merging of tasks) into something that may be referred to as a grain graph which is mapped to processors. The process is summed up diagrammatically as the vertical flow in Figure 5.

The first problem with such a view is that the compilation phase—that is the phase where a task graph is generated from the expression of the sequential program—is not usually tractable. For a review of the subject see Padua and Wolfe [1986] or Polychronopoulos [1988]. In practice the best that compiler technology can do, with standard languages, is the partial unravelment of parallelism. In general short range independencies can be found—aided by the programmer's use of constructs such as DOACROSS (proposed by Cytron [1986]) and DOALL (see Zima [1990], chapter 7 for explanation of both), but inter-procedural analysis is still the subject of intense research effort (eg. Callahan and Kennedy [1988]). As a result, the graph of dependencies that the compilation generates, which contains more than just simple precedence relations (eg. Ferrante *et al.* [1987]), will overspecify the sequentiality of the underlying algorithm, although not quite as drastically as the sequential program.

If we take the partitioning phase—even assuming an absolutely correct compiler—we still face a number of problems with our *vanilla* approach. We can imagine some partitioning heuristic which aims to minimise inter-grain communication, perhaps by tending to allocate tasks which communicate with each otehr to a single grain. General heuristics for this process have been

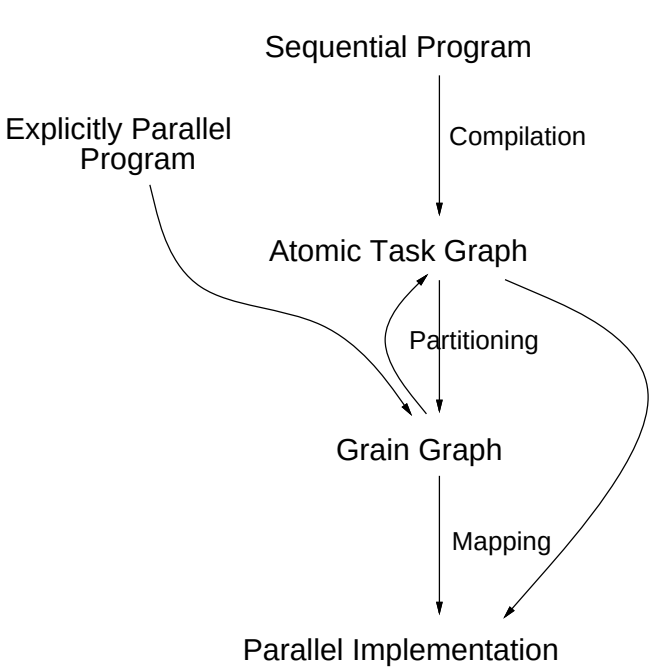

Figure 5: Parallel Programming

proposed, for example, by McGreary and Gill [1989], Agrawal and Jagadish [1988], Kruatrachue and Lewis [1988] and Sarkar [1989]. Algorithms for dealing specifically with some types of numerical programs are given by Peir and Cytron [1989] and Berger and Bokhari [1987].

There are problems with separating the partitioning and mapping phases, some of which are outlined in Kruatrachue and Lewis [1988]. If we, to use the terminology of McCreary and Gill [1989], restrict the partitoning to defining *clans*—that is subgraphs all of whose constituent nodes share common ancestor or descendant relationships—then the grain graph resulting from partitioning a directed acyclic task graph is always another directed acyclic graph. Since we assume that *clans* commence when scheduled, subject to the input to all their constituent tasks having arrived, and communicate when all their constituent processes have terminated, we are able to use standard scheduling techniques such as those outlined in sections 6 and 7 to map the grain graph to processors. In terms of our *a priori* model the partitioning phase, by merging tasks into grains, is avoiding the T_{Comm} of communications within the grain. We still have to take into consideration the effect of latency of communication and load imbalance, that is we make considerations of T_{Idle}.

We can consider a technique, such as that used by Kruatrachue and Lewis [1988] which mapped the task graph directly rather than the grain graph (this is symbolised by a curved arrow bypassing partitioning in Figure 5), and one can clearly imagine ways in which the optimality of grouping of tasks in the partitioning phase would depend upon properties of the processor system to which they were to be mapped. Moreover, if partitioning is restricted to producing a directed acyclic graph, where the grains only communicate on termination, this itself may not be the optimal partition of the task graph. Indeed, it would seem to make little sense for communication to be constrained to occur in bursts after large amounts of computation, since the transfer of results of tasks within a grain is being delayed which might, for example, correspond to an extension of the critical task path.

An alternative approach is to allow partitioning to generate grains which communicate during their computation. This allows cyclic grain graphs and, indeed turns directed relations of

precedence between grains into undirected volumes of communication, analogous to model 5. The problems with such models have been outlined in Section 8: the processing associated with sending and receiving communication has been packaged up into the grains that are being mapped, (except for T_{Route}, which we are ignoring for the purposes of this discussion). Moreover the *latency* of communication cannot be handled by the model since it would require detailed scheduling information at the task level in order to determine when it was that grains were being interrupted.

9.2 Explicitly Parallel Languages

Given the above discussion we can introduce the mapping problems of the programmer who has formulated his problem as a set of communicating sequential processes. Here the programmer has performed some of the process referred to as compilation and partitioning in Figure 5. In the case where the program is described as a dataflow graph it may be considered to be expressed as an atomic task graph. In the case of an occam program, or a c or FORTRAN program with message passing extensions he can be seen to have specified a grain graph. In such cases the parallel programming system has two choices: it may map the grain graph the programmer has supplied, or it may derive from it an atomic task graph (symbolised as the upward pointing arrow on Figure 5), and map that—with or without an intervening re-partition. We can imagine a case where the second scenario would generate performance benefits: a programmer specifies an object oriented program. The objects are modular, and the natural level of granularity at which to map is the object or the object class. If, however, one of the methods associated with one of the objects constitues the majority of the computation of the program, it would be advantageous to map it at a granularity finer than the object or object class.

9.3 Uncertainty in Parameters

To round up our discussion of the problems with the mapping problem, this review has concentrated on mapping of static task graphs where the labels on the edges and the nodes are known to be exact. As alluded to at the beginning of the section, the compiler is not in a position to generate such task graphs from programs written in standard languages. If the programmer does it, the labels will be—to quote Martin and Estrin [1967]—"estimated in an environment of ignorance". A number of authors have looked at the availability of parameters of models. The problems of deriving parameters clearly depend upon the model under consideration. Antonelli *et al.* [1989] use a model similar to Model 5, and derive the communication and computation costs with the aid of a heuristic which considers statements will be executed at a frequency which is dependent upon the depth of loop nesting in which they are found.

In the case of models including precedence relations it is not enough to consider the mean replication frequency of tasks and communications, it is also necessary to construct an instance of an expected task graph. Martin and Estrin consider the transformation process from acyclic representations (eg. those containing loops) to *mean-value* equivalent acyclic graphs. Such an acyclic graph may be thought of as a typical execution. Chu *et al.* [1984] introduce probabilities into the precedence relation, that is the execution of a precedent task may or may not cause the execution of another task, and its associated communication, with a given probability. They use this information to determine an expectation of the communications that will occur between tasks in an acyclic graph in multiple executions of the same task graph.

Mapping algorithms may give bounds on optimality of scheduling, but no such guarantees

will be given for their effectiveness in the presence of uncertain data. Indeed, the task graph structure itself may be uncertain, and un-derivable before the program runs. In this case dynamic algorithms would appear to be called for, and these are outwith the scope of the review (See Casavant and Kuhl [1988] for some references). We might imagine that mapping strategies which are not based upon scheduling (Such as those using Model 5) would be less amenable to problems of dynamic task graphs, but in that case the problems of inappropriate granularity, discussed above, imply that if grains were mapped statically, it would be necessary to allocate computation to them (ie to partition) dynamically.

10 Concluding Remarks

Our concluding remarks are addressed to two sets of readers, namely those intending to *do* research on the mapping problem and those intending to *use* existing research results on the mapping problem.

For applications programmers who intend to use the techniques described in this review, we give the following advice. Firstly, you should assess the relevance of the model implicit in a particular technique to your multicomputer. The summaries contained in the tables associated with the various models should be compared to the characteristics of the hardware. For example, is it a gross distortion to treat interprocessor communications delay as a quantity that is independent of the relative location in the interconnection network? (If, however, your multicomputer resembles Model 4, we suggest that you try to persuade your boss to buy a new multicomputer.)

Secondly, you should assess the relevance of a proposed technique to your particular computation. For example, over how many executions of the software shall the cost of applying a mapping technique be ammortized? If the software "behaves" the same for different inputs of the same size, then it may be worthwhile using a more sophisticated technique.

As we have already stated, any modelling exercise simplifies reality. Each of the mapping techniques that we have described embody a computational model that will simplify, or even ignore, various features of your computation. For example, is there a huge variation between the sizes of the data structures passed between different pairs of tasks corresponding to the edges in your task graph? If so, and you attempt to use a model which treats these quantities uniformly, then this convenient fiction may be a gross distortion.

In any review, such as the present one, it is obligatory to toss out a few remarks on "promising research areas" for the benefit of those readers intending to carry out research in this field. (Of course, if we really *did* have any such ideas, we would be working on them ourselves.)

Multicomputer mapping presupposes the existence of a multicomputer. It is strange, therefore, that very little work has been done on speeding up the mapping process by doing it in parallel. Even though the non-trivial aspects of scheduling algorithms are not known to be in NC, there is still scope for investigating the use of parallelism at a coarser granularity. For example, one might use a form of "speculative parallelism" (see Carriero and Gelerntner [1990]), by running several different scheduling algorithms in parallel, possibly on different processors, and comparing partial results.

As a second example, randomized optimization techniques, such as simulated annealing, can usefully be implemented in parallel. Helerstrom and Kanal [1990] and Udiavar and Stiles [1990] are examples of the approach, but there has been little in the way of systematic application of such techniques to the mapping problem. This may be due to the difficulty in formulating the

way in which the search space should be explored by such techniques.

Trends in the economics and technology of multicomputers suggest that, in the future, research on the mapping problem should be focussed more on the use of the interconnect than on processor utilization. For example, very little has been published on contention in the communications resource during multicomputer computation. Of course, this is partly due to the computational cost of modelling such behaviour. It may be, however that developments in VLSI or optical technology will separate out the interconnect from the processing unit and render graph or queuing network based models of multicomputers inappropriate.

11 Acknowledgements

Edinburgh Parallel Computing Centre is a multidisciplinary project supported by major grants from the Department of Trade and Industry, the Informations Systems Commitee of the Universities Funding Council and the Science and Engineering Research Council. M.G. Norman is supported by the SERC Contract B18534: Novel Architecture Computing Research.

References

Adam, T., Chandy, K., and Dickson, J. (1974). A comparison of list schedulers for parallel processing systems. *Comm. ACM*, 17(12):685–690.

Agrawal, R. and Jagadish, H. (1988). Partitioning techniques for large grain parallelism. *IEEE Trans. Comput.*, C-37(12):1627–1634.

Al-Mouhammed, M. (1990). Lower bound on the number of processors and time for scheduling precedence graphs with communication costs. *IEEE Trans. Software Engrg.*, SE-16(12):1390–1301.

Antonelli, S., Baiardi, F., Pelagatti, S., and Vanneschi, M. (1989). Communication cost and process mapping in massively parallel systems: a static approach. Technical Report TR-12/89, Dipartimento di Informatica, Università di Pisa.

Baccelli, F. and Liu, Z. (1990). On the execution of parallel programs on multiprocessor systems — a queueing theory approach. *J. ACM*, 37(2):373–414.

Bal, H., Stenier, J., and Tanenbaum, A. (1989). Programming languages for distributed computing systems. *Computing Surveys*, 21(3):261–322.

Baxter, J. and Patel, J. (1989). The LAST algorithm: A heuristic based static task allocation algorithm. In *Proc. Intl. Conf. Parallel Comput.*, volume 2, pages 217–222.

Berger, M. and Bokhari, S. (1987). A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput.*, C-36(5):570–581.

Berman, F. and Snyder, L. (1987). On mapping parallel algorithms into parallel architectures. *J. Parallel Dist. Comput.*, 4:439–458.

Błażewicz, J., Drabowski, M., and Węglarz, J. (1986). Scheduling multiprocessor tasks to minimize schedule length. *IEEE Trans. Comput.*, C-35(5):389–393.

Błażewicz, J., Węglarz, J., and Drabowski, M. (1984). Scheduling independent 2-processor tasks to minimise schedule length. *Inform Process. Lett.*, 18(5):267–273.

Bokhari, S. (1981a). On the mapping problem. *IEEE Trans. Comput.*, C-30(3):207–214.

Bokhari, S. (1981b). A shortest tree algorithm for optimal assignments across space and time in a distributed computer system. *IEEE Trans. Software Engrg.*, SE-7(6):583–589.

Bokhari, S. (1988). Partitioning problems in parallel, pipelined and distributed computing. *IEEE Trans. Comput.*, C-37(1):48–57.

Brent, R. (1974). The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206.

Bruno, J., Coffman Jr., E., and Sethi, R. (1974). Scheduling independent tasks to reduce mean finishing time. *Comm. ACM*, 17(7):382–387.

Callahan, D. and Kennedy, K. (1988). Analysis of interprocedural side effects in a aprallel programming environment. *J. Parallel Dist. Comput.*, 5:517–550.

Carriero, N. and Gelernter, D. (1990). *How to Write Parallel Programs*. MIT Press, Cambridge Mass.

Casavant, T. and Kuhl, J. (1988). A taxonomy of scheduling in general purpose distributed computing systems. *IEEE Trans. Software Engrg.*, SE-14(2):141–154.

Chen, G.-I. and Lai, T.-H. (1988). Scheduling independent jobs on hypercubes. In Cori, R. and Wirsing, M., editors, *Proc Conf. Theoretical Aspects of Computer Science*, pages 273–280.

Chen, M.-S. and Shin, K. (1987). Processor allocation in a n-cube multiprocessor using gray codes. *IEEE Trans. Comput.*, C-36(12):1396–1407.

Chen, N. and Liu, C. (1975). On a class of scheduling algorithms for multiprocessor computing systems. In Feng, T.-Y., editor, *Lecture Notes in Computer Science*. Springer, New York.

Cho, Y. and Sahni, S. (1980). Bounds for list schedules on uniform processors. *SIAM J. Comput.*, 9(1):91–103.

Chou, T. and Abraham, J. (1982). Load balancing in distributed systems. *IEEE Trans. Software Engrg.*, SE-8(4):401–402.

Chrétienne, P. (1989). A polynomial algorithm to optimally schedule tasks on a virtual distributed system under tree-like precedence constraints. *European J. Oper. Res.*, 43:225–230.

Chu, L., Lan, M.-T., and Hellerstein, J. (1984). Estimation of intermodule communication (IMC) and its applications in distributed processing systems. *IEEE Trans. Comput.*, C-33(8):691–699.

Chu, W. (1969). Optimal file allocation in a multiple computer system. *IEEE Trans. Comput.*, C-18(10):885–889.

Chu, W., Holloway, L., Lan, M.-T., and Efe, K. (1980). Task allocation in distributed data processing. *Computer*, 13(11).

Chu, W. and Lan, M.-T. (1987). Task allocation and precedence relations for distributed real-time systems. *IEEE Trans. Comput.*, C-36(6):667–679.

Clarke, L. (1990). *Achieving Parallel Performance in Scientific Computations*. PhD thesis, University of Edinburgh.

Coffman Jr., E., editor (1976). *Computer and Job Shop Scheduling Theory*. John Wiley, New York.

Coffman Jr., E., Flatto, L., and Leuker, G. (1984a). Expected makespans for largest-first multiprocessor scheduling. In Gelenbe, E., editor, *Performance '84*, pages 491–506. Elsevier Science Publishers B.V. (North Holland).

Coffman Jr., E., Garey, M., and Johnston, D. (1978). An application of bin-packing to multiprocessor scheduling. *SIAM J. Comput.*, 7(1):1–17.

Coffman Jr., E., Garey, M., and Johnston, D. (1984b). Approximation algorithms for bin-packing – an updated survey. In Ausellio, G., Lucertini, M., and Serafini, P., editors, *Algorithm Design for Computer System Design*, Berlin. Springer-Verlag.

Coffman Jr., E. and Graham, R. (1972). Optimal scheduling for two processor systems. *Acta Informatica*, 1:200–213.

Cole, R. and Vishkin, U. (1988). Approximate parallel scheduling, part 1: The basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput.*, 17(1):128–142.

Conway, R., Maxwell, W., and Miller, L. (1967). *Theory of scheduling*. Addison-Wesley, Reading, Mass.

Cvetanovic, Z. (1987). The effects of problem partitioning, allocation and granularity on the performance of multiple-processor systems. *IEEE Trans. Comput.*, C-36(4):421–432.

Cytron, R. (1986). Doacross: Beyond vectorisation for multiprocessors (extended abstract). In *Proc. International Conference on Parallel Processing*, pages 836–844.

Dally, W. (1990). Network and processor architecture for message-driven computers. In Suaya, R. and Birtwhistle, G., editors, *VLSI and Parallel Computation*, pages 140–222. Morgan Kaufmann, Palo Alto, CA.

Dolev, D., Upfal, E., and Warmuth, M. (1986). The parallel complexity of scheduling with precedence constraints. *J. Parallel Dist. Comput.*, 3:553–576.

Du, J. and Leung, J.-T. (1989). Complexity of scheduling parallel task systems. *SIAM J. Disc. Math*, 2(4):473–487.

Efe, K. (1982). Heuristic models of task assignment scheduling in distributed systems. *IEEE Computer*, June 1982:50–56.

El-Rewini, H. and Lewis, T. (1990). Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Dist. Comput.*, 9:138–153.

Fellows, M. and Langston, M. (1988). Processor utilisation in a linearly connected parallel processing system. *IEEE Trans. Comput.*, C-37(5):594–603.

Fernández, E. and Bussell, B. (1973). Bounds on the number of processors and time for multiprocessor optimal schedules. *IEEE Trans. Comput.*, C-22(8):745–751.

Fernández-Baca, D. (1989). Allocating modules to processors in a distributed system. *IEEE Trans. Software Engrg.*, SE-15(11):1427–143.

Ferrante, J., Ottenstein, K., and Warren, J. (1987). The program dependence graph and its use in optimisation. *ACM Transactions on Programming Languages and Systems*, 9(3).

Fox, G., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Walker, D. (1988). *Solving Scientific Problems on Concurrent Processors*. Prentice Hall, New Jersey.

Gabow, H. (1988). Scheduling UET systems on two uniform processors and length two pipelines. *SIAM J. Comput.*, 17(4):810–811.

Garey, M., Graham, R., and Johnston, D. (1977). Performance guarantees for scheduling algorithms. *Operations Research*, 26(1).

Garey, M. and Johnson, D. (1975). Complexity results for multiprocessor scheduling with

resource constraints. *SIAM J. Comput.*, 4(4):396–411.

Garey, M. and Johnson, D. (1977). Two-processor scheduling with start-times and deadlines. *SIAM J. Comput.*, 6(3):416–426.

Garey, M. and Johnson, D. (1979). *Computers and Intractability*. W.H. Freeman and Co., San Francisco.

Garey, M., Johnson, D., and Stockmeyer, L. (1976). Some simplified NP-complete graph problems. *Theor. Comput. Sci.*, 1:237–267.

Garey, M. and Johnson, R. (1982). Approximation algorithms for bin packing problems – a survey. In G. Ausellio, M., editor, *Analysis and Design of Combinatorial optimisation*, pages 147–172. Springer Verlag, Vienna, Austria.

Gaudiout, J., Pi, J., and Campbell, M. (1988). Program graph allocation in distributed multicomputers. *Parallel Computing*, 7:227–247.

Gonzalez, M. (1977). Deterministic processor scheduling. *Computing Surveys*, 9(3).

Graham, R. (1966). Bounds for certain multprocessing timing anomalies. *Bell System Technical J.*, 45:1563 – 1581.

Graham, R. (1969). Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17(2):416–429.

Graham, R., Lawler, E., Lenstra, J., and Rinnooy Kan, A. (1979). Optimisation and approximation in deterministic sequencing and scheduling a survey. *Annals of Discrete Mathematics*, 5:287–236.

Gusfield, D. (1983). Parametric combinatorial computing and a problem of program module distribution. *J. ACM*, 30(3):551–563.

Gylys, V. and Edwards, J. (1976). Optimal partitioning of workload for distributed systems. In *Proc. Compcon Fall 76*.

Hellstrom, B. and Kanal, L. (1990). Asymmetric mean-field neural networks for multiprocessor scheduling. Technical Report UMIACS-TR-90-99, University of Maryland Institute for Advanced Computer Studies.

Helmbold, D. and Mayr, E. (1987). Two processor scheduling is in NC. *SIAM J. Comput.*, 16(4):747–759.

Hochbaum, D. and Shmoys, D. (1988a). A polynomial approximation scheme for scheduling on uniform processors using the dual approximation approach. *SIAM J. Comput.*, 17(3):539–551.

Hochbaum, D. and Shmoys, D. (1988b). Using dual approximation algorithms for scheduling problems: Theoretical and practical results. *J. ACM*, 34(1):144–162.

Hopfield, J. and Tank, D. (1985). "neural" computation of decisions in optimisation problems. *Biological Cybernetics*, 52:141–152.

Houstis, C. (1990). Module allocation of real-time applications to distributed systems. *IEEE Trans. Software Engrg.*, SE-16(7):699–709.

Hu, T. (1961). Parallel sequencing and assembly line problems. *Oper Res.*, 9:841–848.

Hwang, J.-J., Chow, Y.-C., F.D., A., and C-Y, L. (1989). Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257.

Indurkhya, B. and Stone, H. (1986). Optimal partitioning of randomly generated parallel programs. *IEEE Trans. Software Engrg.*, SE-12(3):483–495.

Jung, H., Kirousis, L., and Spirakis, P. (1989). Lower bounds and efficient algorithms for multiprocessor scheduling of dags with communication delays. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 254–264.

Kafura, D. and Shen, V. (1977). Task scheduling on a multiprocessor system with independent memories. *SIAM J. Comput.*, 6(1):167–187.

Kapelinkov, A., Muntz, R., and Ercegovac, M. (1989). A modelling methodology for the anlaysis of concurrent systems and computations. *J. Parallel Dist. Comput.*, 6:568–597.

Kasahara, H. and Narita, S. (1984). Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. Comput.*, C-33(11):1023–1029.

Kaufmann, M. (1974). An almost-optimal algorithm for the assembly line scheduling problem. *IEEE Trans. Comput.*, C-23(11):1169–1174.

Krämer, O. and Mühlenbein, H. (1989). Mapping strategies in message-based multiprocessor systems. *Parallel Computing*, 9:213–225.

Krishnamurthy, S. (1990). A brief survey of papers on scheduling for pipelined processors. *Sigplan Notices*, 25(7):97–106.

Kruatrachue, B. and Lewis, T. (January 1988). Grain size determination for parallel programming. *IEEE Software*, pages 23–32.

Kruskal, C. and Snir, M. (1989). Cost-bandwidth tradeoffs for communication networks. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 32–41, New York. ACM.

Lam, S. and Sethi, R. (1977). Worst case analysis of two scheduling algorithms. *SIAM J. Comput.*, 6(3):518–536.

Lawler, E. (1982). Preemptive scehduling of precedence constrained jobs on parallel machines. In Dempster, M., editor, *Deterministic and Stochastic Scheduling*. D.Reidel Publishing Co.

Lee, C.-Y., Hwang, J.-J., Chow, Y.-C., and Anger, F. (1988). Multiprocessor scheduling with interprocessor communication delays. *Operations Research Letters*, 7(3):141–145.

Lee, S. and Aggarwal, J. (1987). A mapping strategy for parallel computing. *IEEE Trans. Comput.*, C-36(4):433–442.

Lo, V. (1988). Heuristic algorithms for task assignment in distributed systems. *IEEE Trans. Comput.*, 37(11):1384–1397.

Ma, P.-Y., Lee, E., and Tsuchiya, M. (1982). A task allocation model for distributed computing systems. *IEEE Trans. Comput.*, C-31(1):246–252.

Mak, V. and Lundstrom, S. (1990). Predicting performance of parallel computations. *IEEE Trans. Paral. Distr. Comput.*, 1(3):257–270.

Martel, C. (1988). A parallel algorithm for preemptive scheduling of uniform machines. *J. Parallel Dist. Comput.*, 5:700–715.

Martin, D. and Estrin, G. (1967a). Experiments on models of computation and systems. *IEE Trans. Electronic Computers*, EC-16(1):59–69.

Martin, D. and Estrin, G. (1967b). Models of computational systems – cyclic to acyclic graph transformations. *IEEE Trans. Elect. Comput.*, EC-16(1).

McDowell, C. and Appelbe, W. (1986). Processor scheduling for linearly connected parallel processors. *IEEE Trans. Comput.*, C-35(7):632–638.

McGreary, C. and Gill, H. (1989). Automatic determination of grain size for efficient parallel programming. *Comm. ACM*, 32(9).

McNaughton, R. (1959). Scheduling with deadlines and loss functions. *Management Science*, 6:1–12.

Muntz, R. and Coffman Jr., E. (1969). Optimal preemptive scheduling on two-processor systems. *IEEE Trans. Comput.*, C-18(11):1014–1020.

Nicol, D. (1989). Optimal partitioning of random programs across two processors. *IEEE Trans. Software Engrg.*, SE-15(2):134–141.

Nicol, D. and Saltz, J. (1990). An analysis of scatter decomposition. *IEEE Trans. Comput.*, C-39(11):1337–1345.

Padua, D. and Wolfe, M. (1986). Advanced compiler optimisations for supercomputers. *Comm. ACM*, 29(12).

Papadimitriou, C. and Ullman, J. (1987). A communication-time tradeoff. *SIAM J. Comput.*, 16(4):639–646.

Papadimitriou, C. and Yannakakis, M. (1990). Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.*, 19:322–328.

Peir, J.-K. and Cytron, R. (1989). Minimum distance: A method for partitioning recurrences for multiprocessors. *IEEE Trans. Comput.*, C-38(8).

Pinter, S. and Wolfstahl, Y. (1987). On mapping processes to processors in distributed systems. *Intl. J. Parallel Programming*, 16(1):1–15.

Pippenger, N. (1979). On simultaneous resource bounds (preliminary version). In *Proc. 20th IEEE FOCS*, pages 307–311.

Polychronopoulos, C. (1988). *Parallel Programming and Compilers*. Kluwer Academic Publishers, Norwell Mass.

Pountain, R. (1989). Configuring parallel programs. Part 1. *Byte*, (December):349–352.

Pritchard, D., Askew, C., Carpenter, D., Glendinning, I., Hey, A., and Nicole, D. (1987). Practical parallelism using transputer arrays. In deBackker, J., Nijman, A., and Treleaven, P., editors, *Parallel Architectures and Languages*, Lecture Notes in Computer Science, page 278. Springer Verlag, Berlin.

Ramamritham, K., Stankovic, J., and Shiah, P.-F. (1990). Efficient scheduling algorithms for multiprocessor systems. *IEEE Trans. Paral. Distr. Comput.*, 1(2):184–194.

Rao, G., Stone, H., and Hu, T. (1979). Assignment of tasks in a distributed processor system with limited memory. *IEEE Trans. Comput.*, C-28(4):291–298.

Rayward-Smith, V. (1987). The complexity of preemptive scheduling given interprocessor communication delays. *Inform. Process. Lett.*, 25(2):123–125.

Reed, D. and Fujimoto, R. (1987). Multicomputer network operating systems. In *Multicomputer networks : message-based parallel processing*, pages 177–238. MIT Press, Cambridge Mass.

Sahni, S. (1976). Algorithms for scheduling independent tasks. *J. ACM*, 23(1):116–127.

Sarkar, V. (1989). *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman,

London.

Seitz, C. (1990). Concurrent architectures. In Suaya, R. and Birtwhistle, G., editors, *VLSI and Parallel Computation*, pages 1–84. Morgan Kaufmann, Palo Alto, CA.

Sethi, R. (1976). Algorithms for minimal length schedules. In Coffman Jr., E., editor, *Computer and Job Shop Scheduling Theory*. John Wiley, New York.

Shen, C.-C. and Tsai, W.-H. (1985). A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Trans. Comput.*, C-34(3):197–203.

Shirazi, B., Wang, M., and Pathac, G. (1990). Analysis and evaluation of heuristic methods for static task scheduling. *J. Parallel Dist. Comput.*, 10:222–232.

Sinclair, J. (1987). Efficient comutation of optimal assignments for distributed tasks. *J. Parallel Dist. Comput.*, 4:342–362.

Stone, H. (1977a). Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Trans. Software Engrg.*, SE-3:85–93.

Stone, H. (1977b). Program assignment in three processor systems and tricutset partitioning of graphs. Technical Report ECE-CS-77-7, Univ. Massachusetts, Amherst.

Stone, H. (1978). Critical load factors in distributed systems. *IEEE Trans. Software Engrg.*, SE-4:254–258.

Tanenbaum, A. (1989). *Computer Networks*. Prentice Hall, New Jersey.

Thanisch, P. and Norman, M. (1990). Minimising message path lengths in multicomputers. Technical Report EPCC-TR-90-17, Edinburgh Parallel Computing Centre.

Towsley, D. (1986). Allocating programs containing branches and loops within a multiple processor system. *IEEE Trans. Software Engrg.*, SE-12(10):1018–1024.

Udiavar, N. and Stiles, G. (1990). A simple but flexible model for determining optimal task allocation and configuration on a network of transputers. In Stiles, G., editor, *Transputer Research and Applications 1*, pages 24–32. IOS, Amsterdam.

Ullman, J. (1975). NP-complete scheduling problems. *J. of Computer and System Sciences*, 10:384–393.

Upfal, E. (1984). Efficient schemes for parallel communication. *J. ACM*, 31(4).

Valiant, L. (1982). A scheme for fast parallel communication. *SIAM J. Comput.*, 11:350–361.

Valiant, L. (1990). A bridging model for parallel computation. *Comm. ACM*, 33:103–111.

Valiant, L. and Brebner, G. (1981). Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pages 263–267. ACM, New York.

Vazirani, U. and Vazirani, V. (1989). The two-processor scheduling problem is in random NC. *SIAM J. Comput.*, 18(6):1140–1148.

Veltman, B., Lageweg, B., and Lenstra, J. (1990). Multiprocessor scheduling with communciation delays. *Parallel Computing*, 16:173–182.

Wallace, D. (1991). Algorithms and architectures for grand challenges in physics. In *Proc. Conference on Very Large Scale Computing in the 21st Century*. SIAM Press.

Williams, E. (1983). Assigning processes to processors in distributed systems. In *Proceedings IEEE Conference on Parallel Processing*, pages 404–406.

Xhao, Ramamritham, K., and Stankovic, J. (1987). Preemptive scheduling under time and resource constraints. *IEEE Trans. Comput.*, C-36(8):949–960.

Zima, H. and Chapman, B. (1990). *Supercompilers for Parallel and Vector Computers*. ACM, New York.