A Complexity Theory of Parallel Computation

Ian Parberry

A dissertation submitted for the degree of
Doctor of Philosophy

University of Warwick
Department of Computer Science

May 1984

Dedicated to my wife and my parents,
without whose support and encouragement
this work would not have been possible.

Contents.

Diagrams.

Tables.

Acknowledgements.

I would like to thank my supervisor, Mike Paterson, for his expertise and guidance during all stages of this work. I would be satisfied if it could be said that the technical content comes close to meeting his exceptionally high standards for conciseness, clarity and elegance. I am grateful to a number of people who helped keep me up-to-date with the latest developments in parallel complexity theory by sending manuscripts and correspondence. These include Allan Borodin, Patrick Dymond, Zvi Galil, Tom Leighton, Friedhelm Meyer auf der Heide, Nick Pippenger, Uzi Vishkin and Avi Wigderson. The Commonwealth Scholarship Commission, via their awarding body in Australia, provided the financial support which enabled me to travel to England and undertake this degree. Finally, I would like to thank Tony Cohn for assistance with typesetting problems, Rod Moore for the loan of drawing equipment, and Meurig Beynon for the use of his office while preparing this manuscript.

Declaration.

In the interests of rapid dissemination, preliminary versions of the results in section 4.4, chapter 5 and sections 6.1-6.3 have been published as internal Theory of Computation Reports. Material from chapters 4, 5 and 6 can be found in [51], [50] and [52] respectively.

Summary.

Parallel complexity theory is currently one of the fastest growing fields of theoretical computer science. This rapid growth has led to a proliferation of parallel machine models and theoretical frameworks. Our aim is to construct a unified theory of parallel computation based on a network model. We claim that the network paradigm is fundamental to the understanding of parallel computation, and support this claim by providing new and improved theoretical results, and new approaches to old questions concerning "reasonable" and "practical" models.

This thesis is made up of eight chapters. Chapter 1 contains the introduction. In chapter 2 we define the basic model, and justify our choice of a unit-cost measure of time, a uniform assignment of programs to processors, and simultaneous processor activation. Chapter 3 compares the network model to a variety of others, including fixed-structure networks and shared-memory machines. We explore the concepts of "reasonableness" and "practicality" in parallel machine models, and show that even "reasonable" parallel computers are much faster than sequential ones.

Chapter 4 is devoted to programming techniques for a "practical" network model, (which we call a *feasible* network), covering interconnection patterns, useful algorithms, and some processor-saving theorems. In chapter 5 we find efficient simulations of the general network model on more practical machines, including a universal feasible network, and uniform circuits. Chapter 6 extends the network model, and defines a new resource, that of *arity*. Although increasing arity increases computing power, some efficient constant-arity universal machines are found. Chapter 7 takes a final look at universal networks, concentrating on lower-bounds and the conditions under which they hold. Chapter 8 contains the conclusion.

Chapter 1
Introduction

As recently as 1980, Schwartz [62] complained of an apparent lack of theoretical results concerning the computational complexity of parallel or concurrent algorithms.

"In the serial case, the design of algorithms has come to be illuminated by a growing body of theoretical knowledge concerning the ultimate limits of algorithm performance.... Until a like body of theoretical knowledge has been developed for highly concurrent algorithms, we will have little basis for judging the extent to which a given concurrent approach can be improved."

Two of the most important and fundamental papers in the field of parallel complexity theory (that of Goldschlager [23], later to become [27], and that of Pippenger [53]) had already appeared by the time Schwartz's paper reached publication. Since then, the flow of results has increased from a trickle to a steady stream, and is now threatening to become a flood. Today, parallel complexity theory must be ranked as one of the fastest-growing fields of theoretical computer science.

A theoretical treatment of parallel computation is an attempt to formalize the intuitive concept of a "parallel computer" based on practical experience or reasonable expectations. Amongst the questions which should be addressed by such a formal exposition are the following:

What do we mean by a parallel computation?

What is a good model of a parallel computer?

What are the resources of interest, and how should they be defined?

How should we design a parallel programming language?

Are parallel machines necessarily faster than sequential ones?

What kind of problems can be solved significantly faster by using a parallel algorithm?

Can we obtain asymptotically optimal upper and lower bounds on the parallel resources needed to compute some "natural" functions?

The latter problem appears to be the most popular, judging by sheer volume of contributions (for some examples, consult [56,73] and the references contained therein). In comparison, relatively little attention has been paid to the first four questions, resulting in a proliferation of parallel machine models in the current literature. Even the most popular model, the shared-memory machine (consisting of a collection of RAMs communicating via a shared memory, see, for example, [20,27]) has many variants. There is a growing tendency to "customize" a machine to allow short, elegant proofs of a particular upper or lower bound, with scant regard to the suitability of the model as a vehicle for further research.

Intuitively, a parallel machine should consist of many processors which in some way co-operate in order to compute a result. Obviously there are many ways of formalizing this intuition. Compare the SIMDAG of Goldschlager [27] to the model of Galil and Paul [21]. The processors of the former are RAMs, the latter allow RAMs, RACs or even finite-state machines. Lev, Pippenger and Valiant [42] insist that they must be RACs. Goldschlager has almost *identical* processors which are all started simultaneously at the start of the computation, and communicate via a shared memory. Galil and Paul have *similar* processors which start up at run-time, and communicate via direct processor-to-processor links. Some of the more obvious variations on these models include the instruction-set (for example, should multiplication be allowed?), and memory access conflicts (should multiple attempts to write to a shared register be allowed as in [27], or should even multiple reads be disallowed, as in [42]?).

Some of these differences are merely cosmetic in nature, but some are extremely important. In order to design a useful parallel machine model, we must first determine which choices matter. We have chosen a model which consists of a network of interconnected RAMs; each RAM can in one step perform an internal computation, or read from or write to a register belonging to one of its neighbours. We believe that the network paradigm is fundamental to the understanding of parallel computation. One attraction is the fact that it possesses a certain theoretical elegance. A RAM is just a network consisting of one processor. A shared-memory machine is just a network where all processors can communicate with a single distinguished processor and no other, and that distinguished processor remains idle throughout the computation. The number of extant papers which use the shared-memory model attest to its ease of programming, and its usefulness as a tool for proving and communicating theoretical results. It is widely accepted, however, that the shared-memory model is not in itself a viable architecture. By placing restrictions on our network model, it is possible to define a practical variant in a far more natural way than is possible with shared-memory machines. This makes the network approach doubly attractive.

The aim of this thesis, then, is to shed some light on the nature of parallel computation. We shall do this by presenting a unified theory of parallel computation based on our network model. We shall demonstrate its utility by providing some fairly concise and elegant alternate proofs of results from the current literature, which will quite often lead to improved resource bounds or more general theorems. We will also attempt to provide answers to the questions posed at the start of this introduction, based on this theory.

The main body of this thesis is made up of six chapters. In chapters 2 and 3 we design and justify our parallel machine model. In section 2.1 we define the basic model, and in section 2.2 discuss the consequences of choosing unit-cost RAMs as opposed to log-cost RAMs as processors. This decision can be summed up by what we call the *unit-cost hypothesis:* "the unit-cost measure of time is a valid one for parallel processors". We will refer to this hypothesis often throughout the thesis, and conclude that it holds in most situations of interest. Section 2.3 discusses our assignment of programs to processors, comparing and contrasting it to the SIMD and MIMD approaches of Flynn [19], and section 2.4 our decision to have all processors activated simultaneously at the start of the computation.

Chapter 3 compares the basic network machine to a selection of other models. In section 3.1 we propose an alternative fixed-structure variant. It is shown that a fixed-structure network of $2^{O(n)}$ processors and a non-recursive interconnection pattern can compute any single-valued Boolean function in a constant number of steps, using an instruction-set consisting of addition, subtraction and logical shifts. In section 3.2 we compare networks to shared-memory machines,and conclude that they are almost identical. In section 3.3 we discuss the possible bounds which need to be placed on the resources of our parallel machines in order to make them "reasonable" or "practical". The *parallel computation thesis* of Goldschlager [27] states that time on a "reasonable" model of parallel computation should be polynomially equivalent to sequential space. Goldschlager places strong restrictions on his SIMDAGs (a variant of the shared-memory model considered in section 3.2) in order to make them obey the parallel computation thesis. We find that much less strict bounds are sufficient.

In the light of this discussion, in section 3.4 we define a practical variant of our network model, which we call a ***feasible network***. This has:

(1) Constant degree.

(2) A constant number of registers per processor.

(3) An easy-to-compute interconnection pattern.

(4) Fixed structure.

We will find later that there is an efficient feasible network which is universal for the general model of section 2.1. Thus the user of such a universal machine has the freedom to program in a high-level language which corresponds to a more powerful architecture at little cost, and the theoretician is provided with a motivation for studying the more abstract models.

Section 3.5 is devoted to exploring the speed-ups which can be made by a parallel machine as opposed to a sequential one. Let $B:N \rightarrow N$ be an arbitrary function. Then a $T(n)$ time-bounded deterministic Turing machine can be simulated in time $O(T(n)/B(n))$ by a network of $2^{O(B(n)^2)}+T(n)$ processors. By choosing $B(n)=T(n)$ we find that every function in NP can be computed in constant time by a network of $2^{n^{O(1)}}$ processors. Choosing $B(n)=T(n)^{1-\varepsilon}$ for some $\varepsilon>0$ we find that an arbitrary polynomial speed-up is possible on a machine which obeys the parallel computation thesis. This is a striking result, because an exponential speed-up is not possible for certain natural problems in P unless P⊆POLYLOGSPACE.

In chapter 4 we develop the techniques necessary for the construction of our universal feasible network. Section 4.1 demonstrates the usefulness of the shuffle-exchange [66] and cube-connected-cycles [55] interconnection patterns. In section 4.2 we present a recurrent interconnection pattern CCL_k with 2^k processors and degree 3 with the property that for all $k \geq j \geq 0$, CCL_k has at least 2^{k-j-1} disjoint subgraphs which are isomorphic to CCL_j, yet it is at least as our network model, which we call a *feasible network*. This has:

(1) Constant degree.

(2) A constant number of registers per processor.

(3) An easy-to-compute interconnection pattern.

(4) Fixed structure.

We will find later that there is an efficient feasible network which is universal for the general model of section 2.1. Thus the user of such a universal machine has the freedom to program in a high-level language which corresponds to a more powerful architecture at little cost, and the theoretician is provided with a motivation for studying the more abstract models.

Section 3.5 is devoted to exploring the speed-ups which can be made by a parallel machine as opposed to a sequential one. Let $B: N \to N$ be an arbitrary function. Then a $T(n)$ time-bounded deterministic Turing machine can be simulated in time $O(T(n)/B(n))$ by a network of $2^{O(B(n)^2)} + T(n)$ processors. By choosing $B(n) = T(n)$ we find that every function in NP can be computed in constant time by a network of $2^{n^{O(1)}}$ processors. Choosing $B(n) = T(n)^{1-\varepsilon}$ for some $\varepsilon > 0$ we find that an arbitrary polynomial speed-up is possible on a machine which obeys the parallel computation thesis. This is a striking result, because an exponential speed-up is not possible for certain natural problems in P unless $P \subseteq$ POLYLOGSPACE.

In chapter 4 we develop the techniques necessary for the construction of our universal feasible network. Section 4.1 demonstrates the usefulness of the shuffle-exchange [66] and cube-connected-cycles [55] interconnection patterns. In section 4.2 we present a recurrent interconnection pattern CCL_k with 2^k processors and degree 3 with the property that for all $k \geq j \geq 0$, CCL_k has at least 2^{k-j-1} disjoint subgraphs which are isomorphic to CCL_j, yet it is at least as powerful as the cube-connected-cycles. Further, using the techniques of Meertens [43] we show that any similar interconnection pattern with 2^{k-1} such disjoint subgraphs cannot share this property. Section 4.3 contains some useful algorithms, and section 4.4 some processor-saving theorems. The latter show that for any machine based on the above interconnection patterns, a P(n) processor network can be simulated on one with P'(n) processors, with a time-loss of O(P(n)/P'(n)) for each step. Thus constant multiples in processor-bounds can be ignored without asymptotic time-loss, a fact which simplifies many of our later proofs. A preliminary version of the results of section 4.4 has appeared in [51].

Chapter 5 considers simulations of networks by more practical models. In section 5.1 a general machine-independent simulation theorem is given. Specific instances of this theorem have been seen before in the literature (see, for example, [4,8,21,42,45,71,72]). We use it in section 5.2 to construct our universal feasible network, and again in section 5.3 to simulate networks on width and depth bounded uniform circuits and space and reversal bounded deterministic Turing machines. In section 5.4 we build upon the latter results to improve Pippenger's [53] simulation of space and reversal bounded Turing machines by width and depth bounded uniform circuits. More specifically, a k-tape Turing machine with space S(n) and reversals R(n) can be simulated by a uniform circuit of width $O(S(n)^k)$ and depth $O(R(n).\log^2 S(n).\log\log S(n))$. A preliminary version of the work in chapter 5 has appeared in [50].

Chapter 6 generalizes our model to allow high-arity processors, that is, processors which have the power to communicate with more than a constant number of its neighbours in unit time (and the power to make good use of this ability). High-arity machines have appeared in [8,69,70]. Section 6.1 contains our high-arity model. In section 6.2 we show that increasing arity gives more computing power. In particular, a network with arity A(n) and a polynomial number of processors needs time $\Omega(\frac{\log n}{\log A(n)})$ to add n numbers, each of polynomial size, even in the presence of write-conflicts. Thus, for example, a polynomial-processor PRAM with multiple-writes needs time $\Omega(\log n)$ to add n polynomial-bit numbers. This is the first lower-bound of this nature to be achieved on a model which allows write-conflicts. Section 6.3 explores simulations of high-arity machines on constant-degree universal machines of arity 1. As a corollary, we obtain an improved proof of theorem 8 of [21]. A preliminary version of sections 6.1, 6.2 and 6.3 has appeared in [52]. Section 6.4 contains some examples of high-arity algorithms, most notably the parallel prefix problem.

Section 7.1 is devoted to lower-bounds for universal machines. The universal machine of chapter 5 is found to be optimal for simulations of that nature. The universal machine of chapter 6 is optimal for the simulation of degree-3 machines (Meyer auf der Heide [31] had earlier found it to be optimal for the simulation of constant-degree machines). Section 7.2 contains a new proof of a result of Meyer auf der Heide [33]. In section 7.3 we obtain asymptotic upper and lower bounds of $\Theta(\frac{P(n)}{\sqrt{P'(n)}} + \log P(n))$ for the oblivious simulation of a P(n) processor network on a constant-degree universal machine with P'(n) processors. This extends the results of Borodin and Hopcroft [8] and Lang [39], who prove the same lower and upper bound respectively for $P(n) = P'(n)$.

It should be noted that this is a *theoretical* treatment of parallel computation, and as such is based upon a number of assumptions which are widely accepted amongst workers in the field of parallel complexity theory. Although our model is synchronous (in the sense that the instruction-cycles of the processors are synchronized), we will see in section 3.4 that this is not an important restriction. The advantage of having a synchronous theoretical model is that it is easy to program and reason with. We assume that inter-processor communications can take place within a single instruction-cycle. In the real world, this assumption is unlikely to be true for large numbers of processors; a complexity theory based on this observation will differ quite radically from ours [61]. However, we feel fairly safe in making the assumption for networks consisting of a small number (say in the millions) of fairly large processors (about the size of a microprocessor), even though it is unlikely to hold for, say, individual gates in a VLSI chip.

Finally, the reader should note that throughout this work, all logarithms are to base 2, N denotes the set of non-negative integers, Z the set of integers, and if $c \in N$, $d \in Z$, then d *mod* c is defined to be the unique integer $a \in N$ such that $0 \leq a < c$ and there exists $b \in Z$ such that $a+bc=d$. For those unfamiliar with the "order" notation, we provide the following reminder. Let $f,g:N \rightarrow R^+$ (where R^+ denotes the set of positive real numbers). We say that:

(1) $f(n)=O(g(n))$ if there exists $c \in R^+$, $N \in N$ such that for all $n \geq N$, $f(n) \leq c.g(n)$.

(2) $f(n)=\Omega(g(n))$ if $g(n)=O(f(n))$.

(3) $f(n)=\Theta(g(n))$ if $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$.

(4) $f(n)=o(g(n))$ if $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}=0$.

Chapter 2
Designing a Parallel Machine Model

In this chapter we present our basic parallel machine model, and attempt to justify some of the decisions which contributed to its present form. Informally, the model consists of a network of interconnected random-access machines, or RAMs. In the first section we give a more formal description, providing illustration by way of an example RAM instruction-set. We define the major resources of interest; *processors* (number of RAMs), *time* (number of instructions executed), *degree* (degree of the interconnection pattern), *space* (number of registers required) and *word-size* (the size of those registers). In order to simplify the presentation of algorithms, a high-level pseudo-programming language is sketched.

The second section is devoted to a discussion of our choice of a unit-cost measure of time. We have chosen to charge a single unit of time for each instruction executed, rather than charge according to some notion of "difficulty". This raises an interesting question: for which instruction-sets is this a valid measure of time? We shall see in subsequent chapters that the answer can be provided in many different ways.

Our basic machines have a single program for all processors. In the third section we justify this approach, comparing and contrasting it with the SIMD and MIMD machines of Flynn [19]. In a SIMD machine, the processors have their program-counters synchronized, with each individual processor either executing the common current instruction or remaining dormant for a step. In contrast, we allow each processor to be at a potentially different point in the program. A MIMD machine has a different program for each processor. Our model is seen to be equivalent to a SIMD one, and to a reasonable subset of the MIMD model.

Finally, in the fourth section we justify our decision to start the computation with all processors active, rather than have them become active at run-time. This latter approach places a not altogether unreasonable upper-bound on the number of processors used in a computation, in relation to time. We shall see in chapter 3 that it is sometimes profitable to consider machines with a larger number of processors. Within this limitation, however, the two models are equivalent.

2.1. The Basic Model

Our parallel machine model can be loosely described as an infinite collection of interconnected random-access machines, only finitely many of which are active in any particular finite computation. By "random-access machine" we refer to a variant of the RAM, which is already well-known as a sequential machine model (see, for example, [1,12,63]); and by "synchronous" we mean that the instruction-cycles of the RAMs are synchronized. Each RAM has an infinite number of general-purpose registers $r_0, r_1, \ldots$, each of which is capable of storing a single integer, and a number of read-only registers which are initialized at the start of a computation. These include the processor identity register PID and the input-size register SIZE. The PID of the i^{th} RAM is preset to i, for $i = 0, 1, \ldots$.

More formally, a *network* M consists of a program and a processor-bound. The *program* of M is a finite list of instructions; each instruction has the form either:

(i) Read a value from a register of a neighbouring processor.

(ii) Write a value to a register of a neighbouring processor.

(iii) Perform an internal computation.

(iv) Conditional transfer of control, or halt.

For example, let "$\sim$" denote a binary operation defined on integers. For convenience we divide our example instruction-set into two categories. Local instructions have the form:

$r_i \leftarrow$ constant	(load register with constant)
$r_i \leftarrow r_j \sim r_k$	(binary operation)
$r_i \leftarrow r_{r_j}$	(indirect load)
$r_{r_i} \leftarrow r_j$	(indirect store)
$r_i \leftarrow R$	(store read-only register R)
halt	(end execution)
goto m **if** $r_i > 0$	(conditional transfer of control)

Communication instructions have the form:

$$r_i \leftarrow (r_{r_j} \text{ of } r_k) \quad \text{(read)}$$
$$(r_{r_i} \text{ of } r_j) \leftarrow r_k \quad \text{(write)}$$

The program is to be executed synchronously in parallel by the (finitely many) active processors. As far as local instructions are concerned, their behaviour is that of independent RAMs, that is, references to registers in local instructions are treated as references to their respective local registers. Execution of a read instruction:

$$r_i \leftarrow (r_{r_j} \text{ of } r_k)$$

by processor p has the following effect. Suppose registers r_j, r_k of processor p contain the values q and p' respectively. Then the contents of register r_q of processor p' are read and placed into register r_i of processor p. Similarly, execution of a write instruction:

$$(r_{r_i} \text{ of } r_j) \leftarrow r_k$$

by processor p has the following effect. Suppose registers r_i, r_j of processor p contain the values q and p' respectively. Then the contents of register r_k of processor p are written into register r_q of processor p'.

Multiple reads of the same register are allowed. In the case of multiple writes to a single register, we adopt some reasonable convention whereby a single processor *succeeds* and is allowed to write its value, whilst all others fail. For example (after [27]) the lowest-numbered processor attempting to write succeeds, or (as in chapter 5), the processor which is attempting to write the smallest value succeeds, with ties being broken in favour of the lowest-numbered processor. A local instruction must compete with incoming data on the same basis.

Suppose $f:Z^*\to Z^*$ and $x=\langle x_0,x_1,\ldots,x_{n-1}\rangle$, where $x_i\in Z$ for $0\le i<n$. We will say that x has *size* or *length* n, and write $|x|=n$. Let $m=\max_{|x|=n}|f(x)|$ and $f_n:Z^n\to Z^m$ denote the restriction of f to n arguments (we adopt the convention that unused output places are filled by zeros). We will variously refer to x as an *input* or *input string*, and each x_i as an *argument* or *input symbol*.

Suppose M is a network with processor-bound $P:N\to N$. Let $p=P(n)$. A *computation* of M on input x is defined as follows. Place x_i into register $r_{\lfloor i/p\rfloor}$ of processor (i mod p), and set all other general-purpose registers to zero. Set register SIZE of all processors to n. Simultaneously activate processors $0,1,\ldots,p-1$. These synchronously execute the program of M. For $0\le i<m$ let y_i denote the contents of register $r_{\lfloor i/p\rfloor}$ of processor (i mod p) when all processors have finally halted. We say that M *computes* f if for all $n\ge 0$ and inputs x with $|x|=n$, $f_n(x)=\langle y_0,y_1,\ldots,y_{m-1}\rangle$.

The *interconnection pattern* of M is an infinite family of finite graphs $G=(G_0,G_1,\ldots)$, one for each input-size. For $n\ge 0$, G_n has vertex-set $\{0,1,\ldots,P(n)-1\}$, and an edge between vertices i and j if at any time during the computation of M on an input of length n, processor i attempts to read from or write to a register of processor j. Let $D:N\to N$. M is said to have *degree* D(n) if for all $n\ge 0$, G_n has degree D(n).

Let $T, S, W : N \to N$. M is said to compute within *time* $T(n)$ if for all inputs of size n, all active processors have halted within $T(n)$ steps. For $0 \le t \le T(n)$, let $S_t(n)$ be the maximum (over all inputs of size n) number of registers of M with non-zero contents after t instructions have been executed. Then M uses *space* $S(n)$ if $S(n) = \max_{0 \le t \le T(n)} S_t(n)$. It has *word-size* $W(n)$ if every value which appears in a register during such a computation has absolute value less than $2^{W(n)}$ (note that this includes the inputs, outputs and processor identity registers).

Notes. (1) We have chosen a unit-cost measure of time. This choice will be discussed in more detail in section 2.2.

(2) The space bound is a measure of the number of registers used in a computation. It is slightly unusual - the more usual method (see, for example, [1] for the case of a single RAM) is to define space to be the number of registers which are assigned non-zero contents at any point during the computation. Our reasons will become more apparent in chapter 5.

(3) The word-size is a measure of the width of (inter- and intra-processor) data paths, and a measure of register size. This can be combined with our unit-cost measure of space to provide an upper-bound on log-cost space.

Consider the example instruction-set given earlier in this section. So far, we have not specified exactly which binary operations can be used for "$\sim$". In particular we will be interested in four types of instruction-sets. Each has two-input Boolean functions (defined on single-bit quantities) and the following integer operations.

(1) The *minimal* instruction-set allows addition, subtraction, shifts of a single bit, and extraction of the least-significant bit.

$r_i \leftarrow r_j \pm r_k$
$r_i \leftarrow \lfloor r_j / 2 \rfloor$
$r_i \leftarrow r_j \bmod 2$

(2) In addition, the *restricted arithmetic* instruction-set allows larger shifts and extractions. Suppose $r_k > 0$.

$r_i \leftarrow r_j * 2^{\lfloor \log r_k \rfloor}$
$r_i \leftarrow \lfloor r_j / 2^{\lfloor \log r_k \rfloor} \rfloor$
$r_i \leftarrow r_j \bmod 2^{\lfloor \log r_k \rfloor}$

(3) The *full arithmetic* instruction-set is the minimal instruction-set plus multiplication, integer division and remaindering.

$r_i \leftarrow r_j * r_k$
$r_i \leftarrow \lfloor r_j / r_k \rfloor$
$r_i \leftarrow r_j \bmod r_k$

(4) The *extended arithmetic* instruction-set is the full arithmetic instruction-set plus exponentiation.

$r_i \leftarrow r_j^{r_k}$

A number of questions spring to mind. Are these instruction-sets reasonable? Are they powerful enough? Too powerful? Natural? Clearly an unrestricted instruction-set which allows any computable function as a local instruction is too powerful, but what kind of instruction-set *is* reasonable? We will return to these questions in the next section.

Instead of writing algorithms in the low-level RAM language, we will follow the common practice of using a high-level language which can easily be translated into instructions of this form. We use the usual high-level constructs for flow-of-control, based on sequencing, selection and iteration. Variables of the form (x **of processor** i) will be taken as a reference to variable x of processor i. An unmodified variable x will be taken to mean (x **of processor** PID), that is, a local variable. For example, execution of the statement

```
if $y <$ ($y$ of processor $\lfloor PID/2 \rfloor$)
    then statement$_1$
    else statement$_2$
```

by a network of $P(n)$ processors causes the i^{th} processor, $0 \le i < P(n)$, to simultaneously compare its variable y with variable y of processor $\lfloor i/2 \rfloor$. If it finds that the former is less than the latter, then it executes statement$_1$, otherwise it executes statement$_2$. To aid synchronization, we assume that statement$_1$ and statement$_2$ are translated into blocks of code containing the same number of instructions, by filling with NO-OPs (such as $r_0 \leftarrow r_0$) as necessary. All of the algorithms in this thesis will maintain synchronization by virtue of this simple arrangement. As a notational convenience we may occasionally use multiple, concurrent and conditional assignments.

2.2. The Unit-Cost Measure of Time

In section 2.1 we defined the running-time of our parallel machines to be the number of instructions executed (synchronously) before all active processors have halted. That is, we charge a single unit of time for each instruction executed. This is termed a *unit-cost* measure of time. The use of unit-cost charging is a contentious issue. The alternative is *log-cost* charging, whereby the cost of an instruction is expressed as a function of the size of its arguments, thus tying the time required for a particular computation to its word-size.

We follow Cook [14] in the belief that the major parallel resources of interest are *time* and *hardware*. We also believe that the important issues in the design of a parallel machine are more clear-cut if these two resources are kept completely independent. A hardware measure should take into account the amount of memory used, which is related to word-size. This makes the unit-cost measure of time more attractive, since it alone is independent of word-size, and thus hardware.

Even for purely sequential machines, the selection of unit-cost measures versus log-cost is of fundamental importance. Inter-simulations between various log-cost models (for example [1], Turing machines and log-cost RAMs) can be achieved with only a polynomial increase in time, whereas no such simulation can be obtained between unit-cost and log-cost models. For example, in time t a unit-cost RAM with multiplication can compute (without input) a value as large as $2^{2^{t+\Theta(1)}}$, whereas the same machine with log-cost charging can only compute a value as large as $2^{t+\Theta(1)}$.

From a purely practical standpoint, the choice of charging mechanism depends on the type of computation in question. If the word-size is sufficiently small, then the unit-cost measure is more applicable. Alternatively, if the values being manipulated grow very quickly with input-size, requiring the use of multi-word instructions for quite modest input lengths, then the log-cost measure is preferable. For example, log-cost would more accurately model a small microprocessor, and unit-cost a large mainframe.

This issue is neatly encapsulated in what Goldschlager and Lister [29] call the "sequential computation thesis". This states that time on all "reasonable" sequential models is polynomially related. This is motivated principally by the polynomial-time simulations of one log-cost model by another, but in fact breaks the models into two disjoint classes, those with unit-cost and those with log-cost measure of time. Members of the same class are polynomially related, but two models from different classes are not. Given this observation, the important question which must be addressed by any theoretical treatment is not "which model is better", but "which model is more accurate for the intended application".

measure of time more attractive, since it alone is independent of word-size, and thus hardware.

Even for purely sequential machines, the selection of unit-cost measures versus log-cost is of fundamental importance. Inter-simulations between various log-cost models (for example [1], Turing machines and log-cost RAMs) can be achieved with only a polynomial increase in time, whereas no such simulation can be obtained between unit-cost and log-cost models. For example, in time t a unit-cost RAM with multiplication can compute (without input) a value as large as $2^{2^{t+\Theta(1)}}$, whereas the same machine with log-cost charging can only compute a value as large as $2^{t+\Theta(1)}$.

From a purely practical standpoint, the choice of charging mechanism depends on the type of computation in question. If the word-size is sufficiently small, then the unit-cost measure is more applicable. Alternatively, if the values being manipulated grow very quickly with input-size, requiring the use of multi-word instructions for quite modest input lengths, then the log-cost measure is preferable. For example, log-cost would more accurately model a small microprocessor, and unit-cost a large mainframe.

This issue is neatly encapsulated in what Goldschlager and Lister [29] call the "sequential computation thesis". This states that time on all "reasonable" sequential models is polynomially related. This is motivated principally by the polynomial-time simulations of one log-cost model by another, but in fact breaks the models into two disjoint classes, those with unit-cost and those with log-cost measure of time. Members of the same class are polynomially related, but two models from different classes are not. Given this observation, the important question which must be addressed by any theoretical treatment is not "which model is better", but "which model is more accurate for the intended application".

The parallel analogue of the sequential computation thesis is the so-called "parallel computation thesis" [9,27]. This states that time on all "reasonable" parallel models is polynomially related. Furthermore, it attempts to characterize parallel computers by relating parallel time to a sequential resource. More precisely, it states that time on a "reasonable" parallel computer is polynomially equivalent to log-cost sequential (for example, Turing machine) space. This has two implications. Firstly, a machine which is too weak to simulate an $S(n)$ space-bounded Turing machine in time $S(n)^{O(1)}$ is *not powerful enough* to be called a parallel machine. Secondly, a machine which is so strong that a $T(n)$ time-bounded computation cannot be simulated in space $T(n)^{O(1)}$ by a Turing machine is *too powerful* to be called parallel. We will be concentrating mainly on the latter aspect of the parallel computation thesis, since networks with an unrestricted instruction-set are obviously extremely powerful. Henceforth, by "reasonable" we will mean "not too powerful", in the sense that it is "reasonable" to expect a parallel computer to have only a moderate amount of resources at its disposal.

One way of making our model obey the parallel computation thesis is to restrict the processors to the minimal instruction-set of section 2.1 (this approach was taken by Goldschlager for his SIMDAG [27]). This ensures that the word-size grows by at most one in every time-step, and so the log-cost of the individual instructions executed in any given computation is at most a polynomial in the unit-cost running-time, provided the input integers are sufficiently small. In this case, unit-cost and log-cost are polynomially related. It makes sense to restrict the word-size of parallel processors since (as we saw in the second paragraph of this section) the extra power of unit-cost RAMs over log-cost RAMs seems to stem from their ability to generate large integers quickly. Indeed, a single unit-cost RAM with either the restricted [54] or full arithmetic [30] instruction-sets obeys the parallel computation thesis, so is itself as powerful as a parallel machine.

We claim that the unit-cost measure of time is a valid one for parallel processors. We shall call this the *unit-cost hypothesis*. It is framed as a hypothesis because it depends upon the way in which the word "valid" is to be interpreted; we will meet several interpretations in the remainder of this work. Whilst it is intuitively obvious that the unit-cost measure of time is unrealistic for very powerful instruction-sets which allow the computation of infeasible functions in a single step, we may reasonably expect it to be realistic for fairly weak instruction-sets, such as the minimal instruction-set of section 2.1.

This raises a number of interesting side-issues. We are in effect asking when a unit-cost model is "reasonable". We have seen that restricting the processors to the minimal instruction-set makes our model "reasonable" in the sense that it obeys the parallel computation thesis. But what do we actually mean by "reasonable"? Do models which satisfy the parallel computation thesis successfully formalize the idea of "parallel computers"? What do we really expect from a parallel machine model? These are amongst the issues that we will address in chapter 3.

2.3. The Assignment of Programs to Processors

Although every processor of our parallel machine executes the same program, our model does not fall precisely into the SIMD category of Flynn [19]. This is because the conditional goto instruction takes action depending on the value of a local register, the contents of which may vary from processor to processor. Thus different processors may be at different points in the program at any given time. However, it is fairly easy to show that our model is equal in power to a SIMD one, and to a reasonable subset of MIMD models, including that of Galil and Paul [21].

For the sake of discussion, we will call our assignment of programs to processors a *uniform* one. We use the term "uniform" in the sense of Karp and Lipton [36], meaning that every machine has a finite description (in our case, the program and processor bound). A MIMD model is non-uniform in the sense that it allows a different program for each processor; thus an infinite family of finite descriptions (one for each input size) is needed. Some authors (for example [7,14,59]) use the term "uniform" to denote the fact that an external "constructibility" condition has been enforced on a non-uniform model in order to restrict interest to machines with finite descriptions.

A SIMD machine is a uniform one in which, at any given point in time, all active processors are either executing the same instruction, or are dormant. Our high-level pseudo-programming language allows the user to write non-SIMD programs; we believe that this keeps the language simple, elegant and flexible (it may be argued that it gives the user the flexibility to get into a lot of trouble, but the same is often said of the goto statement in modern programming languages). Furthermore, it is not really necessary to force the programmer to write SIMD programs, since a uniform machine can be simulated by a SIMD one without asymptotic time-loss, using the same number of processors and degree, with space and word-size increasing by only a constant.

Suppose M is a P(n)-processor uniform machine. We will construct a SIMD machine to simulate M as follows. Processor i of the SIMD machine, $0 \le i < P(n)$, simulates processor i of M, using variables PC, VPC, NPC, PR, A and V, and an infinite array R. PC keeps track of the program counter of the simulated processor, whilst for $j \ge 0$, R[j] contains the current contents of its register r_j. VPC (the virtual program counter) will cycle from 1 to the program length (which is a constant, independent of n); when PC = VPC the PCth instruction of the program of M is simulated. NPC receives the new program counter value, and if the instruction involves a data-transfer, A and PR receive the address and PID respectively of the register to be updated, and V its new value. At the end of the cycle, the arrays R are updated to reflect the new register contents, using the information in PR, A and V, whilst PC is updated using the contents of NPC. The process is completed at the end of a cycle in which a halt instruction is simulated.

We present the algorithm in the high-level language of section 2.1. A different interpretation is placed on the control constructs however, in order to make them SIMD. The branches of a selection statement (such as **if** or **case** are tried one at a time, with a processor executing a particular branch if its register contents satisfy the entry condition; all other processors remain dormant during that period. This is opposed to the general (non-SIMD) uniform case, in which all processors are free to start their respective branches at the same time, or to enter and leave the construct at different times.

Suppose M has the example instruction-set of section 2.1. Then the program of the simulating machine is as follows:

```
A:=V:=0
PC:=VPC:=1
**while** PC > 0 **do**
  **for** VPC:=1 **to** program length **do**
    **if** VPC = PC **then**
      PR,A,V:= **case** PC^th instruction of M **of**
                 "$r_i \leftarrow$constant":PID,i,constant
                 "$r_i \leftarrow r_j \sim r_k$":PID,i,R[j]~R[k]
                 "$r_i \leftarrow r_{r_j}$":PID,i,R[R[j]]
                 "$r_{r_i} \leftarrow r_j$":PID,R[i],R[j]
                 "$r_i \leftarrow$PID":PID,i,PID
                 "$r_i \leftarrow r_{r_j}$ **of** $r_k$":PID,i,(R[R[j]] **of processor** R[k])
                 "($r_{r_i}$ **of** $r_j$)$\leftarrow r_k$":R[j],R[i],R[k]
      NPC:= **case** PC^th instruction of M **of**
               "halt":0
               "**goto** m **if** $r_i > 0$":**if** R[i] > 0 **then** m
               "others":PC+1
  (R[A] **of processor** PR):=V
  PC:=NPC
```

Thus we see that our uniform model is equivalent to a SIMD one. Now, a MIMD model allows a different program for each processor. Let $\Delta : N \to N$ be such that $\Delta(i)$ is a reasonable encoding of a RAM program (say, using the example instruction-set of section 2.1), for $i \geq 0$. By "reasonable encoding" we mean that a universal RAM should be able to decode this program, using negligible resources, into a format which allows efficient simulation. A MIMD variant of our model is identical to that of section 2.1, except that processor i of a P(n)-processor network has program $\Delta(i)$, $0 \leq i < P(n)$. Δ is called the *processor assignment* function.

Let M be a P(n)-processor MIMD machine which uses resources $R_1(n)$, whose processor assignment Δ is such that $\Delta^* = \langle \Delta(0), \Delta(1), \ldots, \Delta(P(n)-1) \rangle$ can be computed by a P(n)-processor uniform parallel machine using resources $R_2(n)$.

Then clearly there is a uniform P(n)-processor machine which can simulate M in resources $R_1(n)+R_2(n)$, simply by computing Δ^*, and then having processor i, $0 \le i < P(n)$ simulate program $\Delta(i)$. Each processor of the uniform machine has an identical program made up of two parts, a part to compute Δ^*, and a universal RAM.

Thus we see that (provided the resources needed to compute Δ are kept to a feasible level) a uniform machine can efficiently simulate a MIMD one. This can be summarized as follows: if a MIMD machine is easy to specify, then it can be specified as a uniform machine. Thus a uniform model is equivalent to a useable subset of the MIMD model.

2.4. Processor Activation

In our model as presented so far, all P(n) processors are activated simultaneously at the start of the computation, and begin synchronously executing the first instruction of the program at time t=1. We call this the *initial activation* model. An alternative formulation (*lazy activation*) is to start off with some small number of active processors (for example, just processor 0, or just those which receive input), postponing the activation of the remainder until run-time. This convention has been adopted by Galil and Paul [21] and Savitch [60].

There are two essentially different ways of approaching lazy activation. The first requires that an active processor explicitly activate an inactive one by executing a special "call" instruction (as in, for example, [60]). This implies that the number of active processors can at most double in each time-step. Alternatively, Galil and Paul [21] allow the inactive processors to execute a polling-loop, in order to decide when to become active. This is really only feasible for machines with constant degree, in which case it is asymptotically equivalent to the first approach.

Note that this implies that a T(n) time-bounded machine can have at most $n.2^{O(T(n))}$ (in the case when only input-bearing processors are initially active), or $2^{O(T(n))}$ (in the case when processor 0 is initially active) processors. Our model is more general than this, and we shall see in section 3.3 that it *does* make good sense to talk about T(n) time-bounded machines with $2^{T(n)^{O(1)}}$ processors.

For definiteness, we will assume that:

(1) Initially, only processor 0 is activated.

(2) In a computation on an input of size n, processor 0 is initially given the value of P(n) as part of its input.

(3) If an inactive processor has a value written into it for the first time in a computation during time-step t, it becomes active and executes the first instruction of the program during time-step t+1. Thereafter, it is indistinguishable from any other active processor. A processor which has halted cannot be reactivated.

Lazy activation is essentially the same as initial activation, provided $P(n) \leq 2^{O(T(n))}$. Clearly an initial-activation machine can simulate a lazy-activation one without asymptotic loss in resources, by simply maintaining an activation flag in each processor. Simulation in the other direction is only slightly more difficult. The problem is to activate P(n) processors and synchronize them so that they begin the execution of the program of M at the same time. If M has P(n) processors and runs in time T(n), we will show that the simulating (lazy) machine runs in time O(T(n)+log P(n)), whilst increasing space and degree by only a constant amount.

To simplify the presentation, assume that P(n) is of the form 2^k-1 for some $k>0$. We will activate P(n) processors using an interconnection pattern in the shape of a binary tree (see figure 2.4.1).

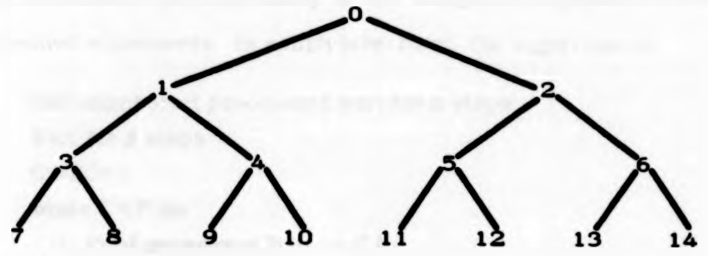


Figure 2.4.1 The binary tree interconnection pattern with 15 vertices.

Each processor has variables C and P. P holds the value of P(n) (we assume that P of processor 0 is set to P(n) at the start of the computation), and C the number of processors activated so far (we assume that C of processor 0 starts out at 0). The algorithm consists of a single loop. At each iteration, a new level of the tree is activated; C is used to detect termination. Upon exiting the loop, all processors are synchronized, and execution of the program of M can begin.

Processor i activates its children at the next level (processors 2i+1 and 2i+2) using the high-level statements

```
(C,P) of processor 2i+1 := C,P
(C,P) of processor 2i+2 := C,P
```

This also initializes their variables C, P so that they can join in the loop at the appropriate stage. Note that the left-hand child is activated before the right-hand, and so potentially enters the loop earlier. We can avoid this by making odd-numbered processors wait for α steps (where α is a suitably chosen constant) before entering the loop. In order to synchronize the newly-activated processors entering the loop with those already inside it, it is necessary to add another delay, this time of β steps (where β is another suitably chosen constant). Note that the values α, β depend only on the exact form of the RAM instruction-set, and the ability of the compiler to generate succinct code from high-level statements. In a high-level form, the algorithm is:

```
Odd-numbered processors wait for α steps
Wait for β steps
C:=2C+1
while C<P do
   (C,P) of processor 2i+1 := C,P
   (C,P) of processor 2i+2 := C,P
   C:=2C+1
```

To make the synchronization method completely transparent, this program would generate the following code (using an instruction-set similar to that of section 2.1, with certain acceptable liberties taken with arithmetic and Boolean expressions in order to ensure brevity). In this case, $\alpha = 2$ and $\beta = 1$.

```
1.  goto 4 if PID mod 2 = 0
2.  NOOP
3.  NOOP
4.  NOOP
5.  C←2*C+1
6.  goto 13 if C≥P
7.  (C of 2*PID+1)←C
8.  (P of 2*PID+1)←P
9.  (C of 2*PID+2)←C
10. (P of 2*PID+2)←P
11. C←2*C+1
12. goto 6
13. etc.
```

Table 2.4.1 gives a trace of this algorithm for P(n) = 7 processors.

Time	Processors																				
	0			1			2			3			4			5			6		
	PC	P	C	PC	P	C	PC	P	C	PC	P	C	PC	P	C	PC	P	C	PC	P	C
0	1	7	0																		
1	4	7	0																		
2	5	7	0																		
3	6	7	1																		
4	7	7	1																		
5	8	7	1	1		1															
6	9	7	1	2	7	1															
7	10	7	1	3	7	1	1		1												
8	11	7	1	4	7	1	4	7	1												
9	12	7	3	5	7	1	5	7	1												
10	6	7	3	6	7	3	6	7	3												
11	7	7	3	7	7	3	7	7	3												
12	8	7	3	8	7	3	8	7	3	1		3				1		3			
13	9	7	3	9	7	3	9	7	3	2	7	3				2	7	3			
14	10	7	3	10	7	3	10	7	3	3	7	3	1		3	3	7	3	1		3
15	11	7	3	11	7	3	11	7	3	4	7	3	4	7	3	4	7	3	4	7	3
16	12	7	7	12	7	7	12	7	7	5	7	3	5	7	3	5	7	3	5	7	3
17	6	7	7	6	7	7	6	7	7	6	7	7	6	7	7	6	7	7	6	7	7
18	13	7	7	13	7	7	13	7	7	13	7	7	13	7	7	13	7	7	13	7	7

Table 2.4.1 Activation and synchronization of 7 processors in a lazy-activation model. Table entry shows the value of the program-counter (PC) and variables P,C for each processor initially (at time 0), and after each of 18 steps.

Chapter 3
Relationships with Other Models

The main aim of this chapter is to compare our network machines to a number of other models. In the first section, we propose a fixed-structure variant of the network model, that is, one in which the interconnection patterns of the machines can be predicted. The more general model computes its own interconnections, which makes it rather difficult to construct as a physical device without increase in degree. It is observed that more efficient machines can be constructed by expending more resources, for example a machine with a non-recursive interconnection function can compute arbitrary (non-recursive) single-valued Boolean functions in constant time, given sufficiently many processors.

The second section compares our model to a shared-memory one. In a shared-memory machine, the processors communicate indirectly via a common shared memory, rather than by direct register access. The two models are easily seen to be almost identical in computing power. The third section investigates the concept of a "reasonable" parallel machine touching on such issues as the parallel computation thesis, bounds on word-size, and restrictions on inter-processor communication.

In the fourth section we define a practical variant of the network machine, the so-called "feasible network". We expound the desirability of a feasible network which is universal for the more general model of section 2.1. Various types of universal machines are considered, according to the manner in which they achieve their simulations. In the fifth and final section, we investigate possible speedups of sequential machines by parallel ones. Any computable function can be computed in constant time if sufficiently many processors are present. Alternatively, an arbitrary polynomial speedup in time can be obtained on a machine which obeys the parallel computation thesis (and, as we shall see in section 3.3, no such exponential speedup is likely).

3.1. A Fixed-Structure Model

The basic model described in section 2.1 falls into Cook's [14] category of machines with "modifiable structure" (since processor interconnections are computed at run-time). This implies that a resource-bound for such a machine is made up of two parts, corresponding to the resources required to compute the interconnections and those required to perform the actual computation. In a "fixed-structure" machine these two components are separated. The former reflects the cost of building the machine, and the latter the cost of using it.

This separation can become significant when the two components differ by a large amount. For example, consider a machine with the example instruction-set of section 2.1, whose only allowable binary operations are (single-bit) two-input Boolean functions, integer division by 2 and multiplication by 2. Let $f:\{0,1\}^*\to\{0,1\}^*$ be defined by $f(x_0, \ldots, x_{n-1}) = \langle y_0, \ldots, y_{n-1} \rangle$ where for $0 \le i < n$, $y_i = x_i \oplus x_{(i+1) \bmod n}$. An n-processor, constant-degree machine can compute f in a constant number of steps, provided processor i knows the value of $(i+1) \bmod n$, $0 \le i < n$. However, the same machine requires $\Omega(\log n)$ steps to actually compute those values. Thus in a model with modifiable structure, the run-time of this machine is $\Omega(\log n)$, under a fixed-structure model the run-time is $O(1)$ (and any reasonable fabrication device which includes addition as part of its instruction-set can compute the interconnections in parallel in a constant amount of time).

A fixed-structure analogue of our basic model can be defined as follows. Galil and Paul [21] call this a model with "predictable communication", since the inter-processor connections need to be known in advance of actually running the machine. Note that all machines have "predictable communication" in the sense that they can be fabricated as a completely-connected machine (with each processor connected to every other), but this may involve an unacceptable increase in degree.

Our fixed-structure model has the same format as the basic model of section 2.1, with a number of minor modifications. Each processor is given a number of additional read-only registers which are preset at the beginning of a computation. These correspond to values which are "hard-wired" into the machine during the fabrication process. They consist of the DEGREE register, and an infinite number of port registers $p_0, p_1, \ldots$, each of which is capable of holding a single integer.

More formally, a parallel machine consists of a program and an interconnection scheme. The *program* is a finite list of instructions, each of which may have the following form (where p is a port register). Either:

(1) Read a value from a register of processor p.

(2) Write a value to a register of processor p.

(3) Perform an internal computation.

(4) Conditional transfer of control, or halt.

In the example instruction-set of section 2.1, the read instruction:

$$r_i \leftarrow (r_{r_j} \text{ of } r_k)$$

would be interpreted as meaning "read the (r_j)th register of processor p_{r_k} and place the result into register r_i", and the write instruction:

$$(r_{r_i} \text{ of } r_j) \leftarrow r_k$$

would be interpreted as meaning "write the value from register r_k into the (r_i)th register of processor p_{r_j}".

An ***interconnection scheme*** consists of three functions, a processor function $P: N \rightarrow N$, a degree function $D: N \rightarrow N$ and an interconnection function

$$G:\{i \mid 0 \le i < P(n)\} \times \{d \mid 0 \le d < D(n)\} \times N \to \{i \mid 0 \le i < P(n)\}.$$

In a $P(n)$-processor computation, processor i is connected to processors $G(i,d,n)$, $0 \le d < D(n)$. We adopt the convention that if $i \in \{G(j,d,n) \mid 0 \le d < D(n)\}$ then $j \in \{G(i,d,n) \mid 0 \le d < D(n)\}$. A *computation* of M, where M has interconnection scheme (P,D,G), is defined similarly to section 2.1, with the following addition. Before the processors are activated, the DEGREE register is set to $D(n)$, and for $0 \le d < D(n)$, $0 \le i < P(n)$, register p_d of processor i is set to $G(i,d,n)$. The resources of space and word-size are modified to include the new registers (the word-size of the port registers may be measured according to their absolute contents, or some concise relative encoding, if such is applicable).

Note that a resource bound for computing any given function must include reference to the complexity of the interconnection scheme. This is because, as might be expected, more efficient machines can be built by investing more resources in their construction. Information can be stored in the interconnection pattern, to be used later as a kind of "look-up table". Take for example the problem of computing an arbitrary (perhaps non-recursive) single-valued Boolean function $f:\{0,1\}^* \to \{0,1\}$. We will show how to compute f on n inputs in a constant number of steps, using $n.2^n$ processors.

If $x = \langle x_0, \ldots, x_{n-1} \rangle$ is an input of size n, let $\text{int}(x) = \sum_{i=0}^{n-1} x_i.2^i$ be a binary encoding of x as an integer. The $n.2^n$ processors are broken up into 2^n *teams* T_i, $0 \le i < 2^n$. For $0 \le i < 2^n$, team T_i consists of the n processors $i.n+j$, for $0 \le j < n$. The smallest-numbered processor of each team is a distinguished processor called the *team-leader*. For each input x, the team-leader of $T_{\text{int}(x)}$ will have the value $f(x)$ encoded as part of its interconnection pattern. Our problem then, given an input x, is to notify the appropriate team-leader.

This is achieved as follows. Each team-leader sets a specified register a to zero. For $0 \le i < 2^n$, $0 \le j < n$ the jth member of team T_i compares the jth symbol of the input to the j^{th} bit of i. If these two values are different, it writes a one to register a of its team-leader. The team-leader of $T_{int(x)}$ will be the only team-leader which is not written to; it then consults its interconnection pattern for the value of $f(x)$, and writes this value to processor 0 for subsequent output.

The following is a high-level implementation of this algorithm. Assume that initially variable x of processor p contains the p^{th} bit (x_p) of the input, $0 \le p < n$. Each processor has two variables i and j which (as in the previous paragraph) record that processor's team number and position within that team. Variable a of the team-leaders will be used for communication with its team members. The result $f(x)$ will end up in variable r of processor 0.

The interconnection pattern is as follows. For $0 \le i < 2^n$, $0 \le j < n$, processor $n.i+j$ (the j^{th} member of T_i) is connected to processor j (the processor in charge of the j^{th} bit of the input), and processor $n.i$ (its team-leader). For each input x, processor $n.int(x)$ is connected to processor $f(x)$ via a special link. It can determine the value of $f(x)$ by reading the PID of that processor.

```
a:=r:=0
i,j:=$\lfloor PID/n \rfloor$,PID mod n
if (x of processor j) $\ne$ (j$^{th}$ bit of i)
   then (a of processor i*n):=1
if (j=0) and (a=0)
   then (r of processor 0) := PID of processor special link
```

Notes. (1) The algorithm as presented uses the extended arithmetic instruction-set. The restricted arithmetic instruction-set can be substituted by increasing the number of processors in each team to 2^n. If the minimal instruction-set is used, the run-time is O(log n). Note that the values $i.n$, $0 \le i < 2^n$, are not computed at run-time, but are stored as part of the interconnection pattern.

(2) The degree can be reduced to a constant by the use of binary trees for

routing. Information about f(x) is encoded using the technique of theorem 4 of Galil and Paul [21]. The run-time is increased to O(log n) on either the full arithmetic, restricted arithmetic or minimal instruction-sets.

(3) The number of processors can be reduced to 2^{n+2} [21]. This increases the run-time to O(n), although it does reduce the degree to a constant, and uses only finite-state machines as processors.

3.2. Shared Memory Machines

A popular alternative model is obtained by constraining processors to communicate via a common memory, rather than communicating by direct processor-to-processor links. Let $D,P,S,T,W,Z:N \to N$.

A *shared memory machine* consists of an infinite number of processors attached to a globally accessible shared memory. Each processor possesses an infinite number of general purpose registers, and a unique read-only processor identity register PID which is preset to i in the i^{th} processor, $i \in N$. A *program* for this machine consists of a finite list of instructions; each instruction is of the form either:

(i) Read a value from a specified place in the global memory.

(ii) Write a value to a specified place in the global memory.

(iii) Perform an internal computation.

(iv) Conditional transfer of control, halt.

The allowable internal computations usually consist of direct and indirect register transfers, logical and arithmetic operations.

More formally, each machine is specified by a program *P* and a processor bound P(n). A computation proceeds roughly as follows. An input of size n (where the "size" measure depends on the problem in question) is broken up into n unit-size pieces, and the i^{th} piece is stored in global memory location i, $0 \leq i < n$. All other memory locations and general purpose registers are set to zero. Processors 0,1,...,P(n)-1 are activated simultaneously; they synchronously execute the program *P*. When all processors have halted, the output is to be found in some specified place in the global memory.

The *processor* bound P(n) is a measure of the number of processors used as a function of input size. The *space* S(n) is the maximum number of non-zero entries in the global memory and registers at any time during the computation. (Note that this includes the input and the processor identity registers). The machine is said to have *word-size* W(n) if the maximum value in any register or global memory location during any computation on an input of size n has absolute value less than $2^{W(n)}$. The *time* bound T(n) is the number of instructions executed before all processors have halted, again as a function of input size.

Variants of this model have appeared, for example, in [8,14,20,21,27,42,47,62,64,68,69,71,72]. We assume some reasonable protocol for dealing with memory access conflicts, as in those references. The general consensus of opinion is that whilst the shared memory model is a powerful theoretical tool, it is not feasibly buildable using any foreseeable technology.

A shared-memory machine M can be simulated by a network with identical internal instruction-set, without asymptotic loss of resources. Suppose M has P(n) processors. Then the network has P(n)+1 processors. Processor 0 remains idle throughout the computation, whilst processor i, $1 \leq i \leq P(n)$ simulates the action of processor i-1 of M. A reference to global memory location m is replaced by a reference to register r_m of processor 0. The extra processor can be eliminated by having processor 0 reserve the odd-numbered registers for its own use, and the even-numbered registers for the contents of the shared memory. A reference to global memory location m is then replaced by a reference to register r_{2m} of processor 0.

Alternatively, the global memory contents can be divided up amongst the processors of the network, provided the instruction-set is sufficiently powerful. Suppose M has P(n) processors and space S(n). Processor i of the network, $0 \leq i < P(n)$, simulates processor i of M, and in addition holds the values of global memory locations $i+j.P(n)$, $j \geq 0$. A reference to global memory location m is replaced by a reference to register $r_{2.\lfloor m/P(n) \rfloor}$ of processor $m \bmod P(n)$ (each processor can reserve its even-numbered registers for memory locations, and the odd-numbered registers for its own use).

This assumes that the instruction-set is at least as powerful as the full arithmetic instruction-set of section 2.1. If the restricted arithmetic instruction-set is used, P(n) should be replaced by $2^{\lceil \log P(n) \rceil}$. For a minimal instruction-set, the time-loss is O(log P(n)) per instruction, using P(n) processors. If sufficiently many processors are used (so that each processor holds at most one memory location) this time-loss can be reduced to a constant multiple.

Similarly, a network M can be simulated by a shared-memory machine without asymptotic loss in resources, provided the instruction-set is sufficiently powerful. The registers of the network are stored in the common memory - each processor of the shared-memory machine need only have a constant number of local registers (note that a similar trick serves to reduce the local-memory requirements of all shared-memory machines, subject to similar conditions). A reference to register r_i of processor j is replaced by a reference to global memory location $P(n).i+j$.

This replacement costs only a constant number of steps per access for machines with the full arithmetic instruction-set. As before, if P(n) is replaced by $2^{\lceil \log P(n) \rceil}$ it also costs a constant number of steps with the restricted arithmetic instruction-set. For machines with the minimal instruction-set, a similar result can be obtained by storing, along with each register r_i, the contents of r_i multiplied by P(n). This requires time proportional to $\left\lceil \frac{n}{P(n)} \right\rceil$.log P(n) to set up for the initial input string. Thereafter, these values can be maintained and used for register access with a constant loss in time for each step of M. Alternatively, the multiplication by P(n) can be computed at access-time, at a cost of O(log P(n)) per access.

3.3. Reasonableness and Practicality

In section 2.2 we raised the following important question: what constitutes a "reasonable" model of parallel computation? In particular, what is a reasonable instruction-set for our processors, given that we have chosen a unit-cost measure of time? Goldschlager, in [27], placed certain restrictions on his SIMDAG's (a variant of the shared-memory model considered in section 3.2) to ensure that they obey the ***parallel computation thesis***: time on any "reasonable" parallel model is polynomially equivalent to sequential (log-cost) space. Evidence for this thesis is provided by a multiplicity of "reasonable" models, for example, alternating Turing machines [9], uniform circuits [7] and vector machines [54], as well as Goldschlager's SIMDAG and conglomerate.

As we shall see later in this section, in order to make networks and shared-memory machines obey the parallel computation thesis, it is necessary to place upper bounds on the word-size and type of instructions allowed. These restrictions can be accepted as "reasonable" purely on practical grounds - for example, one can argue that the word-size of problems tackled in practice should not grow too rapidly with input-size. In this sense, "reasonable" can be equated to "practical".

The parallel computation thesis also provides us with a powerful theoretical tool. Suppose that we are interested in those problems from P which have an exponential speedup in parallel, that is, those members of P which can be solved in time $\log^{O(1)}n$ by a "reasonable" parallel machine. If a "reasonable" machine is one which obeys the parallel computation thesis, then these are precisely the members of P which can be solved in polylog space by a Turing machine.

Let POLYLOGSPACE denote the class of languages which can be accepted in space $\log^{O(1)}n$ by a Turing machine. It is widely conjectured that $P \not\subseteq$ POLYLOGSPACE (although it is not known for sure whether either class contains the other). Evidence is provided for this conjecture by the existence of *log space complete* problems (see, for example, [24,25,26,28,34,35,37]); that is, problems which are members of P, yet if any one of them is a member of POLYLOGSPACE then $P \subseteq$ POLYLOGSPACE. Thus log-space complete problems probably do not have an exponential speedup on any "reasonable" parallel machine, where the parallel computation thesis is used as a criterion for "reasonableness".

Thus we see that there are two facets to the concept of "reasonableness", that which is reasonable from a practical point of view, and that which is reasonable from the theoretical point of view. It may be theoretically interesting to consider networks with an exponential number of processors (as in section 3.4), but it is certainly not reasonable to consider them as a practical proposition for all except the smallest values of n. A theoretical model is an attempt to capture the essence of an intuitive notion of "parallel computation"; a practical model is, in addition, governed by physical and technological constraints.

The remainder of this section is devoted to a closer look at some ways of defining a "reasonable" model. Earlier in this section we referred to some additional conditions which ensure that networks obey the parallel computation thesis. What exactly are these conditions? Firstly, an $S(n)$ space-bounded nondeterministic Turing machine can be simulated by a network with the minimal instruction-set, in time and word-size $O(S(n))$, using the techniques of theorem 2.1 of Goldschlager [27]. Conversely, we have:

Theorem 3.3.1 A $T(n)$ time-bounded network M with word-size $W(n)$ can be simulated by a deterministic Turing machine using space $T(n).W(n)+S(n)$, where $S(n)$ is the space required for the Turing machine to simulate a single instruction of a processor of M.

Proof. Similar to theorem 2.2 of [27]. □

This enables us to throw some light on the unit-cost hypothesis. As far as the parallel computation thesis is concerned, it is reasonable to charge a single unit of time for instructions which can be computed by a Turing machine in space $T(n)^{O(1)}$, where $T(n)$ is the number of steps in the intended computation. Given this condition, networks obey the parallel computation thesis provided $W(n) = T(n)^{O(1)}$. Note that this allows machines with as many as $2^{T(n)^{O(1)}}$ processors; although those who support lazy activation (see section 2.4) insist that $P(n) = 2^{O(T(n))}$, and some authors insist that $P(n) = n^{O(1)}$ (for example, [16,17,42,53]).

To summarize, here are a number of restrictions on network and shared-memory models which can be used to define so-called "reasonable" machines.

(1) *Restrictions on the instruction-set.*

Restrictions on instruction-set are motivated by a desire to see that the unit-cost hypothesis holds.

(a) The first premise is that individual processors should behave like log-cost sequential machines. In particular, the resource of time should be polynomially related to time on an accepted log-cost sequential machine model, such as the deterministic Turing machine (c.f. the sequential computation thesis, section 2.2). Thus instructions which are valid for a T(n) unit-cost-time bounded computation should individually take no more than $T(n)^{O(1)}$ steps on a deterministic Turing machine.

(b) Instructions should be computable in space $T(n)^{O(1)}$ by a Turing machine. This helps to ensure that the parallel computation thesis holds. Note that this condition is implied by part (a) above.

(2) *Bounds upon processors and time.*

Upper-bounds on the number of processors are usually motivated by the observation that, given enough processors, every computable function can be computed in constant time (see section 3.5), which makes time a singularly uninteresting resource.

(a) $P(n) \leq 2^{O(T(n))}$. This is a consequence of the lazy-activation approach (see section 2.4).

(b) $P(n) = n^{O(1)}$, $T(n) = \log^{O(1)} n$. Machines with these two properties are sometimes called *small* and *fast* respectively. See, for example, [16,17,53,59].

(3) *Bounds upon word-size.*

Upper-bounds on word-size are usually motivated by the observation that (as previously noted in section 2.2) single-processor machines with the full [30] or restricted [54] arithmetic instruction-sets obey the parallel computation thesis, and so can be considered "reasonable" parallel machines in themselves. This makes the processor-bound an uninteresting resource.

(a) $W(n) = O(T(n))$. This can be achieved indirectly (as in Goldschlager [27]) by restricting the instruction-set and the size of the input-symbols.

(b) $W(n) = T(n)^{O(1)}$. This condition guarantees that the parallel computation thesis holds, subject to the additional conditions on the instruction-set mentioned in 1(b) above.

(c) $W(n) = n^{O(1)}$. This ensures that the input encoding is "concise" in the sense of [22]. If the input symbols are allowed to be integers with more than a polynomial number of bits, then n is no longer a reliable (to within a polynomial) measure of input-size.

Other restrictions are often made in the literature, motivated, it is often claimed, by practical considerations. These include the following:

(1) Restrictions on degree. It is widely accepted that a completely-connected machine is impractical. Some authors (for example, Galil and Paul [21]) think that degree should be constant (i.e. independent of input-size).

(2) Restrictions on the interconnection pattern. In the case of fixed-structure networks (see section 3.1) it is desirable to restrict oneself to machines with an interconnection pattern which is in a sense easy to compute (see, for example, [21]). This is also the case for uniform circuits [59] and conglomerates [27]. One advantage of this approach is that it avoids the kind of machine described in section 3.1, which can compute a large class of functions (which may even be non-recursive) in an unnaturally small amount of time.

(3) Restrictions on register access. Even if higher-degree machines are acceptable, should every processor necessarily have the freedom of being able to read *any* register of its neighbours? An alternative is to provide each processor with a special communication register, which is the *only*

register accessible to other processors. This approach is taken, for example, in [21,36,68]. We will call machines of this kind ***restricted-access* networks**.

(4) Restrictions on multiple register access. Some authors (for example, [20]) insist that simultaneous writes to a single register be disallowed, others (for example, [42]) insist that simultaneous reads of a single register also be banned.

We will have more to say on these matters in later sections.

3.4. A Practical Model

In the last section we saw various constraints which can be placed on our network model in order to make it "reasonable" or "practical". We are now ready to define our own practical variant of the network model. A ***feasible network*** M is a fixed-structure network (see section 3.1) with interconnection scheme (P,D,G), such that:

(i) Each processor has a constant number of general-purpose registers.

(ii) The degree, D(n), is a constant.

(iii) The interconnection function G can be computed in time O(log P(n)) by a deterministic Turing machine.

These three conditions ensure that the machines are in a sense easy to construct. Each processor has a small amount of memory, and a small number of easy-to-compute interconnections. Machines with similar characteristics have appeared, for example, in [21,45,46,55,62,66,69]. Note that we have made no attempt to make the model "reasonable" by placing bounds on the number of processors, space, time, word-size, or the complexity of the instruction-set, according to the guidelines laid down in the last section. The reader is completely free to make whatever additional restrictions are required, according purely to taste, or in order to model a particular kind of computing environment.

Even if we accept the feasible network as being feasibly constructible, it is unlikely that the fabrication costs would be so low that the average user would be willing to build a new machine for each application. More likely, the user would prefer to present each new machine (in the form of a program) to a universal parallel computer which can simulate it at a small cost in resources. The user would thus be able to trade the fabrication cost of a feasible network for a small increase in resources at run-time.

A further advantage is to be gained if we can find an efficient feasible network which is universal for the general model of section 2.1. From a practical point of view, it would provide the user of a feasible network with a new, flexible high-level programming language. Programs which are written in a high level programming language similar to that of section 2.1 could (although they may correspond to machines which are not feasibly constructible) be run on a feasible universal machine, for a small extra cost in resources. By building a single feasible network the user gains the use of a flexible and elegant virtual architecture, corresponding to a completely-connected network. From a theoretical point of view, we obtain a practical motivation for studying the more esoteric parallel machine models of chapter 2.

Note that the universal machine is far more attractive than the machines that it can simulate. For example:

(1) It is a fixed-structure machine with a small number of easy-to-compute interconnections per processor.

(2) The number of registers per processor is small. The problem of whether to allow access to arbitrary registers of neighbouring processors thus vanishes; each processor can be restricted to communicating via a single communication register (as mentioned in section 3.3) without asymptotic time-loss.

(3) Because its degree is constant, the problem of whether to allow simultaneous access to those communication registers also vanishes. Accesses in the universal machine can be restricted to exclusive reads without asymptotic time-loss (by use of a polling loop).

(4) The requirement that the universal network is synchronized is no longer essential (see [21]).

Yet the machines being simulated need share none of these restrictions.

Exactly what do we mean by a "universal machine"? Suppose U is a P(n) processor (feasible) network, M is an arbitrary network, and $x = <x_0, \ldots, x_{n-1}>$ is an input of size n. A *simulation* of M on input x by U is to proceed as follows. Let $p=P(n)$. Place x_i into register $r_{\lfloor i/p \rfloor}$ of processor (i mod p). Place into the remaining registers of processor 0 a concise finite encoding of the program of M. Set all other general-purpose registers to zero, and simultaneously activate processors $0, 1, \ldots p-1$ on the program of U. Suppose M computes a function f, and $f_n(x) = <y_0, \ldots, y_{m-1}>$ (for definitions see section 2.1). U is said to be *universal* if for all machines M and inputs x, when all processors of U have halted, register $r_{\lfloor i/p \rfloor}$ of processor (i mod p) contains y_i, for $0 \leq i < m$.

We are interested in a particular kind of simulation, which we shall call "step-wise". A simulation of a T(n) time bounded network M on an input of size n is said to be *step-wise* if:

(1) For $0 \le i < S(n)$, $0 \le \tau \le T(n)$ each register i of M has a corresponding ***dedicated processor*** $d(i,\tau)$ in U. Note that we may allow $d(i,\tau) = d(j,\tau)$ when $i \ne j$.

(2) Suppose $t:N \to N$. The simulation consists of three phases:

(a) Initialization. This includes the assignment of, and routing of the input values to the dedicated processors, as well as any pre-computations required for phase (b).

(b) Computation. The computation phase is to take $t(n).T(n)$ steps. For $0 \le \tau \le T(n)$ we require that after $t(n).\tau$ steps of this phase, processor $d(i,\tau)$ has a distinguished register which contains the contents of register i of M after τ steps of M, $0 \le i < S(n)$.

(c) Termination. This includes routing of the output from the dedicated processors to the output processors.

Such a universal machine is said to have ***delay*** $t(n)$. The ***setup time*** is the time required for phases (a) and (c) combined. Note that the set-up time must be independent of $T(n)$. A step-wise simulation is also said to be ***literal*** if a data-transfer from registers i to register j of M, $0 \le i,j < S(n)$, during time-step τ, $1 \le \tau \le T(n)$, gives rise to a communication between processors $d(i,\tau-1)$ and $d(j,\tau)$ of U between time-steps $(\tau-1).t(n)+1$ and $\tau.t(n)$ of phase (b). More formally, define a directed multi-graph G_n as follows (G_n is to reflect the information flow between processors of U during the simulation of time-step τ of M). G_n has vertex-set $\{0,1,\ldots,P'(n)-1\}$ (where U has $P'(n)$ processors), and an edge from vertex u to vertex v, labelled δ, if during time step $(\tau-1).t(n)+\delta$ of phase (ii), processor v of U reads a value from processor u, $1 \le \delta \le t(n)$. (Recall that processors of U use only exclusive-reads for inter-processor communication). We require that there be a path from $d(i,\tau-1)$ to $d(j,\tau)$ in G_n with monotonic increasing labels on the edges. Thus in a literal simulation, a data transfer between registers of the simulated machine can give rise to a data transfer between the corresponding dedicated processors within the simulation of that time-step. In a *non-literal* simulation, the required data may (for example) have started out during the simulation of the previous step of M, and been kept up-to-date by auxiliary processors along the way (see sections 7.1, 7.2).

Later, we will consider a more restrictive form of literal simulation in which the dedicated processor assignment does not change with time. We will call this type of simulation *strongly literal*. In section 6.3 we give an upper-bound of O(log P(n)) on the delay for a strongly-literal simulation of a P(n) processor, constant-degree, restricted-access machine, and match this with a lower-bound in section 7.1.

3.5. Speedup of Sequential Machines

In section 3.3 we briefly touched on the following question: which problems in P have an exponential speedup in time on a "reasonable" parallel machine, where "reasonable" is defined using the parallel computation thesis? The only answer which is currently available is "probably not all of them". Here we tackle an easier question: what speedups *are* offered by our networks (reasonable or otherwise), as opposed to sequential ones. As a partial answer, we provide the following result.

Theorem 3.5.1 Let $B:N\to N$. A T(n) time-bounded deterministic Turing machine can be simulated in time $O(\frac{T(n)}{B(n)})$ by a network with $2^{O(B(n)^8)}+T(n)$ processors, word-size $O(B(n)^8+\log T(n))$ and a constant number of registers per processor.

Proof. (Outline). Let M be a T(n) time-bounded k-tape deterministic Turing machine. A *configuration* of M consists of k.T(n) tape symbols corresponding to the tape contents, and k integers corresponding to the head positions on the k tapes. The network has T(n)+1 processors devoted to holding the current configuration of M in an easily-accessible manner. Processor 0 holds the k head positions, and for $0 < i \leq T(n)$ processor i holds the ith symbol of each tape. The simulation will consist of $\left\lceil \frac{T(n)}{B(n)} \right\rceil$ *phases*, each corresponding to B(n) steps of M. The initial configuration of M is easy to set up, and the simulation will endeavor to maintain it from phase to phase.

A *situation* consists of that portion of the tape which may be altered during the current phase, that is, the $k.(2B(n)-1)$ tape-cells that are within distance B(n) from a head at the start of the phase. During each phase the simulation will be conducted using these situations - at the end of each phase the final situation will be used to update the stored configuration. To be more precise, a situation consists of $k(2B(n)-1)$ tape symbols, and k head-pointers (each of $O(\log B(n))$ bits).

Before the first phase, some pre-computation is carried out. A *computation* of M consists of a string of $B(n)+1$ situations. The processors are broken up into $2^{O(B(n)^2)}$ *teams* (one for each computation), each of $B(n)+1$ processors. The lowest numbered processor of each team is a distinguished processor called the *leader* of that team. Our aim is to notify the leaders of the teams which correspond to valid computations of M.

The jth team is made up of processors for which $\lfloor PID/(B(n)+1) \rfloor = j$. The ith member of this team has PID mod $(B(n)+1) = i$. The value j is interpreted as the encoding of a computation (note that this computation is the same for all members of a team). The ith member of each team $0 < i \leq B(n)$ verifies that the ith situation of the computation follows from the $(i-1)$th one by the rules of M, where the situations of a computation are numbered $0,1,\ldots,B(n)$. If not, then that processor is said to *fail*. The team-leader verifies that the heads of the initial situation of the computation are all at cell B(n) of their respective tapes.

Processors which fail notify their team-leader as follows. Each team-leader sets a pre-determined register r to zero. Failed processors then attempt to write a one to register r of their team-leaders. A number of team-leaders will have their register r remain at zero. The computations of their teams correspond to valid computations of M. They extract the initial and final situations I,F of their respective computations, and write F to processor I.

Each phase is broken up into three parts.

(1) Determine the initial situation from the initial configuration of the phase. This time the processors are broken up into $2^{O(B(n))}$ teams (one for each possible situation), each of $2B(n)-1$ processors. The lowest numbered processor of each team is a distinguished processor called the *leader* of that team. Processors which are not members of a team remain idle.

The i^{th} member of the j^{th} team $(i,j \geq 0)$ has $i = \text{PID} \bmod (2B(n)-1)$ and $j = \lfloor \text{PID}/(2B(n)-1) \rfloor$. Each processor first computes i and j. The value j is interpreted as the encoding of a situation (note that j is the same for all members of any particular team). The i^{th} processor of each team, $0 \leq i < (2B(n)-1)$ decodes the head positions and the i^{th} symbol of each tape from this situation. Every processor of every team then compares its symbols to the corresponding symbols of the stored configuration. If they disagree, the processor is said to *fail*. Each team-leader sets a pre-determined register r to zero. Failed processors then attempt to write a one to register r of their team-leaders.

The team leader whose head-pointers are equal to B(n), and whose register r remains at zero knows that its value of j is an encoding of the initial situation of the phase. It writes this value to processor 0 for safe-keeping.

(2) Determine the final situation of the current phase. Processor 0 can obtain this information from processor I, where I is the initial situation of the current phase computed in (1) above. Processor I obtained this information during the pre-computation stage.

(3) Determine the final configuration of the phase from the final situation. Those processors holding symbols of the configuration which are within distance B(n) from a head update their values using the final situation stored in processor 0. Processor 0 likewise updates its head positions.

The network is then in a position to begin the next phase. T(n) steps of M are simulated by repeating this for $\left\lceil \frac{T(n)}{B(n)} \right\rceil$ phases. The pre-computation and every phase each take a constant number of steps, assuming that each processor has the extended arithmetic instruction-set. With care, the restricted arithmetic instruction-set can be substituted, by using $2^{O(B(n)^2)}$ processors per team in the pre-computation, and $2^{O(B(n))}$ processors per team in each phase. The entire simulation takes $O(\frac{T(n)}{B(n)})$ steps, using $2^{O(B(n)^2)}+T(n)$ processors. □

Note that processor 0 is to be given the value of B(n) before the start of a simulation on an input of size n. This result implies that any computable function can be computed in constant time if sufficiently many processors are present. By taking $B(n) = T(n)$ we can extend the simulation to nondeterministic Turing machines. Thus, for example, every function in NP can be computed in constant time by a network of $2^{n^{O(1)}}$ processors. This is an improvement over the result of Savitch [60], who obtains time O(log n) on a network of $n^{O(1)}$ nondeterministic processors. But what about simulation by a "reasonable" machine? Suppose we require that the parallel computation thesis holds. It is sufficient in this case to bound the word-size to be a polynomial in the parallel running time. This means we can choose B(n) to be $T(n)^{1-\varepsilon}$ for $0 < \varepsilon < 1$. Thus a T(n) time bounded deterministic Turing machine can be simulated in time $T(n)^{\varepsilon}$

by a "reasonable" network, for $0 < \varepsilon < 1$. Thus an arbitrary polynomial speedup is possible. This is an extremely strong result, since, as we observed in section 3.3, there are natural problems in P which probably have no exponential speedup on a parallel machine which obeys the parallel computation thesis.

Dymond [18] has achieved a superior result in the case where word-size is to be linear (instead of just polynomial) in the parallel running-time. He obtains parallel time $O(\sqrt{T(n)})$ on $2^{O(\sqrt{T(n)})}$ processors, compared to our $O(T(n)^{2/3})$ on $2^{O(T(n)^{2/3})}$ processors. We can duplicate his result by doing the pre-computation sequentially, in time $O(B(n))$ using $2^{O(B(n))}$ teams, each of one processor. This gives time $O(\frac{T(n)}{B(n)} + B(n))$ on $2^{O(B(n))}$ processors. By using standard techniques it is possible (Blum [6]) to simulate a $T(n)$ time-bounded deterministic Turing machine in time $O(\log T(n))$ using $2^{O(T(n))}$ processors, without the use of multiple writes. We can achieve the same result by choosing $B(n) = \left\lceil \frac{T(n)}{2} \right\rceil$, and doing the pre-computation recursively. Blum thinks that parallel machines with this many processors are "reasonable", and attacks the parallel computation thesis on this basis.

Chapter 4
Programming Techniques for Feasible Networks

In chapter 3 we suggested the possibility of finding a feasible network which is universal for the general network model. Before we actually tackle this problem, it is instructive to investigate the methods at our disposal. This chapter consists of four sections. The first section deals with possible interconnection patterns, concentrating on the shuffle-exchange of Stone [66] and the cube-connected-cycles of Preparata and Vuillemin [55]. The latter paper also provides us with a useful programming tool - a large class of fast algorithms on the multi-dimensional cube (called *composite* algorithms) which can be simulated without loss of resources on either the cube-connected-cycles or shuffle-exchange. This will allow us to express the program of our universal machine in a high-level form which is to a certain extent independent of interconnection pattern.

The second section deals with recurrent interconnection patterns, that is, interconnection patterns $G = (G_0, G_1, \ldots)$ such that for all $k \geq j \geq 0$, G_k is made up of a collection of disjoint subgraphs, each of which is isomorphic to G_j. We present a recurrent interconnection pattern called the cube-connected-lines, which is equal to the cube-connected-cycles in its ability to simulate composite algorithms. It is shown that a recurrent interconnection pattern with twice as many subgraphs as the cube-connected-lines cannot share this property.

The third section contains some composite sub-algorithms which we will later find useful for constructing universal machines. The fourth and final section presents some theorems which allow a reduction in the number of processors in machines with the shuffle-exchange, cube-connected-cycles or cube-connected-lines interconnection patterns, at a cost in time. A reduction in processors from P(n) to P'(n) results in a delay of O(P(n)/P'(n)). Thus constant multiples in processor bounds can be ignored without asymptotic time-loss, a fact that we will use often in later chapters. A preliminary version of the material contained in the last section has appeared in [51].

4.1. Interconnection Patterns and Programming Tools

As suggested in section 3.4, our aim is to construct a feasible network which can efficiently simulate any general network. There are a number of interconnection patterns available in the literature which we might use for this universal machine. These appear to be roughly equal in computing power. Rather than tie ourselves to one particular interconnection pattern, it would be more instructive to express our program in a language which can be implemented efficiently on several interconnection patterns.

Fortunately, the literature already provides us with some tools. Preparata and Vuillemin [55] consider various algorithms which use a multi-dimensional cube as the interconnection pattern. Although this has non-constant degree, they find that a large class of useful algorithms have strong properties which allow them to be simulated without asymptotic time-loss on a feasibly-buildable machine which they call the cube-connected-cycles.

First, let us introduce some useful notation. Suppose v and i are non-negative integers. If $i \geq 1$, let v_i denote the ith least-significant bit in the binary representation of v, that is, $v_i = \lfloor v/2^{(i-1)} \rfloor \bmod 2$. Where convenient, we may confuse the integer v and a binary representation $v_k v_{k-1} \cdots v_1$ (where $k \geq \lfloor \log v \rfloor + 1$) of v. Also let $v^{(i)}$ denote the integer which differs from v precisely in the ith (least-significant) bit, that is, $v^{(i)} = v + (-1)^{v_i}.2^{(i-1)}$. If $v \in \{0,1\}$, let $\bar{v}$ denote $v^{(1)}$, the complement of v.

Suppose k is a non-negative integer. The *k-cube* C_k has vertex-set $\{v \mid 0 \leq v < 2^k\}$, and each vertex v is joined to vertices $v^{(i)}$ for $1 \leq i \leq k$. C_k has 2^k vertices and degree k; it is this high degree which makes it unsuitable as a realistic interconnection pattern. However, it has played an important part in motivating the degree-3 interconnection patterns which we shall meet below. Figure 4.1.1 shows the four-dimensional cube (commonly called the hyper-cube) C_4, which has 16 vertices and degree 4.

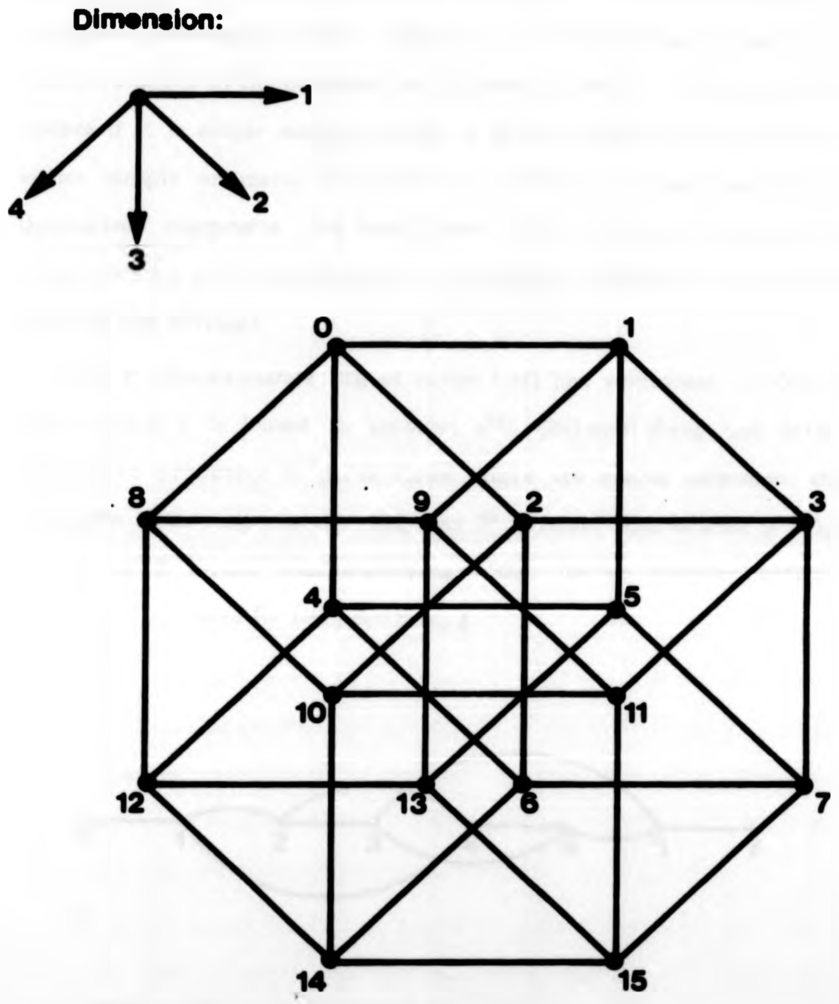


Figure 4.1.1 The hyper-cube, C_4.

Consider a network based on the k-cube, with a constant number of registers per processor. The link between v and $v^{(i)}$ is said to be in *dimension* i. Suppose $k' \leq k$. An algorithm is termed *simple-ascend* (after [55]) if all data transfers occur synchronously along dimension 1, then dimensions 2,3,...,k' in monotone increasing order. Similarly it is called ***simple-descend*** if the data transfers occur in the opposite order, from k' down to 1. An algorithm is called ***simple*** if it is either simple-ascend or simple-descend, and ***composite*** if it is either simple or made up from local instructions and modules which are themselves composite. We learn from [55] that there are fast composite algorithms for a rich selection of data routing problems (such as permutations, merging and sorting).

The ***shuffle-exchange*** SE_k of Stone [66] has vertex-set $\{v \mid 0 \leq v < 2^k\}$, and each vertex v is joined to vertices $v^{(1)}$, $(2v) \bmod 2^k + v_k$ and $\lfloor v/2 \rfloor + v_1.2^{k-1}$. Relative to processor v, these three edges are called ***exchange***, ***shuffle*** and ***unshuffle*** links respectively. SE_k has 2^k vertices and degree 3. Figure 4.1.2 shows the 8 vertex shuffle-exchange SE_3. As an interconnection pattern, $SE = (S_0, S_1, \ldots)$ where for $i \geq 0$, $S_i = SE_{\lfloor \log i \rfloor}$.

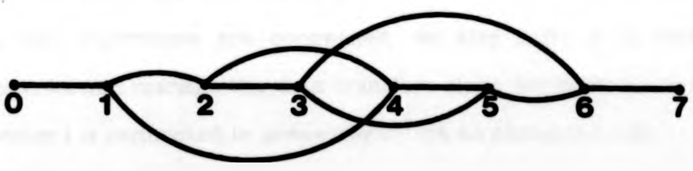


Figure 4.1.2 The 8 vertex shuffle-exchange, SE_3.

From this point onwards, to help avoid possible confusion, we will often call the processors of the simulated machine *processes* in order to distinguish them from the *processors* of the simulating machine. This is consistent with the view that the simulated machine is presented to the simulator as a program, not as a physical collection of processors and wires.

Theorem 4.1.1 A shuffle-exchange with 2^k processors can simulate a 2^k process composite algorithm with constant delay.

Proof. (Outline) Suppose $k' \le k$. Without loss of generality we will prove the result for algorithms whose data transfers occur synchronously along dimensions $1, 2, \ldots, k'-1, k', k'-1, \ldots, 2, 1$ in turn. Both simple-ascend and simple-descend class algorithms fit into this category with constant delay, the former by taking the last k' data transfers to be null, and the latter the first k'. Applying the same technique to each simple module shows that the result also holds for composite algorithms.

Each processor will be assigned the task of simulating one process. Since each process has only a constant number of registers, it is possible to have a simulation in which the processor assignments are flexible. To move a process from one processor to another, we need only transfer the contents of its registers. If this transfer is to take place between neighbouring processors, the entire process can be moved in constant time. We start off with processor i of the shuffle-exchange simulating process i, $0 \le i < 2^k$. Most importantly where composite algorithms are concerned, we also end up in this configuration. Initially, we can manage the data transfers along dimension 1, since for $0 \le i < 2^k$, processor i is connected to processor $i^{(1)}$ via an exchange link.

Next we simply move the entire process from processor i to processor $i_1 i_k i_{k-1} \cdots i_2$ via the unshuffle link out of processor i (which the processor at the other end views as a shuffle link). After this has been done in parallel for all i, $0 \le i < 2^k$, we see that process $i^{(2)}$ is then resident in processor $(i_1 i_k i_{k-1} \cdots i_2)^{(1)}$, which is adjacent to processor $i_1 i_k i_{k-1} \cdots i_2$ via the exchange link. Thus the necessary transfers between processes i and $i^{(2)}$ can take place over the exchange links. After a second unshuffle of processes, data transfers in dimension 3 can take place over the exchange links. This continues up to dimension k', and then is reversed back down to dimension 1. □

The *cube-connected-cycles* CCC_k of Preparata and Vuillemin [55] is defined as follows. Let r be such that $2^{r-1}+r-1<k\leq 2^r+r$. CCC_k has vertex-set $\{(v,p) \mid 0\leq v<2^{k-r}, 0\leq p<2^r\}$, and each vertex (v,p) is joined to vertices:

(i) $(v^{(p+1)},p)$, provided $0\leq p<k-r$,

(ii) $(v,(p+1) \bmod 2^r)$, and

(iii) $(v,(p-1) \bmod 2^r)$.

The first link is called a *cube* edge, the remaining two *cycle* edges. Relative to processor (v,p), the first cycle-edge is called *upcycle*, the second *downcycle*. CCC_k has 2^k vertices and degree 3. Figure 4.1.3 shows the 16 vertex cube-connected-cycles, CCC_4. As an interconnection pattern, $CCC=(G_0,G_1,\ldots)$ where for $i\geq 0$, $G_i=CCC_{\log i}$.

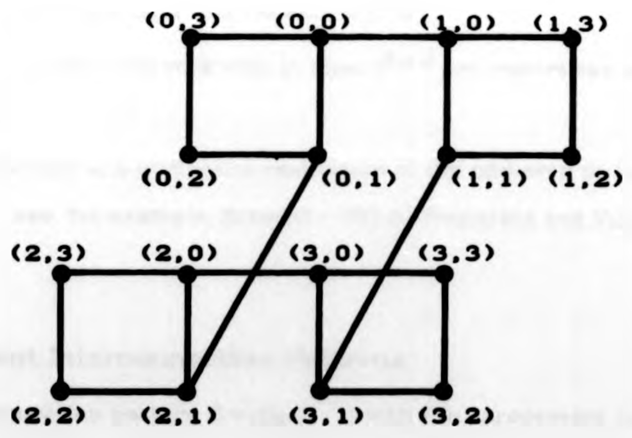


Figure 4.1.3 The 16 vertex cube-connected-cycles, CCC_4.

Theorem 4.1.2 A cube-connected-cycles with 2^k processors can simulate a 2^k process composite algorithm with constant delay.

Proof. Preparata and Vuillemin [55] prove this result for k' (the upper dimension in the simple modules) equal to k. A straightforward modification to the pipelining argument and to LOOPOPER gives us the desired result. □

By application of these theorems we have:

Theorem 4.1.3 A feasible network with at most $2^{\lceil \log n \rceil}$ processors can permute n items according to some fixed permutation in time O(log n) (provided some pre-computation is allowed).

Proof. The algorithm is just a simulation of the permutation network of Waksman [75]. See Schwartz [62] or Preparata and Vuillemin [55]. □

Theorem 4.1.4 A feasible network with at most $2^{\lceil \log n \rceil}$ processors can perform the pre-computation mentioned in theorem 4.1.3 in time $O(\log^4 n)$.

Proof. See, for example, Nassimi and Sahni [46], Schwartz [62], Opferman and Tsao-Wu [48] or Lev, Pippenger and Valiant [42]. □

Theorem 4.1.5 A feasible network with at most $2^{\lceil \log n \rceil}$ processors can sort n items in time $O(\log^2 n)$.

Proof. The algorithm is a composite realization of the odd-even or bitonic sorts of Batcher [4]. See, for example, Schwartz [62] or Preparata and Vuillemin [55]. □

4.2. Recurrent Interconnection Patterns

An interconnection pattern $G = (G_0, G_1, \ldots)$ with P(n) processors is said to be *recurrent* if for all n,m with $0 \le m \le n$, G_n has $\Omega(\frac{P(n)}{P(m)})$ disjoint subgraphs which are isomorphic to G_m. The simplest form of recurrence one might choose is to have G_n constructed from *precisely* $\frac{P(n)}{P(m)}$ such subgraphs. Unfortunately this type of recurrent interconnection pattern is much less powerful than the shuffle-exchange or cube-connected cycles met in section 4.1. Later in this section we will meet a recurrent interconnection pattern where G_n is made up of at least $\frac{P(n)}{2P(m)}$ copies of G_m, which is equal to the cube-connected-cycles in its ability to simulate composite algorithms.

Suppose c is a fixed positive integer (independent of n). More precisely, a *recursive* interconnection pattern is one in which G_n ($n \geq c$) is made up of exactly c disjoint copies of $G_{\lfloor n/c \rfloor}$ (with fixed graphs for $n < c$), joined by extra edges from some graph G'_n.

Theorem 4.2.1 No constant degree recursive parallel machine with O(n) processors can can permute n items in O(log n) steps.

Proof. For a contradiction, suppose $G = (G_0, G_1, \ldots)$ is a recursive interconnection pattern with degree d and O(n) processors which can be used to permute n items in time O(log n). The following technique is due to Meertens [43].

Suppose $n = c^k$ for some $k \geq 0$. Let

P_k denote the number of vertices in G_n.

E_k denote the number of edges in G_n.

E'_k denote the number of edges in G'_n.

Γ_k be $\frac{E_k}{P_k}$.

Note that $\Gamma_k \leq \frac{d}{2}$. (Let S_k be the sum over all vertices v in G_n of the number of edges incident with v. Clearly $S_k \leq d.P_k$. But every edge is counted twice, so $S_k = 2.E_k$). Also, $P_k = \Theta(c^k)$.

We claim that for $0 \leq i < k$,

$$E'_{k-i} = \Omega\left(\frac{c^{k-i-1}}{k-i}\right) \qquad (*)$$

Consider one of the subgraphs of $G_{\frac{n}{c^i}}$ isomorphic to $G_{\frac{n}{c^{i+1}}}$. Pick a permutation section we will meet a recurrent interconnection pattern where G_n is made up of at least $\frac{P(n)}{2P(m)}$ copies of G_m, which is equal to the cube-connected-cycles in its ability to simulate composite algorithms.

Suppose c is a fixed positive integer (independent of n). More precisely, a ***recursive*** interconnection pattern is one in which G_n $(n \geq c)$ is made up of exactly c disjoint copies of $G_{\lfloor n/c \rfloor}$ (with fixed graphs for $n < c$), joined by extra edges from some graph G'_n.

Theorem 4.2.1 No constant degree recursive parallel machine with O(n) processors can can permute n items in O(log n) steps.

Proof. For a contradiction, suppose $G = (G_0, G_1, \ldots)$ is a recursive interconnection pattern with degree d and O(n) processors which can be used to permute n items in time O(log n). The following technique is due to Meertens [43].

Suppose $n = c^k$ for some $k \geq 0$. Let

P_k denote the number of vertices in G_n,

E_k denote the number of edges in G_n,

E'_k denote the number of edges in G'_n,

Γ_k be $\frac{E_k}{P_k}$.

Note that $\Gamma_k \leq \frac{d}{2}$. (Let S_k be the sum over all vertices v in G_n of the number of edges incident with v. Clearly $S_k \leq d.P_k$. But every edge is counted twice, so $S_k = 2.E_k$). Also, $P_k = \Theta(c^k)$.

We claim that for $0 \leq i < k$,

$$E'_{k-i} = \Omega(\frac{c^{k-i-1}}{k-i}) \qquad (*)$$

Consider one of the subgraphs of $G_{\frac{n}{c^i}}$ isomorphic to $G_{\frac{n}{c^{i+1}}}$. Pick a permutation which takes a data item from each (input bearing) vertex of the subgraph to a vertex of $G_{\frac{n}{c^i}}$ outside that subgraph. These data items must pass along the edges of $G'_{\frac{n}{c^i}}$, since these are the only edges linking the subgraph with the rest of $G_{\frac{n}{c^i}}$. Thus in one step, at most E'_{k-i} items can be moved. By hypothesis we can move all the items in $O(k-i)$ steps. There are $\Omega(P_{k-i-1}) = \Omega(c^{k-i-1})$ items to be moved. Hence $c^{k-i-1} = O(E'_{k-i}.(k-i))$ as required.

Now

$$\begin{aligned} E_k &= E'_k + c.E_{k-1} \\ &\geq \sum_{i=0}^{k-1} c^i.E'_{k-i} \\ &= \Omega\left(\sum_{i=0}^{k-1} \frac{c^i.c^{k-i-1}}{k-i}\right) \quad \text{by } (*) \\ &= \Omega\left(\sum_{i=1}^{k} \frac{c^k}{i}\right) \quad \text{by re-indexing} \end{aligned}$$

Thus $\Gamma_k = \frac{E_k}{P_k} = \Omega(\sum_{i=1}^{k} \frac{1}{i})$, which diverges as $k \to \infty$. But this contradicts the fact that $\Gamma_k \leq \frac{d}{2}$, a constant independent of k. Thus no such parallel machine can exist. □

This is in contrast to the corresponding result (theorem 4.1.3) for the cube-connected-cycles and shuffle-exchange.

The following is a recurrent interconnection pattern which is as powerful as the cube-connected-cycles, at least in its ability to simulate composite algorithms. The ***cube-connected-lines***, CCL_k is simply a copy of CCC_k with the edges from vertices $(v,0)$ to $(v,2^r-1)$, $0 \leq v < 2^{k-r}$ deleted. That is, the cycles of the cube-connected-cycles are broken, and thus become lines. Figure 4.2.1 shows some cube-connected-lines graphs with 2,4,8 and 16 vertices. CCL_k has 2^k vertices and degree 3.

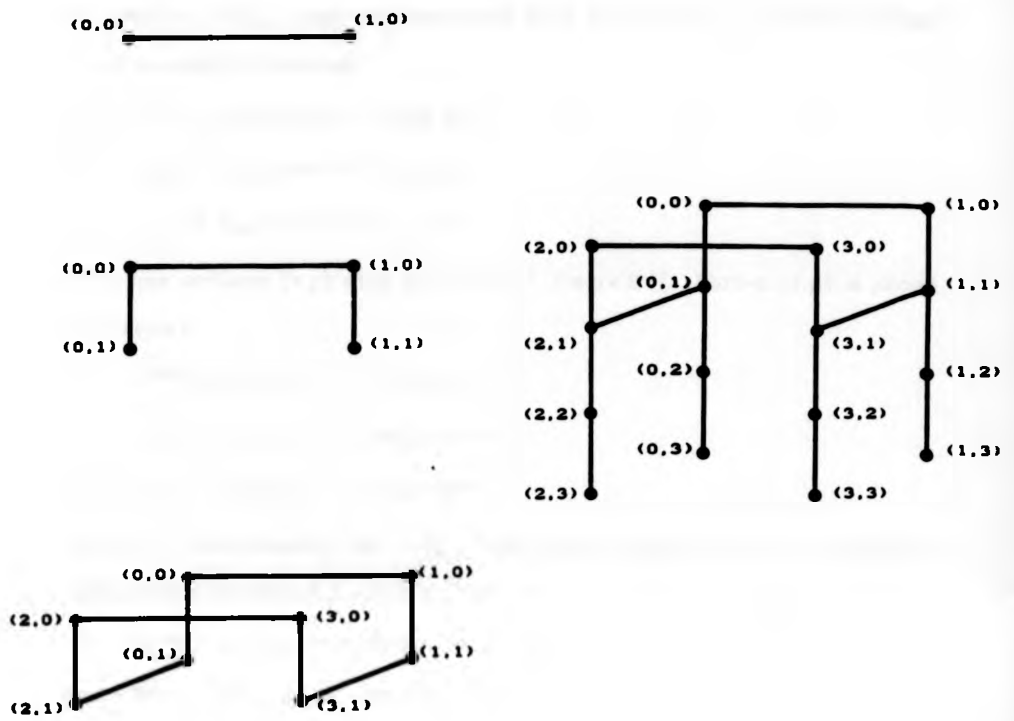


Figure 4.2.1 The 2,4,8 and 16 vertex cube-connected-lines graphs, CCL_1 through CCL_4.

Theorem 4.2.2 For $0 \le j \le k$, CCL_k has at least 2^{k-j-1} disjoint subgraphs which are isomorphic to CCL_j.

Proof. Let $k \ge j \ge 0$, and r be such that $2^{r-1}+r-1 < j \le 2^r+r$. For $r \ge 0$ we call CCL_{2^r+r} a *full* cube-connected-lines graph.

First suppose that $j = k-1$, that is, we wish to break the 2^k vertex CCL_k into half-sized (2^{k-1} vertex) CCL_{k-1}'s. There are two cases to consider, according to whether or not CCL_j is full.

(1) $k-1 = 2^r+r$. CCL_{k-1} has vertices (v,p) with $0 \le v < 2^{k-r-1}$, $0 \le p < 2^r$. Vertex (v,p) is joined to vertices:

(i) $(v^{(p+1)},p)$, $0 \le v < 2^{k-r-1}$, $0 \le p < 2^r$,

(ii) $(v,p+1)$, $0 \le v < 2^{k-r-1}$, $0 \le p < 2^r-1$,

(iii) $(v,p-1)$, $0 \le v < 2^{k-r-1}$, $0 < p < 2^r$.

CCL_k has vertices (v,p) with $0 \le v < 2^{k-r-1}$, $0 \le p < 2^{r+1}$. Vertex (v,p) is joined to vertices:

(i) $(v^{(p+1)},p)$, $0 \le v < 2^{k-r-1}$, $0 \le p < 2^r$,

(ii) $(v,p+1)$, $0 \le v < 2^{k-r-1}$, $0 \le p < 2^{r+1}-1$,

(iii) $(v,p-1)$, $0 \le v < 2^{k-r-1}$, $0 < p < 2^{r+1}$.

Thus CCL_k looks exactly like CCL_{k-1} with lines extended to double the length using vertices without cube links. So CCL_k has only one subgraph which is isomorphic to CCL_{k-1} (see figure 4.2.1 for the case when $k = 2$ and $k = 4$).

(2) $k-1 < 2^r+r$. CCL_{k-1} has vertices (v,p) with $0 \le v < 2^{k-r-1}$, $0 \le p < 2^r$. Vertex (v,p) is joined to vertices:

(i) $(v^{(p+1)},p)$, $0 \le v < 2^{k-r-1}$, $0 \le p < k-r-1$,

(ii) $(v,p+1)$, $0 \le v < 2^{k-r-1}$, $0 \le p < 2^r-1$,

(iii) $(v,p-1)$, $0 \le v < 2^{k-r-1}$, $0 < p < 2^r$.

CCL_k has vertices (v,p) with $0 \le v < 2^{k-r}$, $0 \le p < 2^r$. Vertex (v,p) is joined to vertices:

(i) $(v^{(p+1)},p)$, $0 \le v < 2^{k-r}$, $0 \le p < k-r$,

(ii) $(v,p+1)$, $0 \le v < 2^{k-r}$, $0 \le p < 2^r-1$,

(iii) $(v,p-1)$, $0 \le v < 2^{k-r}$, $0 < p < 2^r$.

Deleting the cube-edges from (v,p) to $(v^{(p+1)},p)$ with $p = k-r-1$ from CCL_k gives two disjoint graphs which are isomorphic to CCL_{k-1} (see figure 4.2.1 for the case when $k = 3$).

Thus for $2^{r-1}+r-1<j\leq k\leq 2^r+r$ we can break CCL_k into 2^{k-j} subgraphs isomorphic to CCL_j, by iterating the procedure in (2) above. By then applying (1), we see that CCL_k has 2^{k-j-1} subgraphs isomorphic to CCL_j when $j=2^{r-1}+r-1$. It remains to show what happens when j and k are separated by more than one full CCL.

Now suppose $k=2^r+r$ and $j=2^{r'}+r'$ for some $r\geq r'\geq 0$. CCL_j has vertices (v,p), $0\leq v<2^{2^{r'}}$, $0\leq p<2^{r'}$. Vertex (v,p) is joined to vertex:

(i) $(v^{(p+1)},p)$, $0\leq v<2^{2^{r'}}$, $0\leq p<2^{r'}$,

(ii) $(v,p+1)$, $0\leq v<2^{2^{r'}}$, $0\leq p<2^{r'}-1$,

(iii) $(v,p-1)$, $0\leq v<2^{2^{r'}}$, $0<p<2^{r'}$.

CCL_k has vertices (v,p), $0\leq v<2^{2^r}$, $0\leq p<2^r$. Vertex (v,p) is joined to vertex:

(i) $(v^{(p+1)},p)$, $0\leq v<2^{2^r}$, $0\leq p<2^r$,

(ii) $(v,p+1)$, $0\leq v<2^{2^r}$, $0\leq p<2^r-1$,

(iii) $(v,p-1)$, $0\leq v<2^{2^r}$, $0<p<2^r$.

Deleting the line-edges between vertices $(v,i.2^{r'}-1)$ and $(v,i.2^{r'})$ for $0\leq v<2^{2^r}$, $0\leq i<2^{r-r'}$ serves to break CCL_k into 2^{k-j} graphs isomorphic to CCL_j. Thus a full CCL_k has 2^{k-j} disjoint subgraphs isomorphic to a full CCL_j.

Finally, we now have the tools to prove the result for general j and k.

(a) First reduce CCL_k into subgraphs isomorphic to the next smaller full CCL, using (1) and (2) as mentioned above. If CCL_j is encountered along the way, then this is sufficient. If j and k are separated by precisely one full CCL, further iterations of (2) are sufficient.

(b) Next, reduce the full CCL immediately below CCL_k into subgraphs isomorphic to the CCL immediately above CCL_j. The latter can be reduced to CCL_j by subsequent iterations of (2).

In this entire process we only once have to reduce a non-full CCL to subgraphs which are isomorphic to full ones. Thus CCL_k consists of 2^{k-j-1} subgraphs isomorphic to CCL_j. □

Note that any attempt to increase the number of subgraphs from 2^{k-j-1} to 2^{k-j} is doomed to failure. For if CCL_k had 2^{k-j} subgraphs isomorphic to CCL_j, it would then be recursive. Thus by theorem 4.2.1 it would be much weaker than the cube-connected-cycles for computing arbitrary permutations. However we have:

Theorem 4.2.3 A cube-connected-lines with 2^k processors can simulate a 2^k process composite algorithm with constant delay.

Proof. Similar to theorem 4.1.2. □

Reif and Valiant [57] have independently discovered a graph which is almost identical to the cube-connected-lines. A degree-4 graph with similar properties was earlier devised by Meyer auf der Heide [31,32].

4.3. Some Useful Algorithms

Having developed the idea of a composite algorithm in section 4.1, we are now ready to describe some simple sub-algorithms which we will find useful in the next three chapters. The algorithms are given for the k-cube interconnection pattern, but can be simulated without asymptotic loss of resources on either the shuffle-exchange, cube-connected-cycles or cube-connected-lines interconnection patterns, as described in sections 4.1 and 4.2. It is important to note that the algorithms are SIMD in nature; synchronization is maintained by the fact that (as we earlier insisted in section 2.1) the code generated for each branch of a selection statement (such as if-then-else, even if the "else" branch is null) has the same number of instructions.

Algorithm 1. Broadcast.

Suppose processor 0 has a value v which it wishes to broadcast to all 2^k processors of a k-cube. This can be achieved in time O(k) by the following simple-ascend algorithm, which terminates with variable V of every processor equal to v.

```
V:= if PID = 0  then v  else 0
for b:=1 to k do
   if $PID_b = 1$  then V:= (V of processor $PID^{(b)}$)
```

If $0 \leq i < 2^k$, define the *b-block* (after [45,47]) of processor i to be the set of 2^b processors $\{ \lfloor i/2^b \rfloor . 2^b + j \mid 0 \leq j < 2^b \}$. It is easy to prove by induction that after the b^th iteration of the for-loop, variable V of all processors in the b-block of processor 0 is equal to v, for b = 0,1,...,k. (By the 0^th iteration, we mean the point immediately before the loop is entered for the first time). Table 4.3.1 shows a trace of the algorithm for k=4. The concept of a b-block will play an important part in the next two algorithms.

	V of Processor															
b	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	v	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	v	v	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	v	v	v	v	0	0	0	0	0	0	0	0	0	0	0	0
3	v	v	v	v	v	v	v	v	0	0	0	0	0	0	0	0
4	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v	v

Table 4.3.1 Trace of algorithm 1 on 16 processors. Table entry shows the contents of variable V of each processor after b iterations of the for-loop.

Algorithm 2. Local Rank.

Suppose every processor of a k-cube holds an integer value in some variable V. The *local rank* of processor i, $0 \le i < 2^k$ is defined to be the number of processors j, $0 \le j < i$, such that for all processors p with $j \le p \le i$, V of processor p equals V of processor i. The following is a simple-ascend class algorithm which sets variable R of each processor to its local rank, and runs in time O(k).

```
VT:=V
R:=RT:=0
for b:=1 to k do
  if ($PID_b$ = 1) and (VT of processor $PID^{(b)}$) = V
    then R := R+(RT of processor $PID^{(b)}$)+1
  if (VT of processor $PID^{(b)}$) = VT
    then RT := RT+(RT of processor $PID^{(b)}$)+1
    else if $PID_b$ = 0 then (VT,RT) := (VT,RT) of processor $PID^{(b)}$
```

At the end of the bth iteration of the for-loop, $0 \le b \le k$, variable R of processor i, $0 \le i < 2^k$ holds that processor's local rank within its b-block. At the same time, variables VT and RT contain the values V and R (respectively) of the topmost processor in its b-block (i.e. processor $\lfloor i/2^b \rfloor . 2^b + 2^b - 1$). The correctness of the algorithm follows by induction. Table 4.3.2 shows a trace for k=3, with V of processor i initially equal to 0,0,0,6,6,6,6,1 for i=0,1,...,7 respectively.

	Processor																							
	0			1			2			3			4			5			6			7		
	V=0			V=0			V=0			V=6			V=6			V=6			V=6			V=1		
b	R	VT	RT	R	VT	RT	R	VT	RT	R	VT	RT	R	VT	RT	R	VT	RT	R	VT	RT	R	VT	RT
0	0	0	0	0	0	0	0	0	0	0	6	0	0	6	0	0	6	0	0	6	0	0	1	0
1	0	0	1	1	0	1	0	6	0	0	6	0	0	6	1	1	6	1	0	1	0	0	1	0
2	0	6	0	1	6	0	2	6	0	0	6	0	0	1	0	1	1	0	2	1	0	0	1	0
3	0	1	0	1	1	0	2	1	0	0	1	0	1	1	0	2	1	0	3	1	0	0	1	0

Table 4.3.2 Trace of algorithm 2 on 8 processors. Table entry shows the contents of variables R, VT and RT of each processor after b iterations of the for-loop.

Algorithm 3. Fan-out.

Suppose every processor of a k-cube holds two integer values x and y (which may be different for each processor). Our aim is to produce a value fanout(i) for each processor i, where for $0 \le i < 2^k$, fanout(i) is defined to be the y-value of the smallest numbered processor j such that for all p with $j \le p \le i$, x of processor p is equal to x of processor i. The following is a simple-ascend class algorithm which sets variable Y of processor i to fanout(i), $0 \le i < 2^k$, in time O(k).

```
Y:=y
XT,YT:= x,y
**for** b:=1 **to** k **do**
  **if** ($PID_b = 1$)and(XT **of processor** $PID^{(b)}$ = x)
    **then** Y:= YT **of processor** $PID^{(b)}$
  **if** (XT = XT **of processor** $PID^{(b)}$)$\Leftrightarrow$($PID_b = 1$)
    **then** (XT,YT) := (XT,YT) **of processor** $PID^{(b)}$
```

At the end of the bth iteration of the for-loop, $0 \le b \le k$, variable Y of processor i contains the value of fanout(i) restricted to its b-block. At the same time, variables XT and YT contain the values of X and Y (respectively) of the highest-numbered processor in its b-block (i.e. processor $\lfloor i/2^b \rfloor . 2^b + 2^b - 1$). The correctness of the algorithm follows by induction. Table 4.3.3 shows a trace for $k = 3$ with (x,y) of processor i equal to (0,99), (0,89), (0,69), (6,95), (6,19), (6,28), (6,56), (1,44) for $i = 0,1,\ldots,7$ respectively.

	Processor																							
	0			1			2			3			4			5			6			7		
	x=0 y=99			x=0 y=89			x=0 y=69			x=6 y=95			x=6 y=19			x=6 y=28			x=6 y=56			x=1 y=44		
b	Y	XT	YT	Y	XT	YT	Y	XT	YT	Y	XT	YT	Y	XT	YT	Y	XT	YT	Y	XT	YT	Y	XT	YT
0	99	0	99	89	0	89	69	0	69	95	6	95	19	6	19	28	6	28	56	6	56	44	1	44
1	99	0	99	99	0	99	69	6	95	95	6	95	19	6	19	19	6	19	56	1	44	44	1	44
2	99	6	95	99	6	95	99	6	95	95	6	95	19	1	44	19	1	44	19	1	44	44	1	44
3	99	1	44	99	1	44	99	1	44	95	1	44	95	1	44	95	1	44	95	1	44	44	1	44

Table 4.3.3 Trace of algorithm 3 on 8 processors. Table entry shows the contents of variables X, XT and YT of each processor after b iterations of the for-loop.

Algorithm 4. Scatter.

For the moment we briefly step away from the main theme of this chapter, and allow the processors of our machines to have more than a constant number of registers each. In particular, we want each of the 2^k processors of a k-cube to have an array of 2^k elements. Suppose processor 0 has 2^k items of data in its array, and wishes to scatter these amongst processors $0,1,\ldots,2^k-1$ in such a manner that each processor receives precisely one value. The algorithm consists of k stages. At the end of the i th stage, the 2^i processors p, $0 \le p < 2^i$, are each in possession of 2^k data items. Stage i consists of processor p, $0 \le p < 2^i$ sending 2^{k-i} of its data items to processor $p^{(i)}$. In the following implementation, processor 0 starts off with 2^k items of data in an array $d[1..2^k]$. Each processor receives its value into variable $d[1]$.

```
for i:=1 to k do
  for j:=1 to $2^{k-i}$ do
    if $PID_i = 1$
      then $d[j] := (d[j+2^{k-i}]$ of processor $PID^{(i)})$
      else $d[j+2^{k-i}] := 0$
```

Table 4.3.4 shows a trace for $k = 3$, with $d[i]$ of processor 0 initially containing i, $1 \le i \le 8$.

The algorithm runs in time $O(\sum_{i=1}^{k} 2^{k-i}) = O(2^k)$ on a k-cube, but is not strictly simple-ascend (because dimension i is used 2^{k-i} times in succession, not merely once). This makes very little difference as far as the shuffle-exchange is concerned (see the proof of theorem 4.1.1). A minor modification to the proofs of theorems 4.1.2 and 4.2.3 serves to give the same result for the cube-connected-cycles and cube-connected-lines interconnection patterns, the key point being the fact that after the i^{th} iteration, $0 \le i \le k$, only 2^i processors are in possession of data items.

		d(j) of processor							
i	j	0	1	2	3	4	5	6	7
0	1	1							
	2	2							
	3	3							
	4	4							
	5	5							
	6	6							
	7	7							
	8	8							
1	1	1	5						
	2	2	6						
	3	3	7						
	4	4	8						
2	1	1	5	3	7				
	2	2	6	4	8				
3	1	1	5	3	7	2	6	4	8

Table 4.3.4 Trace of algorithm 4 on 8 processors. Table entry shows the contents of d[j] of each processor for various values of j after i iterations of the outer for-loop. A blank entry denotes a content of 0.

4.4. Reducing the Number of Processors

In this section we examine a particular kind of time/processor trade-off, namely, the question of whether a reduction in the number of processors of a network based on the shuffle-exchange, cube-connected-cycles or cube-connected-lines interconnection patterns can be made at a reasonable cost in time. We find that a small machine based on these interconnection patterns can simulate a larger one by having each processor of the former simulate many of the latter. The fact that this simple approach works is of course due to the highly regular form of the interconnection patterns under consideration.

This section is partly motivated by a result of Galil and Paul. In [21] they transform the standard n-processor algorithm for bitonic sort into an algorithm which can sort n numbers using $m \le n$ processors in time $O(\frac{n}{m}.\log m.\log n)$ on any of the graphs considered above. This raises some interesting questions. Do similar results hold for *all* algorithms on these graphs? Does the result of Galil and Paul depend on some special property of the sorting algorithm? We are able to provide an affirmative answer to the first question. With regard to the second question, (as one might expect) the time-bounds achieved by our general transformations are slightly inferior to that of the special-purpose transformation for bitonic sort. The results work both for the general model of register access proposed in section 2.1, and the restricted-access model.

Our processor-saving theorems have many applications. Firstly, as pointed out in [21], in many situations the input to a parallel computer cannot be read in parallel (this assumption has been made, for example, in [44,49,76]). In this case our results can be applied to slow down various fast parallel algorithms to the speed at which the input becomes available, and thus make a large decrease in the number of processors without any observable increase in time. Secondly, we can throw some light on the importance of constant factors in processor bounds. For example, Galil and Paul [21, theorem 7] are able to reduce the number of processors in their universal parallel machine from O(p) to p while increasing time by only a constant multiple. We are able to extend this result by showing that the number of processors in any parallel machine based on many popular interconnection patterns can be reduced by a constant factor without asymptotic time-loss.

Constant multiples in processor-bounds pervade the current literature, due to the fact that the commonly used interconnection patterns come only in certain sizes, typically 2^k or $k.2^k$ for some non-negative integer k. Thus to process n inputs may take more than n processors. For example, it has been shown (see theorem 4.1.3) that it is possible to permute n items on a $2^{\lceil \log n \rceil}$ processor cube-connected-cycles or shuffle-exchange in time O(log n) by simulating Waksman's [75] permutation network. In this case it may be necessary to use as many as $2n-2$ processors. Our results enable us to remove these "hidden" constants without asymptotic time loss.

Our technique is motivated by the following result.

Theorem 4.4.1. For all $d \geq 0$, C_k can simulate C_{k+d} with delay $O(2^d)$.

Proof. In order to simulate a single step of C_{k+d}, every processor i, $0 \leq i < 2^k$ will synchronously execute a single step of processes $2^d.i + j$, for $0 \leq j < 2^d$. This takes place in two stages.

(1) For $0 \leq j < 2^d$ simulate the communication between process $2^d.i + j$ and its neighbouring processes. Suppose, for example, process $2^d.i + j$ wishes to communicate with process $(2^d i + j)^{(m)}$ for some m with $1 \leq m \leq k + d$. If $m \leq d$ then no interprocessor communication is necessary. Otherwise $d < m \leq k + d$ and so process $(2^d.i + j)^{(m)} = 2^d.i^{(m-d)} + j$ is being simulated by processor $i^{(m-d)}$. Since processor $i^{(m-d)}$ is a neighbour of processor i in C_k, the desired communication can be carried out in O(1) steps, the exact constant being dependent on the type of instruction-set in use. Note that communications with process $2^d.i + j$ may be initiated by processes $2^d.i^{(m-d)} + j$, $1 \leq m \leq k$, resident in every neighbour $i^{(m)}$ of processor i. We

assume that the communication protocol of C_k deals with these possible clashes in the same manner as that of C_{k+d}. Clashes involving a communication between two processes resident in the same processor are to be dealt with in a manner which is compatible with this protocol.

(2) Finally, simulate the current step of processes $2^d.i+j$, $0 \leq j < 2^d$, making possible use of the information obtained in (1). This is assumed to take O(1) steps per process (the constant again being dependent on the instruction-set).

At this point, C_k is ready to simulate the next step of C_{k+d}. Thus we have simulated a single step of a 2^{k+d} processor machine C_{k+d} on a 2^k processor C_k with a time-loss of only $O(2^d)$.□

Note that the simulation of theorem 1 can be carried out in such a manner that it maintains the simple-ascend or simple-descend property of section 4.1. Thus it is strong enough to achieve the desired savings in processors for composite algorithms.

As observed earlier in this section, in some instances the input to a parallel machine may not be available in parallel. The example taken by Galil and Paul in [21] is that of matrix multiplication. It is well-known [55] that two $n \times n$ matrices can be multiplied in time $O(\log n)$ using $O(n^3)$ processors on any of the graphs listed in sections 4.1 and 4.2 provided the input can be read in parallel. Suppose, to the contrary, that the input can only be read sequentially, so that there is an *a priori* lower bound of $\Omega(n^2)$. By applying theorem 1 with $k = \lfloor \log(n.\log n) \rfloor$ and $d = \lceil 3.\log n \rceil - k$, we have a linear-time (i.e. $O(n^2)$) algorithm on $n.\log n$ processors. If a row (or column) of the input can be read in parallel, theorem 1 with $k = \left\lfloor \log(n^2.\log n) \right\rfloor$ and $d = \lceil 3.\log n \rceil - k$ gives us an $O(n)$ time algorithm on $O(n^2.\log n)$ processors. This improvement over the corresponding results in [21] stems from the fact that they have used a universal parallel machine, which leads to a significant degradation in performance.

Whilst theorem 1 gives the claimed savings in processors for an important class of algorithms on the shuffle-exchange, cube-connected-cycles and cube-connected-lines, our aim is to produce a stronger result which holds for *all* algorithms on these graphs. Fortunately the interconnection patterns are sufficiently like the k-cube for similar techniques to work. The shuffle-exchange, for instance, is particularly amenable.

Theorem 4.4.2 For all $d \geq 0$, SE_k can simulate SE_{k+d} with delay $O(2^d)$.

Proof. In order to simulate one step of SE_{k+d}, every processor i, $0 \leq i < 2^k$ will synchronously simulate one step of processes $2^d.i+j$, for $0 \leq j < 2^d$. As in the proof of theorem 1, this takes place in two phases - each processor first carries out any communications required by its processes from their respective neighbours, then updates their configurations once all the information has been gathered.

Suppose $0 \leq j < 2^d$ and process $2^d.i+j$ wishes to communicate with one of its neighbours in SE_{k+d}. There are three cases to be considered, according to whether process $2^d.i+j$ wishes to communicate with its neighbour along the exchange, shuffle or unshuffle edge.

Either:

(1) *Exchange link*. It wishes to communicate with process $(2^d.i+j)^{(1)}$. In this case, no interprocessor communication is necessary, since process $(2^d.i+j)^{(1)} = 2^d.i+j^{(1)}$ is also being simulated by processor i.

(2) *Shuffle link*. It wishes to communicate with process $i_{k-1}i_{k-2} \cdots i_1 j_d j_{d-1} \cdots j_1 i_k$. This process is being simulated by processor $i_{k-1}i_{k-2} \cdots i_1 j_d$. There are two cases to consider:

(a) $i_k = j_d$. Processor $i = j_d i_{k-1} i_{k-2} \cdots i_1$ can communicate with processor $i_{k-1} i_{k-2} \cdots i_1 j_d$ directly through its shuffle link.

(b) $i_k \neq j_d$. Processor $i = \bar{j}_d i_{k-1} i_{k-2} \cdots i_1$ can communicate indirectly with processor $i_{k-1} i_{k-2} \cdots i_1 j_d$ by utilizing the shuffle link to processor $i_{k-1} i_{k-2} \cdots i_1 \bar{j}_d$, and the exchange link from there to processor $i_{k-1} i_{k-2} \cdots i_1 j_d$.

(3) *Unshuffle link*. It wishes to communicate with process $j_1 i_k i_{k-1} \cdots i_1 j_d j_{d-1} \cdots j_2$. This process is being simulated by processor $j_1 i_k i_{k-1} \cdots i_2$. Again, there are two cases to be considered:

(a) $i_1 = j_1$. Processor $i = i_k i_{k-1} \cdots i_2 j_1$ can communicate directly with processor $j_1 i_k i_{k-1} \cdots i_2$ through its unshuffle link.

(b) $i_1 \neq j_1$. Processor $i = i_k i_{k-1} \cdots i_2 \bar{j}_1$ can communicate indirectly with processor $j_1 i_k i_{k-1} \cdots i_2$ by utilizing the exchange link to processor $i_k i_{k-1} \cdots i_2 j_1$ and the unshuffle link from there to processor $j_1 i_k i_{k-1} \cdots i_2$.

Thus we have shown how to simulate one step of the 2^{k+d} processor SE_{k+d} on the 2^k processor SE_k with a time-loss of only $O(2^d)$, the constant multiple being dependent on the instruction-set used. □

The results for the variants of the cube-connected-cycles are proved in a similar manner.

Theorem 4.4.3 For all $d \geq 0$,

(*i*) CCC_k can simulate CCC_{k+d}, and

(*ii*) CCL_k can simulate CCL_{k+d},

with delay $O(2^d)$.

Proof. We will demonstrate the technique for $d = 1$. This can be looked upon as a recursive algorithm for the processor assignment, along with a proof that the assignment is valid. We consider CCC_k first.

First, suppose that k is of the form $2^r + r$ for some integer $r \geq 0$. Then the processors of the simulating machine are of the form (v,p) where $0 \leq v < 2^{2^r}$, $0 \leq p < 2^r$. In contrast, the processes of the simulated machine have the form (v,p) with $0 \leq v < 2^{2^r}$, $0 \leq p < 2^{r+1}$. Processor (v,p), $0 \leq v < 2^{2^r}$, $0 \leq p < 2^r$ will simulate processes (v,p) and $(v, 2^{r+1}-p-1)$. As before, each processor synchronously carries out the communications requested by all its processes, and then updates their configurations internally.

Suppose process (v,p) wishes to communicate with one of its neighbours. There are three cases to consider.

(1) *Cube link*. It wishes to communicate with process $(v^{(p+1)}, p)$. This process is being simulated by processor $(v^{(p+1)}, p)$ which is directly connected to processor (v,p) via a cube link.

(2) *Upcycle link*. It wishes to communicate with process $(v, p+1)$. If $0 \leq p < 2^r-1$ then process $(v, p+1)$ is being simulated by processor $(v, p+1)$, which is directly connected to processor (v,p) by an upcycle link. Otherwise $p = 2^r-1$ and process $(v, 2^r)$ is also being simulated by processor (v,p), so no interprocessor communication is necessary.

(3) *Downcycle link*. It wishes to communicate with process $(v, (p-1) \bmod 2^{r+1})$. If $0 < p < 2^r$ then process $(v, p-1)$ is being simulated by processor $(v, p-1)$, which is directly connected to processor (v,p) by a downcycle link. Otherwise $p = 0$ and process $(v, 2^{r+1}-1)$ is also being simulated by processor (v,p), so no interprocessor communication is necessary.

Now suppose that process $(v, 2^{r+1}-p-1)$ wishes to communicate with one of its neighbours. This is handled similarly to (2) and (3) above (remembering that processes of this form have only cycle links).

This completes the case where k is of the form $2^r + r$. Now suppose k is not of that form. Let r be such that $2^{r-1} + r - 1 < k < 2^r + r$. The processors of the simulating machine are of the form (v,p) where $0 \leq v < 2^{k-r}$, $0 \leq p < 2^r$. The processes of the simulated machine are of the form (v,p) where $0 \leq v < 2^{k-r+1}$, $0 \leq p < 2^r$.

Processor (v,p) will simulate processes (v,p) and $(v + 2^{k-r}, p)$. As always, in order to simulate a single step, each processor first carries out the communications required by its processes, and then updates their configurations internally.

Suppose process (v,p) wishes to communicate with one of its neighbours. There are two cases to consider.

(1) *Cube link*. It wishes to communicate with process $(v^{(p+1)}, p)$. If $p < k - r$ then process $(v^{(p+1)}, p)$ is being simulated by processor $(v^{(p+1)}, p)$. Otherwise $p = k - r$ and process $(v^{(p+1)}, p) = (v + 2^{k-r}, p)$ is also being simulated by processor (v,p), so no interprocessor communication is necessary. Note that p cannot exceed k−r since processes (v,p) with $p > k - r$ have no cube links.

(2) *Cycle links*. It wishes to communicate with process $(v, (p \pm 1) \bmod 2^r)$. This process is being simulated by processor $(v, (p \pm 1) \bmod 2^r)$ respectively, which is connected to processor (v,p) by a cycle link.

This completes the simulation of process (v,p). Process $(v + 2^{k-r}, p)$ is handled similarly.

Thus we have shown how to simulate a step of CCC_{k+d} on CCC_k in time $O(2^d)$. Since CCL_k is a subgraph of CCC_k, part (*ii*) of the theorem follows immediately.□

Note that the set-up times for theorems 4.4.1 and 4.4.2 are far superior to that of theorem 4.4.3. Not only are the assignments of processes to processors easier to compute, but also the input symbols are placed into the correct processors at the start of a computation according to the convention established in section 2.1.

Chapter 5
Practical Simulations

This chapter is devoted to simulations of our general network model on more practical models of parallel computation. The first section contains a general theorem which characterizes the computational power needed to simulate a resource-bounded network. Many specific instances of this theorem (for particular machine models) have already appeared in the literature [4,8,21,42,45,71,72]. In the second section we construct our universal feasible network. This feasible network is, as has already been mentioned, to be universal for the general network model. The result follows as a fairly straightforward corollary to the general theorem of the first section, by application of the techniques developed in chapter 4.

In the third section we propose a hardware measure for general networks. This hardware measure is compared to popular definitions of hardware which have appeared in the literature (including size and width of uniform circuits), using simulations based on the result of section 1. We examine the extended parallel computation thesis [16,17] as a criterion for "reasonableness" based on the resources of time and hardware. This states that time and hardware on any reasonable parallel machine model are simultaneously polynomially related to space and reversals on a deterministic Turing machine. The fourth and final section is devoted to obtaining improved simulations of space and reversal-bounded deterministic Turing machines by width and depth-bounded uniform circuits.

A preliminary version of the material in this chapter can be found in [50].

5.1. A General Simulation Theorem

The central result of this section is a theorem which describes the computational power needed to simulate a resource-bounded network. As a fairly easy corollary, we will in section 5.2 be able to construct a feasible network which is universal for the general model of section 2.1. In order to keep the proof as manageable as possible, the simulation will be functional rather than machine-based.

If $n>0$, define an *n-tuple* X (for $n>0$) over some set S to be a sequence of n elements $<X_0,X_1, \ldots ,X_{n-1}>$, such that $X_i \in S$, $0 \le i < n$. Let S^n denote the set of all n-tuples over S and $S \times T$ denote $\{<s,t> | s \in S, t \in T\}$ for arbitrary sets S and T. Ordering of n-tuples is done lexicographically (first-field-first). For example, if $X,Y \in Z^n$, then $X<Y$ iff there exists j with $0 \le j < n$ such that $X_j < Y_j$ and for $0 \le i < j$, $X_i = Y_i$. Let $S^* = \bigcup_{n \ge 0} S^n$.

Let M be a P(n) processor, S(n) space bounded network. To simplify the presentation we will assume that:

(i) All local instructions operate only on registers r_0,r_1,r_2. Only registers r_i for $i \ge 3$ can be read from, or written to.

(ii) Read instructions have the form "extract values p,a from r_0,r_1 respectively, read register r_a of processor p and place the value obtained into r_0".

(iii) Write instructions have the form "extract values p,a,w from registers r_0,r_1,r_2 respectively and write w into register r_a of processor p".

(iv) Multiple reads are allowed, and in the case of write conflicts, the smallest value being written into a register is the one which succeeds.

Note that (i), (ii) and (iii) are sufficient for the example instruction-set of section 2.1, since a processor can address its own registers by the use of reads and writes. The general case follows in a similar manner.

For convenience we define a special null element *null* and adopt the conventions that for all X and n, *null* is always a member of X^n, and that for all $i \geq 0$, $null_i = S(n)$. Define the *configuration* of M to be a member of $C_M = (Z^3 \times N)^{P(n)} \times (N^2 \times Z)^{S(n)}$. For example, we take

$$<<x_0, x_1, \cdots, x_{P(n)-1}>, <y_0, y_1, \cdots, y_{S(n)-1}>>$$

to indicate the following. If $x_i = <a_i, b_i, c_i, d_i>$ then processor i has values a_i, b_i, c_i in its registers r_0, r_1, r_2 respectively, and it is to execute the d_ith instruction of the program of M next (with d_i out of range indicating that the processor has halted). If $null \neq y_i = <p_i, a_i, v_i>$, $a_i \geq 3$, then register r_{a_i} of processor p_i contains v_i. In particular, where the latter is concerned we insist that:

(i) Only registers with non-zero contents are listed. These are listed left-justified in increasing order of $<p_i, a_i>$.

(ii) The remaining entries are filled, if necessary, with *null* s.

Definitions. We now define some useful functions. Let $\text{sorted}((Z^n)^m) \subseteq (Z^n)^m$ be the set of m-sequences X such that $X_0 \leq X_1 \leq \cdots \leq X_{m-1}$. For convenience, if $X \in (Z^n)^m$, let $(X_{-1})_0 = (X_m)_0 = -1$. Then

(1) $\text{sort}:(Z^3)^n \rightarrow (Z^3)^n$ maps unsorted sequences of ordered pairs into sorted ones. More precisely,

$$\text{sort}(X)_i = \begin{cases} X_j & \text{if } i = |\{k \mid 0 \leq k < n, X_k < X_j\}| \\ null & \text{otherwise} \end{cases}$$

(2) $\text{merge}:\text{sorted}((Z^3)^n) \times \text{sorted}((Z^3)^m) \rightarrow (Z^4)^{n+m}$ merges two sorted sequences of ordered pairs.

$$\text{merge}(X,Y)_i = \begin{cases} <(X_j)_0, (X_j)_1, (X_j)_2, 1> & \text{if } X_j \neq \text{null},\ i = |\{k \mid X_k < X_j\}| + \\ & |\{k \mid <(Y_k)_0, (Y_k)_1> \leq <(X_j)_0, (X_j)_1>\}| \\ <(Y_j)_0, (Y_j)_1, (Y_j)_2, 0> & \text{if } Y_j \neq \text{null},\ i = |\{k \mid Y_k < Y_j\}| + \\ & |\{k \mid <(X_k)_0, (X_k)_1> < <(Y_j)_0, (Y_j)_1>\}| \\ null & \text{otherwise} \end{cases}$$

(3) fanout:sorted$((Z^4)^n) \to (Z^8)^n$ achieves the fan-out of data values to multiple read requests.

$$\mathrm{fanout}(X)_i = \begin{cases} \langle (X_i)_2, (X_j)_2 \rangle & \text{if there exists } j \neq i \text{ such that } (X_j)_0 = (X_i)_0 \text{ and } (X_j)_1 = (X_i)_1 \\ & \text{and } \langle (X_{j-1})_0, (X_{j-1})_1 \rangle \neq \langle (X_j)_0, (X_j)_1 \rangle \text{ and } (X_j)_3 = 0 \\ \langle (X_i)_2, 0 \rangle & \text{if there exists } j \text{ such that } (X_j)_0 = (X_i)_0 \text{ and } (X_j)_1 = (X_i)_1 \\ & \text{and } \langle (X_{j-1})_0, (X_{j-1})_1 \rangle \neq \langle (X_j)_0, (X_j)_1 \rangle \text{ and } (X_j)_3 = 1 \\ \mathit{null} & \text{otherwise} \end{cases}$$

(4) deliver:$(Z^4)^n \to (Z^3)^n$ performs the fan-in of multiple write requests.

$$\mathrm{deliver}(X)_i = \begin{cases} \langle (X_i)_0, (X_i)_1, (X_i)_2 \rangle & \text{if } (X_i)_3 = 0 \text{ and } X_{i+1} \neq \langle (X_i)_0, (X_i)_1, v, 1 \rangle \text{ for all } v \in Z, \\ & \text{or } (X_i)_3 = 1 \text{ and } (X_i)_2 \neq 0 \text{ and } X_{i-1} \neq \langle (X_i)_0, (X_i)_1, v, 1 \rangle \\ & \text{for all } v \in Z. \\ \mathit{null} & \text{otherwise} \end{cases}$$

(5) concentrate:$(Z^*)^n \to (Z^*)^n$ moves all non-*null* entries to the left-hand end of the sequence.

$$\mathrm{concentrate}(X)_i = \begin{cases} X_j & \text{if } X_j \neq \mathit{null} \text{ and } i = |\{ X_k \mid X_k \neq \text{null}, 0 \leq k < j \}| \\ \mathit{null} & \text{otherwise} \end{cases}$$

Let $\delta_M : C_M \to C_M$ be the next-configuration function of M. That is, if $C \in C_M$ then $\delta_M(C)$ is the configuration which follows from C according to the program of M. Let M' be the machine obtained from M by changing all read and write instructions to null operations (for example, add zero to a register), and define $\delta'_M = \delta_{M'}$.

Theorem 5.1.1 Suppose a machine can compute the functions:

(i) Merge using resources $R_1(n+m)$.

(ii) Fanout, deliver and concentrate using resources $R_2(n)$.

(iii) Sort using resources $R_3(n)$.

(iv) δ'_M using resources $R_4(P(n))$.

Then it can compute δ_M using resources

$$R_1(P(n)+S(n))+R_2(P(n)+S(n))+R_3(P(n))+R_4(P(n)).$$

Proof. We make the assumption that the model is capable of storing configurations of M in such a manner that they can be dismantled and reassembled using negligible resources. For example, we assume that readrequest,writerequest:$C_M \to (Z^3)^{P(n)}$, data:$C_M \to (Z^3)^{S(n)}$ defined by

$$\text{readrequest}(X,Y)_i = \begin{cases} <(X_i)_0,(X_i)_1,i> & \text{if the } (X_i)_3^{th} \text{ instruction} \\ & \text{of } M \text{ is a read} \\ \textit{null} & \text{otherwise} \end{cases}$$

$$\text{writerequest}(X,Y)_i = \begin{cases} <(X_i)_0,(X_i)_1,(X_i)_2> & \text{if the } (X_i)_3^{th} \text{ instruction} \\ & \text{of } M \text{ is a write} \\ \textit{null} & \text{otherwise} \end{cases}$$

$\text{data}(X,Y) = Y$

can be computed easily.

Let $C \in C_M$ be a configuration of M. The aim is to simulate a single step of M starting in configuration C. Internal computations can be handled directly by application of δ'_M. Read requests are satisfied by computing:

$x = \text{sort}(\text{readrequest}(C))$
$y = \text{merge}(x,\text{data}(C))$

The new processor configurations can then be obtained from sort(concentrate(fanout(y))). For example, suppose in a particular step, processors 0,1,2 and 3 wish to read register 4 of processor 3, register 6 of processor 0, register 7 of processor 1 and register 6 of processor 0 respectively.

Further suppose that the only non-zero registers at that time are register 6 of processor 0, register 9 of processor 1 and register 4 of processor 3, which contain the values 99, 89 and 69 respectively. Table 5.1.1 gives the simulation steps in this case.

Step	Processors				Registers		
	0	1	2	3			
Readrequest	<3,4,0>	<0,6,1>	<1,7,2>	<0,6,3>	<0,6,99>	<1,9,89>	<3,4,69>
Sort	<0,6,1>	<0,6,3>	<1,7,2>	<3,4,0>	<0,6,99>	<1,9,89>	<3,4,69>
Merge	<0,6,99,0>	<0,6,1,1>	<0,6,3,1>	<1,9,89,0>	<1,7,2,1>	<3,4,69,0>	<3,4,0,1>
Fanout	null	<1,99>	<3,99>	null	<2,0>	null	<0,69>
Concentrate	<1,99>	<3,99>	<2,0>	<0,69>	null	null	null
Sort	<0,69>	<1,99>	<2,0>	<3,99>	null	null	null
Result	69	99	0	99			

Table 5.1.1 Simulation of read requests by 4 processors.

Write requests are simulated by computing:

x = sort(writerequest(C))
y = merge(x,data(C))

The new register contents can then be computed from concentrate(deliver(y)). For example, suppose processors 0, 1, 2 and 3 wish to write values 0, 77, 50 and 28 to register 4 of processor 3, register 6 of processor 0, register 7 of processor 1 and register 6 of processor 0 respectively. Further suppose that the current register-contents are exactly the same as in the read-request example above (see table 5.1.1). Table 5.1.2 gives the simulation steps in this case.

Step	Processors				Registers		
	0	1	2	3			
Writerequest	<3,4,0>	<0,6,77>	<1,7,50>	<0,6,28>	<0,6,99>	<1,9,89>	<3,4,69>
Sort	<0,6,28>	<0,6,77>	<1,7,50>	<3,4,0>	<0,6,99>	<1,9,89>	<3,4,69>
Merge	<0,6,99,0>	<0,6,28,1>	<0,6,77,1>	<1,7,50,1>	<1,9,89,0>	<3,4,69,0>	<3,4,0,1>
Deliver	null	<0,6,28>	null	<1,7,50>	<1,9,89>	null	null
Concentrate	<0,6,28>	<1,7,50>	<1,9,89>	null	null	null	null

Table 5.1.2 Simulation of write requests by 4 processors.

□

5.2. A Universal Parallel Machine

Specific instances of theorem 5.1.1 (the simulation of networks or shared-memory machines on other parallel machine models) have appeared many times over in the current literature. Theorem 5.1.1 is a powerful general result. It can be used to:

(1) Provide general communication between the processors of a feasible network (which is equivalent to simulating a network on a feasible network) [45].

(2) Simulate restricted-access networks on a universal network with constant degree and easy-to-compute interconnections [21].

(3) Simulate shared-memory machines on a network with constant degree and easy-to-compute interconnections. This has been observed in the case where no memory access conflicts are allowed [42], or $P(n) = S(n)$ [8].

(4) Remove memory access conflicts from shared-memory machines [72].

(5) Simulate shared memory machines on a variant of the feasible network which uses a small number of "large" processors (with a large amount of local memory and "powerful" instruction set) and a larger number of "small" processors (with a constant amount of local memory and minimal instruction set) [71].

(6) Construct a multi-access memory [4] to provide a practical implementation of a shared-memory machine as a physical device.

(7) Simulate space and reversal bounded Turing machines by width and depth bounded uniform circuits (and vice-versa) [53].

This latter application will be explored further in the next two sections. In this section we will concentrate on the first application.

Corollary 5.2.1 There is a feasible network which can simulate any network of P(n) processors and space S(n), using S(n) processors, the same word-size as the simulated machine, set-up time O(log S(n)) and delay

$$O\left(\frac{\log^2 P(n)}{\log S(n) - \log P(n) + 1} + \log S(n)\right).$$

Proof. A P(n)+S(n) processor feasible network can be used as follows. Note that $S(n) \geq n$, so initially every processor has at most one input symbol. The first P(n) processors are to simulate the processes (keeping only registers r_0, r_1, r_2 of their respective process); the remaining S(n) are to hold the remaining register contents. The set-up time comes from the need to first concentrate the input values (to get rid of any zeros), and route them out to the register-holders using procedures Rank and Concentrate from [45,47]. An additional O(log P(n)) steps are required to broadcast the program of the simulated machine to the first P(n) processors using algorithm 1 of section 4.3. The result then follows from theorems 5.1.1 and 4.4.1, noting that a P(n)+S(n) processor feasible network based on either the shuffle-exchange or cube-connected-cycles can:

(1) Sort P(n) items in time $O\left(\frac{\log^2 P(n)}{\log S(n) - \log P(n) + 1} + \log P(n)\right)$ using the algorithm of [47].

(2) Merge P(n)+S(n) items in time O(log S(n)) by using a Batcher [4] merge (see for example [55,62,66]).

(3) Fan-out P(n)+S(n) items in time O(log S(n)) by using algorithm 3 of section 4.3. Alternatively, procedures Rank, Concentrate and Generalize from [45] can be utilized, as in that reference.

(4) Deliver P(n)+S(n) items in time O(log S(n)) by using procedure Concentrate from [45,47].

(5) Concentrate P(n)+S(n) items in time O(log S(n)) using procedures Rank and Concentrate from [45,47]. □

The time complexity of theorem 5.2.1 is dominated by the cost of sorting the read and write requests. This can be reduced by substituting the sorting algorithm of Ajtai, Komlós and Szemerédi [2] for (1). Although this results in a better asymptotic time-bound, the constant multiple is too large to be of any practical use. The algorithm as presented in [2] has a constant multiple of several million, although this has more recently been reduced to around 1400 by M. S. Paterson. For our purposes, corollary 5.2.1 is to be regarded as superior. There are a number of ways of making the substitution. P(n) values can be sorted in time O(log P(n)) by using O(P(n).log P(n)) processors, giving rise to:

Corollary 5.2.2 There is a feasible network which can simulate any P(n) processor, S(n) = Ω(P(n).log P(n)) space bounded network using O(S(n)) processors, the same word-size and delay O(log S(n)).

Alternatively, P(n) values can be sorted in time O(log P(n).loglog P(n)) on O(P(n)) processors, by pipelining a P(n)/ log P(n) processor sorting network, and merging the log P(n) sorted sequences that result using a Batcher merge. This has also been noted in [40]. More recently, Leighton [41] has discovered an elegant method for sorting P(n) items in time O(log P(n)), using only P(n) processors. Thus we have:

Corollary 5.2.3 There is a feasible network which can simulate any S(n) space-bounded network using O(S(n)) processors, the same word-size and delay O(log S(n)).

What if the processors of the universal machine are allowed to have more than a constant amount of memory? Then:

(1) $O(\log S(n))$ delay, with a more reasonable constant multiple, can be achieved on a probabilistic machine (with overwhelming probability) on $S(n)$ processors by using the sorting algorithm of [57].

(2) The processor-bound in (1) can be reduced to $P(n)$, increasing the delay to $O(\log^2 P(n))$, by using the techniques of Upfal [68].

(3) The bounds of (2) can be achieved on a deterministic machine for the simulation of restricted-access networks [45]. (The delay can also be reduced by the use of the technique of corollary 5.2.3).

Note that the universal machine conserves many of the notions of "reasonableness" mentioned in section 3.3. For example

(1) If the machine being simulated obeys the parallel computation thesis, then so does the universal machine.

(2) If the simulated machine is small and fast (provided $T(n) = \log^{O(1)} P(n)$) the universal machine is small and fast.

(3) Bounds upon word-size are maintained.

5.3. A Hardware Measure

In this section, we attempt to capture the idea of a hardware measure on our network model. The amount of hardware needed to build a universal feasible network is governed by the amount of memory needed, and the complexity of the instruction-set. To simplify matters, we will concentrate on networks with the minimal instruction-set. We claim that space×wordsize is a good hardware measure for such a machine (or indeed, any machine where memory-costs dominate the cost of a processing-unit). In order to justify this claim, we can relate this to the measures of hardware on other popularly-accepted models, whilst maintaining time to within a polynomial.

A *uniform circuit* is an infinite family $C = (C_0, C_1, \ldots)$ of combinational circuits, one for each input size (see, for example [7,13,53,59]). Without loss of generality we assume that the circuits are built using gates which realize functions drawn from the class B_2 of two-input Boolean functions. An input of size n is presented, in some suitably encoded form, to the inputs of C_n. The output of C_n is then taken as the output of C. C is said to have *depth* D(n) if the length of the longest path from an input to an output in C_n is at most D(n), for $n \geq 0$. It has *width* W(n) if C_n has width (as defined in [53]) W(n) and *size* Z(n) if C_n has Z(n) gates. We assume $D(n) = \Omega(\log W(n))$.

The function $f: N^2 \times \{left, right\} \rightarrow N$ where for $n \geq 0$ the j-input of gate $i \geq n$ is connected to the output of gate f(i,n,j), is called the *interconnection function* of C. We assume that gates 0,1,...,n-1 are distinguished gates representing the inputs. The function $g: N^2 \rightarrow B_2$, where for $n \geq 0$ gate $i \geq n$ of C_n is a g(i,n)-gate, is called the *gate function* of C. We insist that the interconnection and gate functions be computable in space O(log Z(n)) by a deterministic Turing machine.

Corollary 5.3.1 Every network with P(n) processors, space S(n), time T(n) and word-size W(n) can be simulated by a uniform circuit of depth $O(T(n) . \log S(n) . \log W(n))$ and width $O(S(n) . W(n))$.

Proof. (Sketch). The circuit consists of T(n) levels, one for each simulated time-step. Each level has P(n) sub-circuits corresponding to a single step of a processor, and a further S(n) sub-circuits carrying register values. Between each level is a circuit for carrying out inter-processor communication, built out of a sorter, merger, concentrator etc. as in corollary 5.2.2. Each processor unit takes as input the program-counter, current values of registers r_0, r_1 and r_2, and incoming values from read requests. It produces outgoing read and write requests, and updated values for the aforementioned program-counter and registers (see figure 5.3.1). These units fit together as in figure 5.3.2.

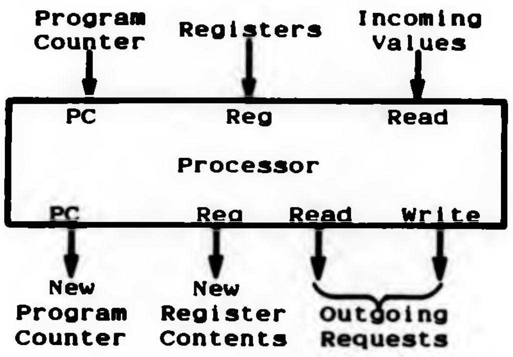


Figure 5.3.1 Block diagram representation of a circuit to compute a single step of a processor.

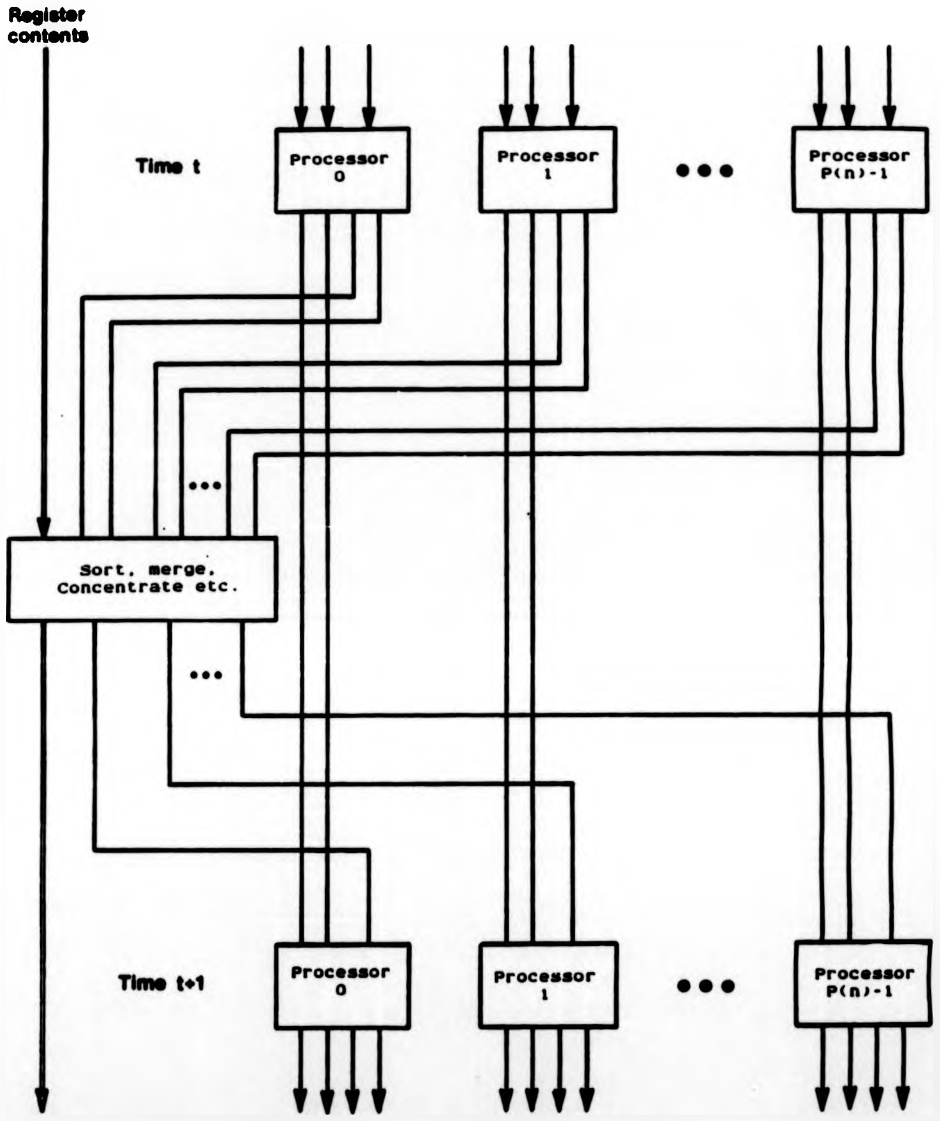


Figure 5.2.2 Block diagram of a circuit to compute a single step of a network.

Each processor-unit has circuitry which

(1) Deals with incoming data which has arrived in response to a read request in the last step.

(2) Performs a single instruction, issuing a read or write request as necessary.

The processor units have width $O(W(n))$ and depth $O(\log W(n))$. The circuits for sort, merge, concentrate etc. have width $O(S(n).W(n))$ and depth $O(\log S(n).\log W(n))$. The register contents have width $O(S(n).W(n))$. Thus the complete circuit has width $O(S(n).W(n))$ and depth $O(T(n).\log S(n).\log W(n))$.□

Note that in the general case, if the internal instructions can be computed by a uniform circuit of depth $d_I(n)$ and width $w_I(n)$, then the above circuit has depth $O(T(n).(\log S(n).\log W(n)+d_I(n)))$ and width $O(S(n).W(n)+P(n).w_I(n))$.

In section 3.3 we saw a number of different ways of characterizing a "reasonable" parallel machine model. For example, the parallel computation thesis states that a parallel machine model is reasonable if time on that model is polynomially related to sequential space. Dymond [16,17] gives an extended version of the parallel computation thesis which takes into account both the time and the amount of hardware used. This can be loosely summed up as follows: time and hardware on any reasonable parallel machine model are simultaneously polynomially related to Turing machine reversals and space respectively (a *reversal* is said to occur when any tape-head changes direction).

This raises an obvious question: when are our network machines a "reasonable" parallel machine model according to the extended parallel computation thesis, given that space×wordsize is taken as a measure of hardware? We find that a $T(n)$ time, $P(n)$ processor bounded network which uses space $S(n)$ and has word-size $W(n)$ obeys the extended parallel computation thesis provided:

(i) Local instructions can be computed by a deterministic Turing machine using space $(W(n).S(n))^{O(1)}$ and $T(n)^{O(1)}$ reversals.

(ii) $P(n) = 2^{T(n)^{O(1)}}$.

Part (i) provides more evidence for the unit-cost hypothesis. Note that the Turing machine is to be given the value of P(n) in binary along with any input of size n.

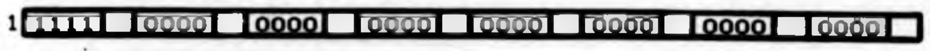
In particular, for a machine with the minimal instruction-set:

Corollary 5.3.2 Every P(n) processor network which runs in time T(n), space S(n) and word-size W(n) can be simulated by a deterministic Turing machine using space $O(S(n).W(n))$ and reversals $O\big(T(n).(\log^2 P(n) + \log S(n))\big)$.

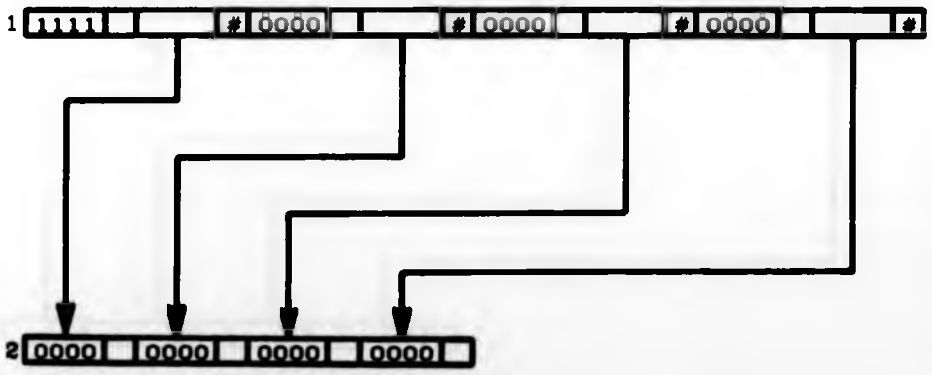
Proof. (Sketch). This result follows from theorem 5.1.1 much in the same manner as corollary 5.2.1, substituting the sorting algorithm of theorem 4.1.5 (Batcher sort) for that of [47]. The composite sub-algorithms used thus have simple modules whose upper dimension is easy to compute.

Consider a simple-ascend class sub-algorithm which ascends to the full value of k, and uses the minimal instruction-set. Suppose the n inputs are initially encoded as binary strings on tape 1 of the Turing machine, each separated by a special blank symbol. The Turing machine computes in k phases (one for each dimension), each of which consists of a constant number of passes over two tapes. The first phase does the following. First, copy every alternate string on to tape 2. In a constant number of left-to-right scans over the tapes, perform the necessary data transfers in dimension 1, and the internal operations. Copy the (updated) strings from tape 2 back to tape 1. The word-size can increase by only a constant, so the overflow from each string can be stored temporarily by using a large tape alphabet, and the tape contents can be moved along as part of the copying process by making extra use of the second tape. (Extra tapes may be necessary for more powerful instruction-sets which increase the word-size more rapidly). This is the end of the first phase. Phase i, $2 \le i \le k$ achieves data transfers in dimension i by similarly copying alternate blocks of 2^{i-1} strings from tape 1 to tape 2, performing the transfers in a constant number of left-to-right scans, and copying the strings back to tape 1.

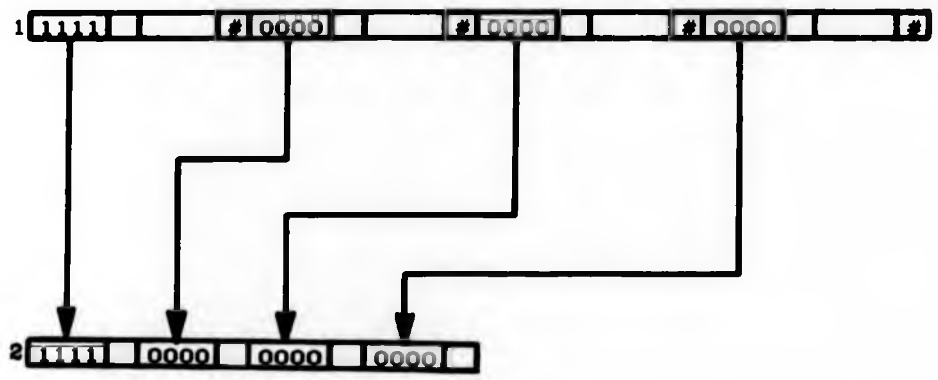
Take for example algorithm 1 of section 4.3, the algorithm to broadcast a single value to all processors. On input



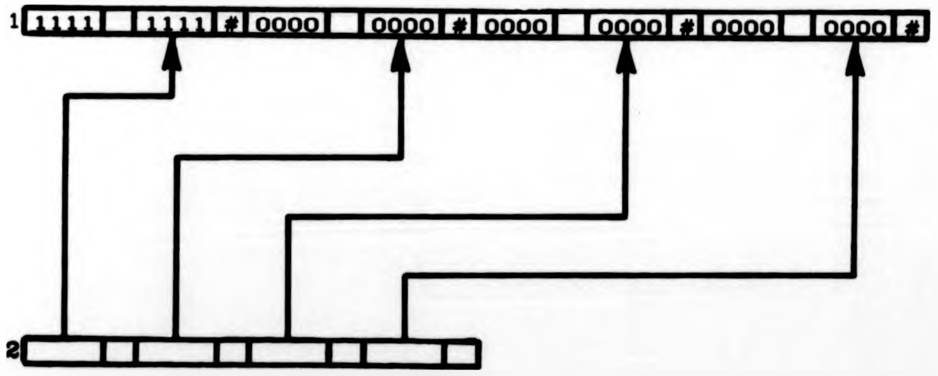
copy every alternate string to tape 2, leaving a special mark on tape 1 at the end of every copied string.



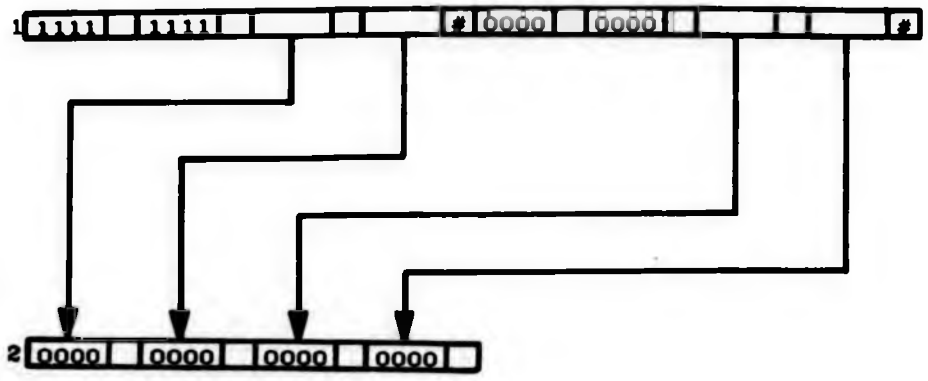
Perform data transfers in dimension 1 in a single left-to-right scan of both tapes.



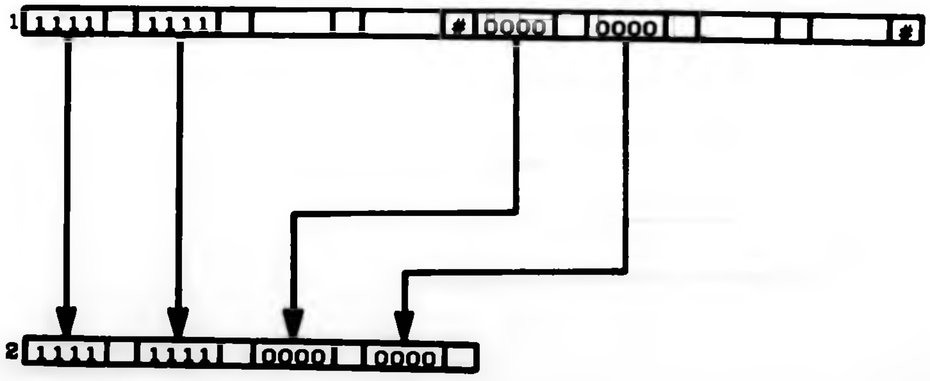
Now copy the strings back to tape 1.



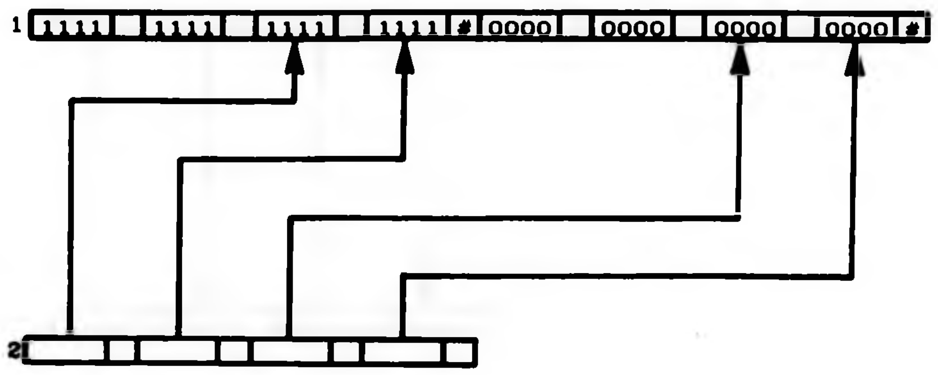
To handle transfers in dimension 2, first copy every alternate block of two strings to tape 2 (again leaving a special mark at the end of every copied block).



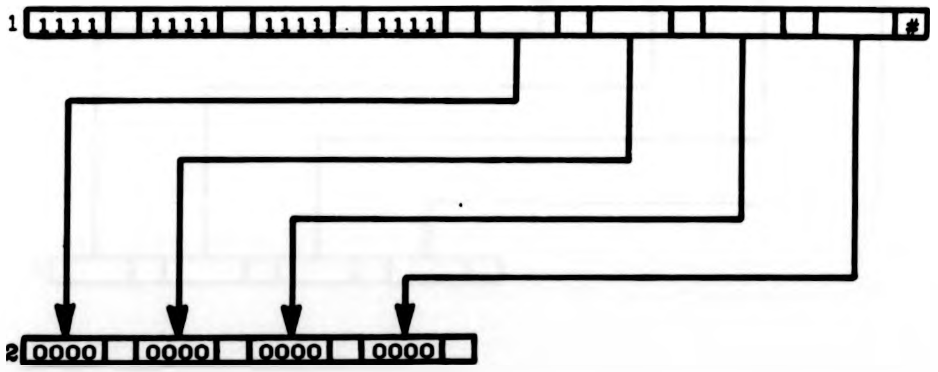
Perform the data transfers,



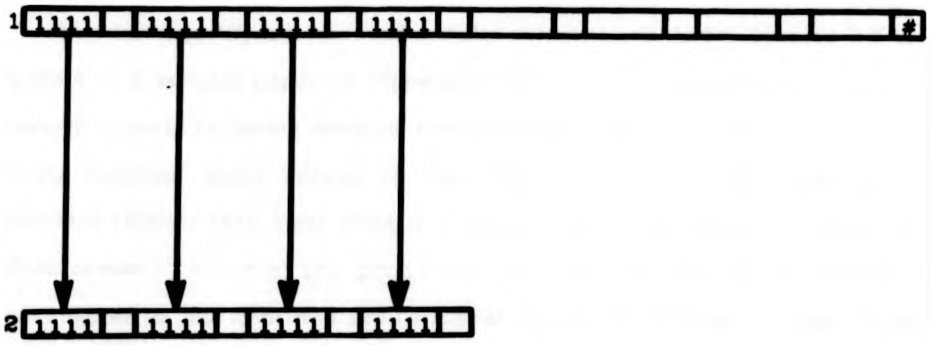
and copy back.



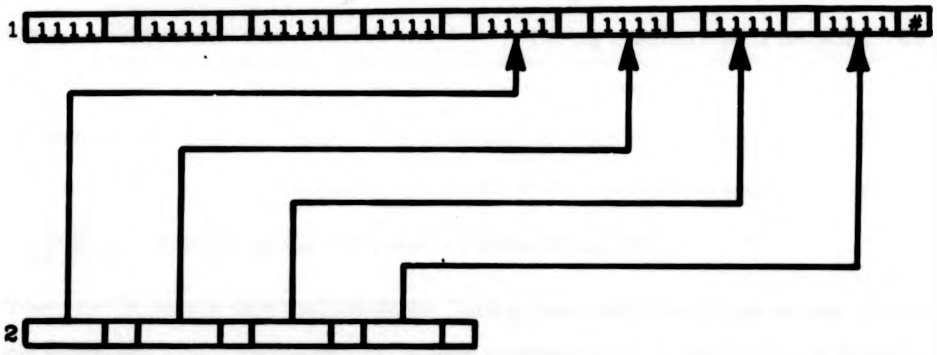
Finally, for dimension 3, first copy across every alternate block of four strings.



Perform transfers.



and copy back.



□

Note that the sorting algorithm of [47] can be used (as in corollary 5.2.1) to give an improved reversals bound, provided the upper dimensions in the simple sub-modules can be computed by a deterministic Turing machine within the stated resources.

5.4. Circuits and Turing Machines

In order to justify his extended parallel computation thesis, Dymond [16,17] appeals to a seminal paper by Pippenger [53] which relates depth and size of uniform circuits to Turing machine reversals and time. Dymond prefers to use Turing machine space instead of time, and circuit width as a measure of hardware (rather than size) since it is a measure of the amount of hardware which comes into play at any given instant in time. We can use the results of this chapter to gain improved simulations of space and reversal bounded Turing machines by uniform circuits.

We follow the general structure of the proof appearing in [53]. Pippenger simulates a Turing machine on an oblivious Turing machine, and then simulates this by a uniform circuit. We will simulate a Turing machine on a network. We can then build a uniform circuit by application of corollary 5.3.1.

Theorem 5.4.1. An $S(n)$ space, $R(n)$ reversal bounded k-tape deterministic Turing machine can be simulated on a network with processors and space $O(\frac{S(n)^k}{\log S(n)})$, time $O(R(n).\log S(n))$ and wordsize $O(\log S(n))$.

Proof. Let M be a k-tape deterministic Turing machine which runs in space $S(n)$ and reversals $R(n)$. Following [53] define a ***phase*** to be all the steps of M from one reversal to the next (the first move is counted as a reversal for this purpose), and a ***situation*** to be the control state and head positions of M. It may be assumed that all transition rules of M which write a new value onto a tape cell also move the head away from that cell. This implies that symbols written during one phase cannot be read until the next. Let $d(n) = 2^{\lceil \log \log S(n) \rceil}$ and call a situation ***special*** if it has at least one head on the $(i.d(n))^{th}$ cell of its tape, for some $i \in N$. Note that there are at most $O(\frac{S(n)^k}{\log S(n)})$ different special situations, and that at most $O(\log S(n))$ steps of M can occur between one special situation and the next.

In order to make the proof more readable, we will present the algorithm on a shared-memory machine. This reinforces the observation in section 3.2 that completely-connected networks are almost identical to shared-memory machines. The simulation proceeds roughly as follows. The tape contents at the start of the current phase, the head directions and the initial situation for the current phase are stored in the global memory. This is easy to do at the start of the initial phase; the algorithm will maintain this information from phase to phase. We reserve one processor (and two global memory locations) for each special situation. The aim is to have these processors confer, via the global memory, and decide which special situations are involved in the current phase. The processors corresponding to these special situations then simultaneously update the tape cell contents in global memory; the final situation (which is detected by an attempted reversal) determines the head directions and the initial situation for the next phase. This proceeds for a total of $R(n)$ phases.

The simulation of a phase is achieved as follows. Processor i handles the i^{th} special situation. Firstly, each processor i computes in parallel the special situation which follows from special situation i, by doing a step-by-step read-only simulation of M on the tape contents in global memory (by "read-only simulation" we mean that the tape-contents are not updated). This value is stored into array element s[i] in global memory. If an illegal situation occurs during this process, or a reversal is detected (determined by examining the head directions for the current phase, which are stored in global memory) then s[i] is set to i. All processors i execute the following code synchronously in parallel. Upon termination, global array element active[i] will be set to true iff special situation i occurs in the current phase. Each processor can determine whether its special situation is the first special situation to occur in the current phase by using a step-wise read-only simulation of M starting at the initial situation of the phase.

```
active[i]:= if (first special situation in this phase) = i
                then true
                else false
for b:=1 to ⌈log S(n)⌉ do
    if active[i] then active[s[i]]:=true
    s[i]:=s[s[i]]
```

Those processors i with active[i] = true can then update the tape contents; the last special situation is readily available (in all entries of s), from which the final situation of the current phase can be determined.

The running time is dominated by O(log S(n)) for each phase. This comes from:

(1) Decoding of PIDs (each of O(log S(n)) bits) into special situations.

(2) Determining the first special situation from the initial situation and the final situation from the last special situation by simulating at most O(log S(n)) steps of M.

(3) Computing the special-situation transition function by simulating O(log S(n)) steps of M.

(4) Computing the active array in O(log S(n)) steps.

(5) Updating the tape contents by simulating O(log S(n)) steps of M.

Repeating this for R(n) phases gives us the required result. □

Corollary 5.4.2. An S(n) space, R(n) reversal bounded deterministic k-tape Turing machine can be simulated by a uniform circuit of depth $O(R(n).\log^2 S(n).\log\log S(n))$ and width $O(S(n)^k)$.

Proof. By theorem 5.4.1 and corollary 5.3.1. □

Corollary 5.4.3 A T(n) time, R(n) reversal bounded deterministic k-tape Turing machine can be simulated by a uniform circuit of depth

$$O(R(n).\log^2 T(n).\log\log T(n))$$

and size

$$O(R(n).T(n)^k.\log^2 T(n).\log\log T(n)).$$

This is a small improvement over the results of Pippenger [53] who obtains depth $O(R(n).\log^4 T(n))$ and size $O(R(n).T(n)^k.\log^4 T(n))$.

Chapter 6
High-Arity Machines

In the general network model as described in section 2.1, a communication line between two processors A and B is made up of two bidirectional links, one under the control of A, and the other under the control of B. We call a processor's links *active* if they are under its control, and *passive* otherwise. The active links of processor A are those which it can use to initiate communication (by executing a read or write instruction), whereas its passive links are used for communication initiated by its neighbours (attempts to read from or write to a register of A).

In section 2.1 we made the assumption that during any time-step, each processor can make use of only one of its active links (albeit a potentially different one at each time-step). In this chapter we extend our network model to give each processor the use of more than one active link simultaneously (and the power to make efficient use of the information thus obtained). We call the number of active links which can be used by a single processor in any time-step the *arity* of the network.

Although machines with non-constant arity have already appeared in the literature, there has so far been no systematic investigation into the extra computing power offered by high-arity instructions. For example, the random routing results of [69,70] initially appeared in a high-arity form, although this has since been redressed [3,5,57,67]. The oblivious lower-bound of [8] is also presented for high-arity machines.

In the first section we present our high-arity model. In the second section we show that machines of arity and degree A(n) are potentially more powerful than those of arity o(A(n)). In particular we are interested in networks with reasonably small arity and degree; more precisely, those with P(n) processors and arity and degree O(log P(n)). Whilst it is apparent from section 2 that these machines are more powerful than those of constant degree, we will show in section 3 that they are not too much more powerful, in the sense that there exists an efficient constant-degree universal machine. Finally, section 4 gives some examples of the speed-ups which can be obtained by increasing arity. A preliminary version of the material contained in the first three sections of this chapter has appeared in [52].

6.1. A High-Arity Model

Our aim is to generalize the network model to give each processor the ability to communicate with asymptotically more than a constant number of its neighbours in unit time, and sufficient power to make good use of this ability. The basic high-arity model is defined in the much the same manner as the constant-arity one of section 2.1, except for the fact that we allow the processors to have instructions which can be simulated in time A(n) by the processors of that section. A(n) is then called the *arity* of the machine. For example, we can replace the example instruction-set of section 2.1 with the following:

(1) $r_i[r_j \leftarrow \text{constant}]$	(block-load constant)
(2) $r_i[r_j \leftarrow r_k]$	(duplicate register)
(3) $r_i[r_j \leftarrow r_k \sim r_l]$	(element-by-element operation)
(4) $r_i[r_j \leftarrow \sim r_k]$	(prefix ~)
(5) $r_i[r_j \leftarrow r_{r_k}]$	(indirect loads)
(6) $r_i[r_{r_j} \leftarrow r_k]$	(indirect stores)
(7) $r_j \leftarrow \text{PID}$	
(8) halt	
(9) **goto** m **if** $r_j > 0$	
(10) $r_i[r_j \leftarrow (r_{r_k}$ **of** $r_l)]$	(write)
(11) $r_i[(r_{r_j}$ **of** $r_k) \leftarrow r_l]$	(read)

Instructions (7-9) are as in section 2. Instructions (1-3,5,6,10,11) have the same effect (in unit time) as the high-level statement:

for m:=0 **to** $r_i - 1$ **do** S

where statement S is respectively

(1) r_{j+m}:=constant.

(2) $r_{j+m} := r_k$.

(3) $r_{j+m} := r_{k+m} \sim r_{l+m}$.

(5) $r_{j+m} := r_{r_{k+m}}$.

(6) $r_{r_{j+m}} := r_{k+m}$.

(10) $r_{j+m} := r_{r_{k+m}}$ **of processor** r_{l+m}

(11) $(r_{r_{j+m}}$ **of processor** $r_{k+m}) := r_{l+m}$

Instruction (4) has the same effect as

```
$r_j := r_k$
for m:=1 to $r_i - 1$ do
    $r_{j+m} := r_{j+m-1} \sim r_{k+m}$
```

In this particular model, a parallel machine has arity A(n) if for all inputs of size $n \geq 0$, the largest value present in register r_i during the execution of instructions of the form (1-6,10,11) is at most A(n).

A fixed-structure variant of this model can be defined by augmenting the processors with an infinite collection of read-only port registers, and interpreting communication instructions after the manner of section 3.1. In section 6.3 we will consider a restricted-access, fixed-structure model. Each processor is augmented with a communication register COM, and communication instructions are restricted to allow reads and writes of those registers only. Instructions (10) and (11) of the example instruction-set are replaced by:

```
(10') $r_i[r_j \leftarrow \text{COM of } p_{r_k}]$   (read)
(11') $r_i[(\text{COM of } p_{r_j}) \leftarrow r_k]$   (write)
```

which have the same effect as the high-level statements:

```
(10') for $m := 0$ to $r_i - 1$ do
        $r_{j+m} := \text{COM of processor } p_{r_{k+m}}$

(11') for $m := 0$ to $r_i - 1$ do
        $(\text{COM of processor } p_{r_{j+m}}) := r_{k+m}$
```

respectively.

6.2. The Computational Power of High-Arity Machines

Some of the power of high-arity machines comes from the fact that they have high degree. It is easy to show that a machine with degree $D(n)$ is asymptotically faster that any machine of degree $o(D(n))$ (with arity kept constant). Consider the problem of broadcasting a single value amongst n processors. More formally, we wish to compute, in parallel, the function $f: N^* \rightarrow N^*$ defined by $f(x_0, \ldots, x_{n-1}) = (y_0, \ldots, y_{n-1})$ where $y_i = x_0$ for $0 \leq i < n$. Suppose $d \geq 3$. The following is an n processor, degree d, arity-1 algorithm for computing f on inputs of size n in time $O(\frac{\log n}{\log d})$. Assuming that initially variable x of processor i contains x_i, the algorithm terminates with variable x of processor i containing

x_0, $0 \le i < P(n)$. The interconnection pattern used is a (d-1)-ary tree. Figure 6.2.1 shows four levels of a D-ary tree, for arbitrary D.

```
b:=1
while b<n do
  x:=x of processor $\left\lfloor \frac{PID-1}{d-1} \right\rfloor$
  b:=b.(d-1)
```

The time-bound attained by the above algorithm is asymptotically optimal. It is easy to see that a degree d machine must take time $\Omega(\frac{\log n}{\log d})$ to compute f, regardless of how complicated its interconnections are, how many processors are used, or what the arity of those processors is. This follows from the observation that there must be a path in the interconnection graph of size n from processor 0 to processor i, $0 \le i < P(n)$. From this we can conclude that a degree D(n), constant arity parallel machine with n processors is asymptotically faster than than any machine of degree o(D(n)). Indeed, the latter machine may even be allowed to have a different non-recursive program for each processor, which may vary with input size.

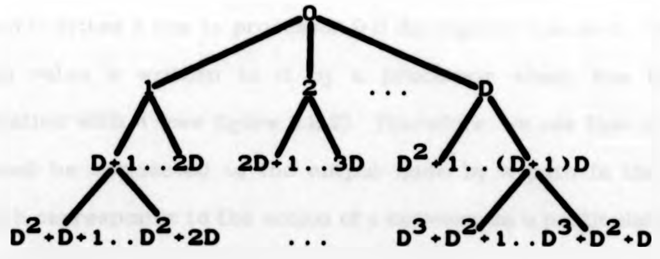


Figure 6.2.1 A D-ary tree.

Furthermore, we wish to show that increasing the arity of a parallel machine increases its power. Unfortunately there are no good lower-bounds available for even constant-arity machines with simultaneous writes (an exception is the recent paper by Wigderson and Vishkin [74], which uses a very restrictive model). Even without simultaneous writes, an argument based on "information-flow" is often quite difficult, for in many cases, the concept of "information" is subtle. Even though a single processor can only receive one communication in each time-step, it may receive it from potentially many different sources, depending on the input values. Information can be encoded both in the value written, and the identity of the source.

Even without simultaneous writes, the situation may not be as easy as it looks. For example, information can even be passed by a processor choosing not to write. Suppose processor A has a value $v \in \{0,1\}$ which it wishes to communicate to processor D. Although A must be connected to D by a path in the interconnection graph of the machine, every subgraph which corresponds to a particular computation may have A and D disconnected, as follows. Two extra processors each initialize a register r to zero. In the first step, A writes a one to register r of processor B if $v = 1$, and a one to register r of processor C if $v = 0$. (Thus register r of processor B holds v, and register r of processor C contains its complement). In the second step, B writes a zero to processor D if its register r is zero, and C writes a one to processor D if its register r is zero. Thus processor D has the value v written to it by a processor which has had no direct communication with A (see figure 6.2.2). Therefore, we see that not every input symbol need be connected to the output node by a path in the computation graph which corresponds to the action of a network on a particular input.

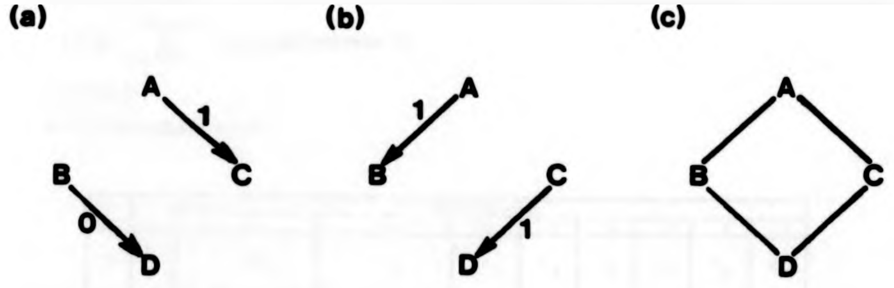


Figure 6.2.2 Communications between processors A, B, C, D, (a) when $v=0$, (b) when $v=1$, and (c) the interconnection graph.

Despite these difficulties, good lower-bounds for some natural problems have been achieved in arity-1 machines without simultaneous writes (see [15,58,65]).

We wish to show that a network with arity and degree $D(n)$ is asymptotically faster than any machine of arity $o(D(n))$, even in the prescence of simultaneous writes. Consider the problem of computing the sum of n integers. More formally define $sum:N^* \rightarrow N$ by

$$sum(x_0, \ldots, x_{n-1}) = \sum_{i=0}^{n-1} x_i.$$

Suppose $d \geq 2$. The following is an n processor algorithm with degree $d+1$ and arity d for computing the sum of n integers in time $O(\frac{\log n}{\log d})$. Initially, we assume that variable x of processor i contains x_i, $0 \leq i < P(n)$. On termination, variable s of processor 0 contains $sum(x_0, \ldots, x_{n-1})$. The interconnection pattern is a d-ary tree. Table 6.2.1 shows a trace of this algorithm for $n=8$, $d=3$.

```
s,b:=x,1
while b < n do
  s:=x+ $\sum_{i=PID.d+1}^{PID.d+d}$ (s of processor i)
  b:=b.d
if PID > 0 then s:=0
```

i	b	Processor							
		0	1	2	3	4	5	6	7
0	1	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
1	3	$x_0+x_1+x_2+x_3$	$x_1+x_4+x_5+x_6$	x_2+x_7	x_3	x_4	x_5	x_6	x_7
2	9	$x_0+\dots+x_7$	$x_1+x_4+x_5+x_6$	x_2+x_7	x_3	x_4	x_5	x_6	x_7
Result		$x_0+\dots+x_7$	0	0	0	0	0	0	0

Table 6.2.1 Trace of summation algorithm on 8 processors of arity 3. Table entry shows the value of b and variable s of each processor after i iterations.

Thus for $D(n) \geq 2$, an n processor, arity D(n), degree D(n)+1 parallel machine can compute the sum of n integers in time $O(\frac{\log n}{\log D(n)})$. This algorithm is asymptotically optimal for all machines of arity O(D(n)). In fact, we will show that an arity D(n) machine must take at least $\left\lceil \frac{\log n}{\log (D(n)+2)} \right\rceil$ steps to sum n integers.

Suppose M is a $P(n) \geq n$ processor parallel machine of arity D(n) which can sum n integers in time T(n), and let $x = \langle x_0,\dots,x_{n-1} \rangle$ be an input string consisting of n symbols, each of which is a non-negative integer. Let G_x be the directed graph with vertices (p,t), $0 \leq p < P(n)$, $0 \leq t \leq T(n)$, and an edge from (p_1,t_1) to (p_2,t_2) if $t_2 = t_1+1$ and either $p_1 = p_2$ or during time-step t_2 of the computation of M on input x, either processor p_2 reads a value from p_1, or p_1 successfully writes a value to p_2. The ith symbol of x is said to be *reachable* if there is a path from vertex (i,0) to vertex (0,T(n)) in G_x. The *reachable string* is the string derived from x by deleting all unreachable symbols. The *unreachable string* is similarly derived by deleting all reachable symbols.

Suppose the values to be added together are all less than N. We claim that (provided N is sufficiently large) there is an input string with all symbols reachable. For a contradiction, suppose that every input has at least one unreachable symbol. Fix a graph G_x and consider the strings y such that $G_x = G_y$. Each reachable symbol of y can take on N possible values, giving a total of N^r possible reachable strings, where $r < n$ is the number of reachable symbols. Further, for each reachable string, the corresponding unreachable string must sum to a fixed value, dependent only on the reachable string in question. This follows because M must give the same result for two inputs y_1, y_2 such that $G_{y_1} = G_{y_2}$ and y_1, y_2 have identical reachable strings. Since $m \geq 1$ non-negative integers can sum to a fixed value at most N^{m-1} times, we see that there are at most N^{n-r-1} unreachable strings which can appear with any reachable string, and thus there are at most N^{n-1} choices of y. That is, each graph G_x can be used for at most N^{n-1} different input strings x.

Let $G(n) = |\{G_x | x \in N^n\}|$. By the pigeonhole principle, at least one graph must be used for at least $N^n / G(n)$ input strings. If N is chosen such that $N > G(n)$ then this value is greater than N^{n-1}, which contradicts the result of the previous paragraph. Thus there must be an input string for which all symbols are reachable. Since for all x, G_x has in-degree $D(n)+2$, this implies that $T(n) \geq \left\lceil \frac{\log n}{\log (D(n)+2)} \right\rceil$. □

Unfortunately, this proof is based heavily on the use of extremely large integers as summands. Indeed, it may be necessary to choose N to be as large as $P(n)^{(D(n)+1).P(n).T(n)}$. Thus:

(1) If we insist that $W(n) = T(n)^{O(1)}$ (which, as we saw in section 2, ensures that the parallel computation thesis holds), then the lower-bound is not valid.

(2) For machines with $W(n) = n^{O(1)}$ (which is a reasonable restriction since it ensures that the input encoding is "concise" in the sense of [22]), the lower-bound holds provided $P(n) = n^{O(1)}$.

(3) If the word-size is arbitrary, the lower-bound holds regardless of arity or number of processors. This is despite the fact that machines with large word-size are (as observed in section 3.5) exceptionally powerful.

6.3. A Constant-Degree Universal Machine

From the results in the previous section, it is apparent that high-arity machines are more powerful than those of constant degree. In this section we propose to show that they are not too much more powerful, in the sense that efficient constant-degree universal machines exist. We will concentrate mainly on a fixed-structure, restricted-access model (see section 6.1). Each dedicated processor will be initialized with the processor-identities of the neighbours of its processes. We shall see that slightly more efficient simulations are made possible by the prescence of this extra information (which cannot be provided in a model with modifiable structure). In a fixed-structure model, it is quite reasonable to expect the user to provide this information (perhaps in the form of an easy-to-compute interconnection function, in which case its resource requirements should be added to the setup time for the universal machine), since it forms part of the specification which would be required by a fabrication device, should the network be realized in hardware.

Theorem 6.3.1 There is a $P(n).D(n)$ processor universal parallel machine which can simulate any $P(n)$ processor machine of arity-and-degree $D(n)$, with delay $O(\log P(n) + D(n))$ and setup time $O(\log^4 P(n) + D(n))$.

Proof. (Outline). Suppose $m = \lceil \log P(n) \rceil$ and $m' = \lceil \log D(n) \rceil$. We describe our algorithm on an $(m+m')$-cube. Let M be a $P(n)$ processor parallel machine with degree and arity $D(n)$. Processor i, $0 \le i < P(n)$ of the universal machine will simulate processor i of M. Let $i[d]$ denote the d^{th} neighbour of processor i of M, in order of ascending PID. For $0 \le i < P(n)$ let W_i be the m'-cube consisting of processors $2^m.k + i$, for $0 \le k < 2^{m'}$, of the universal machine.

As part of the initialization, each processor i $0 \le i < P(n)$ will receive $D(n)$ identification numbers $I^i_{i[d]}$ for $0 \le d < D(n)$ such that:

(1) $0 \le I^i_{i[d]} < D(n)$ for all $0 \le i < P(n)$, $0 \le d < D(n)$, and

(2) For all $0 \le i, i', j < P(n)$, if I^i_j and $I^{i'}_j$ are both defined, and $I^i_j = I^{i'}_j$ then $i = i'$.

In particular, processor i will be the $(I^i_{i[d]})^{th}$ neighbour of processor $i[d]$ in ascending order of PID.

This is achieved as follows. Processor i $0 \le i < P(n)$ prepares $D(n)$ packets $(i[d], i)$, $0 \le d < D(n)$, and scatters them around the $2^{m'} \ge D(n)$ processors of W_i, at most one packet per processor, using algorithm 4 of section 4.3. These packets are then sorted within the $(m+m')$-cube in lexicographic (first-field-first) order. Each processor j, $0 \le j < 2^{m+m'}$ thus receives some packet $(i_j[d], i_j)$. It then sets variable V to $i_j[d]$. Running algorithm 2 of section 4.3 on the $(m+m')$-cube computes the local rank of each processor, which in this case is $I^{i_j}_{i_j[d]}$. Armed with this information, for $0 \le i < P(n)$ the processor in charge of packet $(i[d], i)$ transforms it into $(i, i[d], I^i_{i[d]})$. These packets are sorted back to their respective W_i's, and then gathered back into processor i by reversing the scattering algorithm.

After the initialization phase, each step of the simulation proceeds as follows. First, requests to read communication registers are fulfilled. Processor i $0 \le i < P(n)$ prepares $D(n)$ request packets $(i[d], I_{i[d]}, i)$, $0 \le d < D(n)$. These are scattered at most one-per-processor around the processors of W_i, using algorithm 2. Once this has been carried out, let Π be the permutation which carries packet $(i[d], I_{i[d]}, i)$ to processor $2^m . I_{i[d]} + i[d]$, for $0 \le i < P(n)$. Once Π has been applied, W_i contains the $D(n)$ requests from the neighbours of processor i of M, $0 \le i < P(n)$. Processor i can then fulfill the $D(n)$ requests by broadcasting the contents of the communication register of processor i of M around the processors of W_i using algorithm 1 of section 4.3. The fulfilled requests are routed back to their originating processors by reversing the above process. Processor i of the universal machine can then simulate the internal computation of processor i of M, $0 \le i < P(n)$. Finally, requests to write to communication registers are handled in a similar manner.

Repeating this t times enables us to simulate t steps of M. Note that Π is the same for each step. It is well-known (see theorem 4.1.3) that a fixed permutation can be carried out in time $O(\log P(n))$ by simulating one of Waksman's permutation networks. This requires $O(\log^4 P(n))$ setup time, however (see theorem 4.1.4). The total setup time is thus comprised of:

(1) $O(\log^2 P(n) + D(n))$ to compute the identification numbers $I_{i[d]}$. The $\log^2$ term comes from sorting using a straightforward simulation (see theorem 4.1.5) of one of Batcher's sorting networks. The $D(n)$ term comes from the use of algorithm 4 of section 4.3 to scatter $D(n)$ values.

(2) $O(\log^4 P(n))$ to set up Π.

The delay is comprised of:

(1) $O(D(n))$ to prepare and scatter the request packets,

(2) $O(\log P(n))$ to compute Π,

(3) $O(\log D(n))$ to fulfill the request packets,

which gives us the required result.□

Corollary 6.3.2 There is an $O(P(n).\log P(n))$ processor universal parallel machine which can simulate, with delay $O(\log P(n))$, any $P(n)$ processor parallel machine of arity and degree $O(\log P(n))$.

The proof of theorem 6.3.1 can be modified slightly to give

Theorem 6.3.3 There is a $P(n).D(n)$ processor universal machine which can simulate any $P(n)$ processor, degree $D(n)$, arity-1 machine, with delay $O(\log P(n))$ and setup time $O(\log^4 P(n) + D(n))$.

Note that by making $D(n)$ constant in theorem 6.3.3, and using the processor-saving theorems of section 4.4 we obtain a shorter proof of theorem 8 of [21] (a $P(n)$ processor universal machine which can simulate any $P(n)$ processor, constant degree parallel machine with delay $O(\log P(n))$). In section 7.1 we will see that this result is optimal for this type of literal simulation (although in section 7.2 we will see that more efficient non-literal simulations are possible, using polynomially more processors). In comparison, corollary 6.3.2 uses only $\log P(n)$ times as many processors to achieve the same delay for simulating machines of arity and degree $O(\log P(n))$. Thus in a sense, networks with non-constant arity and degree are not much more powerful than those with constant arity and degree, provided the former are kept to a reasonable level.

In contrast, for a modifiable-structure machine we have:

Theorem 6.3.4 There is an A(n).P(n) processor, constant degree universal parallel machine which can simulate, with delay O(A(n)+log P(n)), any P(n) processor machine of arity A(n).

Proof. Substitute the sorting technique of [41] for the permutation Π in the proof of theorem 6.3.1. □

We should note that the improvement of theorem 6.3.4 stems from the use of the sorting algorithm of [2], so as such, the constant multiples are too large for the result to be of any practical use.

Finally, we turn to the simulation of high-arity machines by feasible networks. We can prove a theorem which is analogous to theorem 5.2.1 by having each arity-A(n) processor represented by a feasible network of A(n) processors. The delay of that theorem is increased by the time for these subnetworks to simulate local instructions. For fairly simple instruction-sets similar to that of section 6.1, this is an increase of only O(log A(n)), which is dominated by routing costs. The use of the techniques of theorem 6.3.1 thus leads to a delay of O(log P(n)) on a feasible network of S(n) processors.

6.4. Examples of High-Arity Algorithms

In section 6.2 we saw some examples of functions which can be computed faster by increasing arity. These examples were simple in nature, but sufficient to meet the lower-bounds of that section. In this section we give two slightly more difficult examples, designed to illustrate that the same speed-ups can be achieved for a much wider range of problems. As we have already observed, time-bounds for some routing problems can be reduced by simply increasing degree (while the arity remains fixed at 1).

Suppose we have n processors, and processor i, $0 \leq i < n$ has a pair of input values (a_i, x_i). Suppose also that for $0 \leq i, j < n$:

(1) if $0 \leq a_i, a_j < n$ and $i \neq j$ then $a_i \neq a_j$, and

(2) $|a_i - a_j| \leq |i - j|$.

For $0 \leq i < n$, we wish to route x_i to processor a_i. In [45,47], Nassimi and Sahni provide a $2^{\lceil \log n \rceil}$ processor algorithm which achieves this process (which they call *concentration)* in time O(log n). We made good use of this algorithm in corollary 5.2.1, when constructing our universal feasible network. Our aim is to produce a $2^{\lceil \log n \rceil}$ processor, degree D(n) algorithm which runs in time $O\left(\frac{\log n}{\log D(n)}\right)$.

Definitions. Suppose $k \geq d \geq 0$. Define the functions $\text{shuffle}_k^d, \text{unshuffle}_k^d : N \to N$ and $\text{exchange}_k^d : N^2 \to N$ by

$$\text{shuffle}_k^d(i) = \left\lfloor \frac{i}{2^{k-d}} \right\rfloor + (i \bmod 2^{k-d}).2^d$$
$$\text{unshuffle}_k^d(i) = \text{shuffle}_k^{k-d}(i)$$
$$\text{exchange}_k^d(i,j) = \left\lfloor \frac{i}{2^d} \right\rfloor .2^d + j$$

Suppose $k \geq d \geq 0$. The *d-way shuffle* graph S_k^d is defined as follows (a similar graph appears in [69]). S_k^d has vertex-set $\{0, 1, \cdots, 2^k - 1\}$, and each vertex v is joined to vertices:

(1) $\text{shuffle}_k^d(v)$

(2) $\text{unshuffle}_k^d(v)$

(3) $\text{exchange}_k^d(v,j)$, $0 \leq j < 2^d$

(4) $\text{shuffle}_k^{k \bmod d}(v)$

(5) $\text{unshuffle}_k^{k \bmod d}(v)$

S_k^d has 2^k vertices and degree at most $2^d + 4$. Figure 6.4.2 shows the d-way shuffle for $d = 2$, $k = 3$.

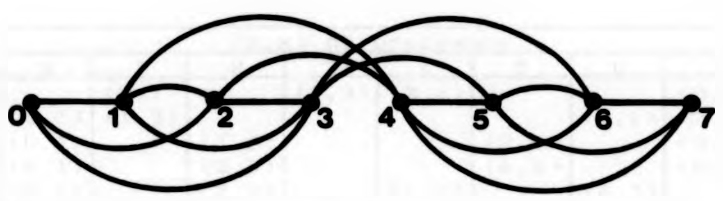


Figure 6.4.1 The 8 vertex 2-way shuffle, S_8^2. Shuffle and unshuffle links appear above the vertices; the remainder are exchange links.

The following is a 2^k processor algorithm based on the d-way shuffle, for concentrating 2^k items in time $O(k/d)$. Suppose initially variables a and x of processor i contain a_i and x_i respectively. Each processor has an extra register r to help with the transfer of data. The latter is achieved with the help of a special procedure, defined as follows.

```
Procedure transfer(p)
  $r := 0$
  if $a < n$ then $(a, x, r)$ of processor p := $(a, x, 1)$
  if $r = 0$ then $a := n$
```

The main body of the algorithm is then:

```
for $b := 1$ to $\lfloor k/d \rfloor$ do
    (1) transfer($\text{exchange}_k^d(\text{PID}, a \bmod 2^d)$)
    (2) transfer($\text{unshuffle}_k^d(\text{PID})$)
    (3) if $a < n$ then $a := \text{unshuffle}_k^d(a)$
(4) transfer($\text{exchange}_k^{k \bmod d}(\text{PID}, a \bmod 2^{k \bmod d})$)
(5) transfer($\text{unshuffle}_k^{k \bmod d}(\text{PID})$)
```

Table 6.4.1 shows a trace of this algorithm for $d = 2$, $k = 3$ and $a_1 = 0$, $a_3 = 1$, $a_4 = 2$, $a_7 = 3$ and $x_i = i$, $0 \le i < 7$, with all other a values equal to 8. Those processors with a-values equal to 8 have an empty entry in the table.

Step	(a,x) of processor							
	0	1	2	3	4	5	6	7
Input		(0,1)		(1,3)	(2,4)			(3,7)
(1)	(0,1)	(1,3)					(2,4)	(3,7)
(2)	(0,1)		(1,3)			(2,4)		(3,7)
(3)	(0,1)		(2,3)			(4,4)		(6,7)
(4)	(0,1)		(2,3)		(4,4)		(6,7)	
(5)	(0,1)	(2,3)	(4,4)	(6,7)				
Result	1	3	4	7				

Table 6.4.1 Trace of concentration algorithm for $d = 2$, $k = 3$. Table entry shows the values of variables a and x of each processor initially (at input) and after each labelled step of the algorithm.

The correctness of the algorithm follows from exactly the same argument as used by Nassimi and Sahni in [47]; essentially, every one of our parallel data transfers achieves the same result as d of theirs.

In section 6.2 we gave an arity D(n) algorithm for summing n items in time $O(\frac{\log n}{\log D(n)})$. In fact, we can compute the parallel prefix sum in asymptotically the same amount of time. Let "~" denote an arbitrary associative binary operation. Let prefix:$N^* \to N^*$ be defined by prefix$(x_0, \ldots, x_{n-1}) = (y_0, \ldots, y_{n-1})$ where $y_0 = x_0$ and for $1 \leq i < n$, $y_i = y_{i-1} \sim x_i$. Then the parallel prefix problem is the problem of computing the function prefix using a parallel machine whose processors can compute "~" in unit time. Nassimi and Sahni [45] give a $2^{\lceil \log n \rceil}$ processor algorithm (which they call Rank, using the operation of integer addition for "~") for the parallel prefix problem, which runs in time O(log n) and has degree 3 and arity 1.

Ladner and Fischer [38] provide a parallel prefix *circuit* which has depth O(log n)) and uses O(n) two-input "~"-gates. An easy generalization gives us a parallel prefix circuit of depth $O(\frac{\log n}{\log d})$, using O(n) d-input "~"-gates. Let $P_d(n)$

denote an n-input prefix circuit built from gates of arity d. Figure 6.4.2 shows a recursive construction of $P_d(n)$ from $P_d(\lceil n/d \rceil - 1)$.

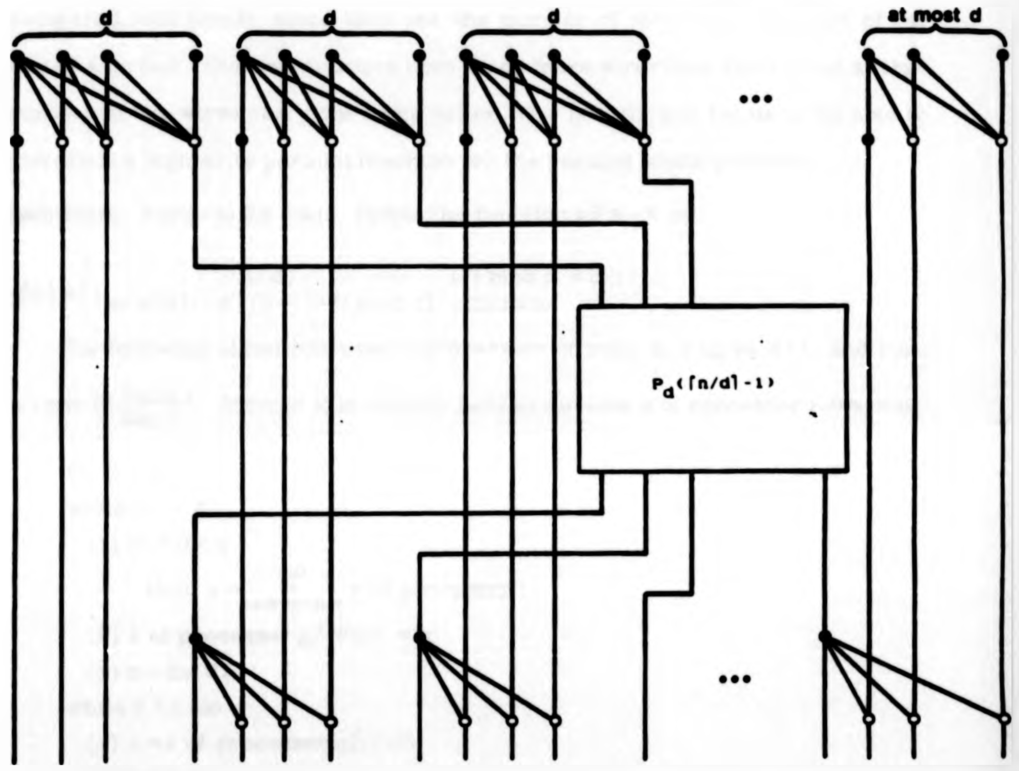


Figure 6.4.2 Recursive construction of an n-input prefix circuit from one with $\lceil n/d \rceil - 1$ inputs. Solid dots are connections, circles denote "~"-gates.

Let $T_d(n)$ and $Z_d(n)$ denote the depth and number of "~"-gates in an arity-d prefix circuit, respectively. Clearly $T_d(n) \le T_d(\lceil n/d \rceil - 1) + 2$, and thus $T_d(n) = O(\frac{\log n}{\log d})$. Also, $Z_d(n) \le (n - \lceil n/d \rceil) + Z_d(\lceil n/d \rceil - 1) + (n - \lceil n/d \rceil - d + 1)$, from which we can conclude that $Z_d(n) \le 2(n-1) - (d-1).\frac{\log n}{\log d}$, and so $Z_d(n) = O(n)$ as claimed. Prefix circuits using gates with unbounded fan-in have been studied in detail by Chandra, Fortune and Lipton [10,11]. Note that our results cannot be compared with theirs, since they use the number of wires as a measure of the size of a circuit. Our construction uses many more wires than theirs, but a large number of the wires carry the same value. This is sufficient for us to be able to construct a high-arity parallel machine for the parallel prefix problem.

Definition. Suppose $2 \le d \le n$. Define the function $g_n^d : N \to N$ by

$$g_n^d(i) = \begin{cases} \lfloor i/d \rfloor & \text{if } i \bmod d = d\text{-}1 \\ \lceil n/d \rceil + \lfloor i/d \rfloor.(d-1) + (i \bmod d) & \text{otherwise} \end{cases}$$

The following algorithm uses n processors of arity d, degree d+1, and runs in time $O(\frac{\log n}{\log d})$. Assume x_i is initially held in variable x of processor i, $0 \le i < n$.

```
(0) $b := n$
**while** $b > 1$ **do**
  (1) **if** $PID < b$
        **then** $x := \overset{PID}{\underset{i=\lfloor PID/d \rfloor d}{\sim}}$ x **of processor** i
  (2) x **of processor** $g_n^d(PID) := x$
  (3) $b := \lfloor b/d \rfloor$
**while** $b < n$ **do**
  (4) $x := x$ **of processor** $g_n^d(PID)$
  (5) $b := b.d$
  (6) **if** $(PID < b)$ and $(PID \bmod d \ne d-1)$ and $(\lfloor PID/d \rfloor > 0)$
        **then** $x := x \sim x$ **of processor** $(\lfloor PID/d \rfloor.d - 1)$
```

Table 6.4.2 shows a trace of this algorithm for $n = 8$ and $d = 3$, and figure 6.4.3 shows the interconnection graph used by the algorithm for those values.

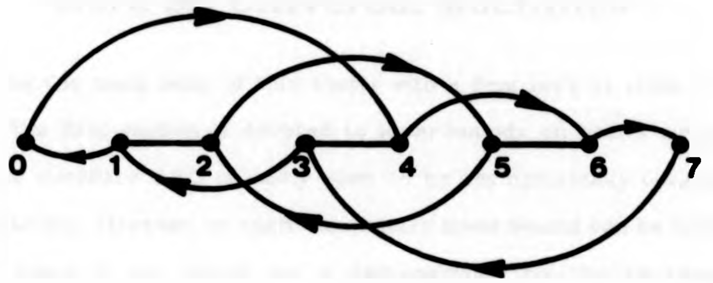


Figure 6.4.3 Interconnection pattern used by the parallel prefix algorithm for $n=8$, $d=3$. The arrows point from vertex i to vertex $g_n^d(i)$, $0 \le i < 8$.

Step	b	Processors							
		0	1	2	3	4	5	6	7
(0)	8	[0,0]	[1,1]	[2,2]	[3,3]	[4,4]	[5,5]	[6,6]	[7,7]
(1)	8	[0,0]	[0,1]	[2,2]	[2,3]	[4,4]	[4,5]	[6,6]	[6,7]
(2)	8	[0,1]	[2,3]	[4,5]	[6,7]	[0,0]	[2,2]	[4,4]	[6,6]
(3)	4	[0,1]	[2,3]	[4,5]	[6,7]	[0,0]	[2,2]	[4,4]	[6,6]
(1)	4	[0,1]	[0,3]	[4,5]	[4,7]	[0,0]	[2,2]	[4,4]	[6,6]
(2)	4	[0,3]	[4,7]	[2,2]	[6,6]	[0,1]	[4,5]	[0,0]	[4,4]
(2)	2	[0,3]	[4,7]	[2,2]	[6,6]	[0,1]	[4,5]	[0,0]	[4,4]
(1)	2	[0,3]	[0,7]	[2,2]	[6,6]	[0,1]	[4,5]	[0,0]	[4,4]
(2)	2	[0,7]	[6,6]	[4,5]	[4,4]	[0,3]	[2,2]	[0,1]	[0,0]
(3)	1	[0,7]	[6,6]	[4,5]	[4,4]	[0,3]	[2,2]	[0,1]	[0,0]
(4)	1	[0,3]	[0,7]	[2,2]	[6,6]	[0,1]	[4,5]	[0,0]	[4,4]
(5)	2	[0,3]	[0,7]	[2,2]	[6,6]	[0,1]	[4,5]	[0,0]	[4,4]
(6)	2	[0,3]	[0,7]	[2,2]	[6,6]	[0,1]	[4,5]	[0,0]	[4,4]
(4)	2	[0,1]	[0,3]	[4,5]	[0,7]	[0,0]	[2,2]	[4,4]	[6,6]
(5)	4	[0,1]	[0,3]	[4,5]	[0,7]	[0,0]	[2,2]	[4,4]	[6,6]
(6)	4	[0,1]	[0,3]	[0,5]	[0,7]	[0,0]	[2,2]	[4,4]	[6,6]
(4)	4	[0,0]	[0,1]	[2,2]	[0,3]	[4,4]	[0,5]	[6,6]	[0,7]
(5)	8	[0,0]	[0,1]	[2,2]	[0,3]	[4,4]	[0,5]	[6,6]	[0,7]
(6)	8	[0,0]	[0,1]	[0,2]	[0,3]	[0,4]	[0,5]	[0,6]	[0,7]

Table 6.4.2 Trace of the parallel prefix algorithm for $n=8$, $d=3$. Table entry shows contents of variable x of each processor after each labelled step of the algorithm. [a,b] denotes $\overset{b}{\underset{i=a}{\sim}}$ (x of processor i).

Chapter 7
More on Universal Machines

We close the main body of this thesis with a final look at some universal networks. The first section is devoted to lower-bounds on literal simulations. The delay of corollary 5.2.3 is easily seen to be asymptotically optimal for a literal simulation. However, no such elementary lower-bound can be found for a simulation which is not literal, as is demonstrated by the existence of a nondeterministic universal machine which has constant average delay. We find that the delay of theorem 6.3.1 is optimal for a strongly-literal simulation of degree-3, arity-1 networks.

In the second section we find that the latter lower-bound can be beaten by a non-literal simulation, by giving a simplified presentation of a result of Meyer auf der Heide [33]. The third section considers oblivious universal machines. A literal simulation is said to be *oblivious* (after Borodin and Hopcroft [8]) if the routes taken by data packets sent in response to read or write requests depend only upon their respective sources and destinations. By extending the work of Borodin and Hopcroft [8] and Lang [39] we obtain asymptotically matching upper and lower-bounds of $\Theta(\frac{P(n)}{\sqrt{P'(n)}}+\log P(n))$ for the delay required for an oblivious simulation of a P(n) processor network on a P'(n) processor, constant-degree universal machine.

7.1. Some Lower Bounds

In section 5.2 we saw several examples of an S(n)-processor feasible network which can perform an O(log S(n)) delay literal simulation of any S(n) space-bounded network (see section 3.4 for definitions). It is easy to see that this delay is optimal for a literal simulation. For suppose $T:N \rightarrow N$ is such that

$T(n) \leq n$. Consider the n-processor machine with the following program, where x of processor i is initially the ith piece of input, $0 \leq i < n$.

```
y:=x
for i:=0 to T(n)-1 do
  y:=y+(x of processor i)
```

M runs in time $O(T(n))$, yet every constant-degree universal machine must take time $\Omega(T(n).\log n)$ to perform a literal simulation of M (no matter how many processors are available), even if the universal machine is allowed to have more than a constant number of registers per processor. For if there are at most $O(\frac{n}{\log n})$ dedicated processors then one processor must be looking after $\Omega(\log n)$ registers of M, which requires time $\Omega(\log n)$ to keep up to date (assuming the universal machine has asymptotically the same arity as the simulated machine). Otherwise, since the simulation is literal, there is ample opportunity for the contents of the requisite register to be broadcast to the $\Omega(\frac{n}{\log n})$ other dedicated processors during each iteration, which takes time $\Omega(\log n)$ on a constant-degree machine (see section 6.2).

In section 5.2 we also saw that a P(n) processor universal machine can achieve delay $O(\log P(n))$ when simulating a P(n) processor restricted-access network. Machine M above also serves to give us a matching lower-bound in this case. Note that these lower-bounds rely on two important facts, the limited data-carrying capacity of constant-degree networks, and the fact that a literal simulation creates a large amount of traffic. If we relax the requirement that the simulation be literal, then no such simple lower-bound technique is available. For example a nondeterministic universal machine can achieve a constant average delay.

We define a nondeterministic network similarly to the deterministic model of section 2.1, with the following modifications:

(1) Two extra instructions are allowed.

(a) $r_i \leftarrow$ *random* (r_j), and

(b) fail.

The former assigns to register r_i a value between 0 and the contents of r_j, and is called a *guess*. The latter is a special kind of halt instruction. A processor which has executed it is said to have *failed*.

(2) A computation is said to *succeed* if no processor has failed. A nondeterministic parallel machine M is said to *compute* a relation $R \subseteq N^* \times N^*$ if for every input x, $<x,y> \in R$ iff there is a sequence of guessed values such that M succeeds and produces output y. Resources are defined in the obvious manner.

Theorem 7.1.1 There is a P(n).log P(n) processor, constant-degree nondeterministic universal parallel machine which can simulate any T(n) time, P(n) processor bounded nondeterministic restricted-access network in time O(T(n)+log P(n)).

Proof. Suppose M is a T(n) time, P(n) processor bounded restricted-access network. For the present, assume $T(n) = \Omega(\log P(n))$. Fix n, and let $m = \lceil \log P(n) \rceil$, $m' = \lceil \log m \rceil$. We will describe our algorithm on an (m+m'+2)-cube. Processor i of the universal machine will simulate process i, $0 \le i < P(n)$. The algorithm consists of $\left\lceil \frac{T(n)}{m} \right\rceil$ phases, each of which corresponds to m steps of M.

The first phase proceeds as follows. Suppose at the t^th^ step of M, $1 \le t \le m$, process i wishes to read the communication register of process j_i^t. Instead of obtaining the correct value from processor j_i^t, it nondeterministically guesses some value d_t which it uses instead, having recorded it, along with the value j_i^t, for later verification. The m values c_t, $1 \le t \le m$, where c_t denotes the contents of the communication register of processor i at time t, are also recorded. The effects of possible write-attempts are also guessed and recorded in a similar manner

For each i such that $0 \leq i < P(n)$ let W_i be the $(m'+2)$-cube consisting of processors $2^m.j+i$, $0 \leq j < 2^{m'+2}$. Having simulated m steps with guessed data values, the verification procedure is as follows. Processor i, $0 \leq i < P(n)$ prepares m read-request packets (j_i^t, t), each being a request for the contents of the communication register of process j_i^t at time t. It also prepares m data packets (i, t, c_t), and similarly m write-request packets. These are scattered around the $2^{m'+2} \geq 4m$ processors of W_i, at most one packet per processor, using algorithm 4 of section 4.3. The request packets are fulfilled using the techniques of theorem 5.1.1. The guessed data values are then compared to the fulfilled requests, and any processor which detects a discrepancy fails immediately.

Thus m steps of M can be simulated in time $O(\log P(n)+m) = O(m)$. Note that $O(P(n).\log P(n))$ items can be sorted in time $O(\log P(n))$ using $O(P(n).\log P(n))$ processors by guessing a set of switch positions of the Waksman permutation network (see theorem 4.1.3), and verifying afterwards that the permuted values are in sorted order. By repeating this for $\left\lceil \frac{T(n)}{m} \right\rceil$ phases we are able to simulate T(n) steps of M in time $O(T(n))$ as required. A set-up time of $O(\log P(n))$ is required to broadcast the program of the simulated machine, using algorithm 1 of section 4.3. This extra term in the time-bound also takes care of the case when $T(n) = o(\log P(n))$. □

In section 6.3 we saw a very special kind of literal simulation of a P(n) processor, constant-degree, restricted-access network M on a universal machine U. This had the property that

(1) Each *processor* i of M has a dedicated processor d_i in U.

(2) This dedicated processor looks after *all* registers of processor i of M.

(3) $d_i \neq d_j$ for $i \neq j$.

(4) The initial dedicated-processor assignment is the same for all simulated machines.

(5) The dedicated processor assignment does not change with time.

Under these very strict conditions, a delay of O(log P(n)) was achieved using only P(n) processors. We will call simulations with property (5) ***strongly-literal***. Meyer auf der Heide [31] has shown that the above delay is optimal for a strongly-literal simulation of a constant-degree, restricted-access network. We can strengthen this to show that the delay is optimal even for the simulation of networks with degree 3.

Theorem 7.1.2 A strongly-literal universal parallel machine with $P(n)^{aP(n)}$ processors, where $a < \frac{1}{2}$, and degree D(n) must have delay $\Omega(\frac{\log P(n)}{\log D(n)})$ when simulating a P(n) processor, degree-3 network.

Proof. Suppose we have an m-processor, degree-d universal parallel machine which can carry out a strongly literal simulation of any degree-3, p-processor network with delay k. We will show that $k = \Omega(\frac{\log p}{\log d})$ when simulating a special kind of machine whose interconnection pattern is a degree-3 graph called a ***matched-cycle***. A p-vertex matched-cycle has vertex-set $\{0,1, \cdots ,p-1\}$ and

(1) Vertex v is joined to vertices $(v\pm1)$ mod p (cycle links).

(2) The remaining edges form a graph of degree 1 (that is, they constitute part of a matching).

Let M be a network with one register per processor, whose interconnection pattern is a matched cycle. Each processor i of M, $0 \leq i < p$ is assigned a dedicated processor d_i in the universal machine. Without loss of generality we will assume that each processor of the universal machine is to be assigned to at most one processor of M (for each multiply-assigned dedicated processor of U can be replaced by a ring of distinct dedicated processors, without disturbing the time or processor bounds in the statement of the theorem). Let

$N =$ the number of matched-cycle graphs,

$N_1 =$ the number of dedicated processor assignments which work for at least one matched-cycle graph, and

$N_2 =$ the maximum number of matched-cycles for which any given assignment can be used.

Claim 1. $N \geq \frac{p!}{2^{\frac{3p}{2}}(\frac{p}{2})!}$

Without loss of generality suppose p is even. Then there are $(p-1).(p-3).(p-5)\ldots1 = \frac{p!}{2^{\frac{p}{2}}(\frac{p}{2})!}$ matchings on p vertices. At most 2^p of these matchings can give rise to the same matched-cycle (by filling in the missing cycle edges), so there are at least $\frac{p!}{2^{\frac{3p}{2}}(\frac{p}{2})!}$ matched-cycles.

Claim 2. $N_1 \leq m.d^{k(p-1)}$.

If a particular processor assignment is to work for a matched-cycle, then processor d_i must be at distance at most k from $d_{(i+1) \bmod n}$ in the interconnection pattern of the universal machine, $0 \leq i < p$. Thus there are m choices for d_0, but at most d^k choices for d_1, and similarly d^k choices for each of $d_2, d_3, \ldots, d_{p-1}$.

Claim 3. $N_2 \leq d^{kp}$.

Fix a processor assignment. Consider the machines M for which that processor assignment works. Each processor i of M can be adjacent to the (at most) d^k

processors j such that d_i and d_j are at distance at most k in the interconnection pattern of the universal machine. Thus each vertex in the interconnection pattern of M can be adjacent to at most d^k other vertices via a matching link.

If the universal machine is to simulate all matched-cycles within the stated resource-bounds, then we must have $N_1.N_2 \geq N$. Thus

$$\frac{p!}{2^{\frac{3p}{2}}(\frac{p}{2})!} \leq m.d^{k(2p-1)}$$

$$\text{i.e. } \frac{p}{2}\log p < k(2p-1).\log d + \log m + O(p)$$

Hence if $m < p^{\alpha p}$ for all $\alpha < \frac{1}{2}$, we see that $k = \Omega(\frac{\log p}{\log d})$. □

Thus a constant-degree universal machine must have delay $\Omega(\log P(n))$. Here is yet another approach to the unit-cost hypothesis. It is valid to charge one time-step for an internal computation which takes time $O(\log P(n))$ on the instruction-set of the universal machine.

7.2. A Non-Literal Simulation

In section 7.1 we saw an $\Omega(\log P(n))$ lower-bound on the delay of a strongly-literal simulation of a P(n) processor, constant-degree restricted-access network. Here we will see that relaxing the literalness condition allows a more efficient simulation of fixed-structure machines. In a literal simulation there is ample opportunity during the simulation of a single step for the data to be routed from the dedicated processors in response to read or write requests. In a non-literal (but step-wise) simulation, this information may start out from the dedicated processors at an earlier point in time, being kept up-to-date along the way by auxiliary processors. Using this technique, Meyer auf der Heide [33] obtains a constant delay (on average) for the simulation of constant-degree, fixed-structure, restricted-access machines. The following is a much-simplified presentation.

Theorem 7.2.1 There is a constant-degree universal machine with $P(n)^{1+\epsilon}$ processors for any $\epsilon > 0$ which can simulate any $P(n)$ processor, $T(n)$ time-bounded, constant-degree, fixed-structure, restricted-access network in time $O(T(n)+\log^4 P(n))$.

Proof. (Sketch). Suppose the machine to be simulated has degree d. Without loss of generality we can assume that it communicates by reads alone. The universal machine has $P(n)$ dedicated processors, one for each process. Each dedicated processor is the root of a complete binary tree of depth $t.\lceil \log d \rceil$, where $t > 0$ is some value to be determined later. Vertices at depth $i.\lceil \log d \rceil$, $0 \le i \le t$ are said to be on the i^{th} *level*. The dedicated processors are thus on the 0^{th} level.

The simulation will proceed in $\lceil T(n)/t \rceil$ phases, each corresponding to t steps of the simulated machine. The trees will be initialized so that each processor on the i^{th} level will be attempting to simulate one of the processes which are adjacent to the process of its predecessor on the $(i-1)^{st}$ level. Each process thus has many processors attempting to simulate it. A request from a process in a processor on the i^{th} level, $0 \le i < t$, to read the communication register of one of its neighbours is passed on to whichever $(i+1)^{st}$-level successor of that processor is attempting to simulate that neighbour. A request by a process in a processor on the t^{th} level to read the communication register of one of its neighbours is ignored.

Thus after i steps have been simulated, the processors on level $t-i+1$ have probably been led astray in their simulation of a process by being misinformed by processors on the next level. All other processors have simulated correctly. After t steps, only the dedicated processors can be guaranteed to have not deviated. This part of the simulation takes $O(t)$ steps.

Meanwhile, the dedicated processors have been saving the communication register contents of their processes at each of the t simulated time-steps. These t values are to be routed to the processors on all levels which are attempting to simulate the same process. Armed with this information, these processors can re-compute the last t steps internally, and get back to a correct state. The trees are then ready to simulate another t steps without further initialization.

Suppose all processors of the trees are at the head of a distinct sub-cube of $2^{\lceil \log t \rceil}$ processors, and that further edges are added to make the whole structure into a multidimensional cube (with embedded trees) of $2^{\lceil \log t \rceil}.P(n).d^t$ processors. The correction stage can be carried out by having each level-i processor, $1 \le i < t$, prepare t requests for the correct communication register contents at each of the t steps of the phase. The dedicated processors prepare t packets which provide this information. We then

(1) Scatter them around the sub-cubes in time O(t) using algorithm 4 of section 4.3.

(2) Permute the requests and data into sorted order. Note that the permutation is the same for each phase, so theorem 4.1.3 can be applied to give time $O(t+\log P(n))$.

(3) The techniques of corollary 5.2.1 are then used to satisfy these requests in time $O(t+\log P(n))$.

(4) The satisfied requests are gathered back by reversing (1) and (2).

This gives a time-bound of $O(t+\log P(n))$ to simulate t steps. A total of $O(P(n).t.d^t)$ processors are needed. Choosing $t = \varepsilon.\log_d P(n) - \log_d \log_d P(n)$ for some $\varepsilon > 0$ gives a constant average delay using $O(P(n)^{1+\varepsilon})$ processors. The set-up time consists of:

(a) Time to assign processes to processors at each level of the trees.

(b) Initialization of the permutation used to sort the requests in part (2) above.

(c) Distribution of inputs, outputs and the program of the simulated machine.

The process in (a) can be achieved level-by-level starting at the dedicated processors, at a cost of $O(\log^2 P(n))$ per level for routing the identities of the new processes from the dedicated processors by use of sorting. This cost of $O(\log^3 P(n))$, and the cost of (c), is dominated by the $O(\log^4 P(n))$ required in theorem 4.1.4 for (b).

All algorithms which use the cube-part of the interconnection pattern are composite, and thus the interconnection pattern can be thought of as a shuffle-exchange or cube-connected-cycles with embedded trees, of degree 6. □

7.3. Oblivious Simulations

We complete this chapter by considering a very strict form of a strongly literal simulation of a restricted-access network, which we shall call *oblivious*. Consider a single step of a strongly literal simulation with dedicated processors d_i. If process i wishes to read the contents of the communication register of process j, then during this simulated time-step the required value can be provided by dedicated processor d_j and routed to d_i. (Similarly, if process i wishes to write into the communication register of process j, the value can be routed from d_i to d_j). If the routes taken by these data items depend solely on the source d_j and the destination d_i (respectively d_i and d_j in the write-mode case), then the strongly literal simulation is said to be *oblivious*. If in addition the next step in the route depends solely on the current location of the data packet and the eventual destination, then it is said to be *source-oblivious*.

The following lower bound is a generalization of theorem 1 of [8].

Theorem 7.3.1 An oblivious simulation of a P(n) processor parallel machine on a constant-degree, P'(n) processor universal parallel machine must have delay $\Omega(\frac{P(n)}{\sqrt{P'(n)}}+\log P(n))$.

Proof. Fix n, and let $p = P(n)$, $p' = P'(n)$. Suppose the universal machine has degree d, dedicated processors d_i, $0 \le i < p$, and interconnection graph G.

For $0 \le i,j < p$ let $R_{j,i}$ be the path in G corresponding to the route taken by a data packet sent from processor d_j to processor d_i of the universal machine in response to a request by process i to read the communication register of process j. Since the simulation is to be oblivious, these paths are invariant with time. Note that a path may consist of a single vertex (in the case where $d_i = d_j$), and that two paths may coincide along part, or even all, of their length. For $0 \le i < p$ let G_i be the graph obtained from G by removing all edges which do not lie on some route $R_{j,i}$, for $0 \le j < p$.

Suppose $k \ge 0$. For each i, $0 \le i < p$ construct a set of vertices V_i from the vertices of G_i as follows. Initially V_i consists solely of vertex d_i (the destination of the routes which comprise G_i). Repeat the following until no new vertices are added: if $|\{j \mid v \text{ lies on } R_{j,i}\}| \ge k$ (i.e. v lies on at least k routes in G_i) and v is adjacent to some vertex $v' \in V_i$ in G_i, then add v to V_i. Thus V_i consists of the largest set of vertices of G_i, clustered around the destination d_i, which are on k or more routes.

Let $T_i = |V_i|$, and $\bar{V}_i$ denote the set of vertices of G not in V_i. Let q be the maximum number of processes to be simulated by any dedicated processor (i.e. $q = \max_{0 \le i < p} |\{j \mid 0 \le j < p \text{ and } d_i = d_j\}|$). Then at most $T_i.q$ routes of G_i start from vertices in V_i, so at least $p - T_i.q$ start from $\bar{V}_i$. In order to get from the vertices of $\bar{V}_i$ to vertex i, these must pass through the vertices of $\bar{V}_i$ which are adjacent to vertices of V_i in G_i. By the definition of V_i, each of these can carry at most $k-1$ routes of G_i. Hence there must be at least $\frac{p-T_i.q}{k-1}$ such vertices. Furthermore, since G has degree d, there can be at most $T_i.(d-1)$ vertices of $\bar{V}_i$ which are adjacent to vertices if V_i in G. Hence

$$T_i.(d-1) \geq \frac{p-T_i.q}{k-1}$$

$$\text{i.e. } T_i \geq \frac{p}{(d-1)(k-1)+q}$$

Let $T = \min_{0 \leq i < p} T_i$, and for each vertex j of G, $0 \leq j < p'$, let $C_j = |\{i \mid 0 \leq i < p \text{ and } j \in V_i\}|$. Now $\sum_{j=0}^{p'-1} C_j \geq pT$, so there must be (by the pigeonhole principle) a vertex v with

$$C_v \geq \frac{pT}{p'} \geq \frac{p^2}{p'.((d-1)(k-1)+q)}$$

Choose $k = \frac{p}{\sqrt{p'.(d-1)}} - \frac{q}{d-1} + 1$. If $k<2$ then the result follows: if $q > \frac{p\sqrt{d-1}}{2\sqrt{p'}}$ then a lower bound of $\Omega(\frac{p}{\sqrt{p'}})$ follows immediately (assuming the universal machine has asymptotically the same arity as the machine being simulated), otherwise $\frac{p}{\sqrt{p'}} = O(1)$, and so a lower bound of $\Omega(\frac{p}{\sqrt{p'}})$ is trivial.

Now suppose $k \geq 2$. Then $C_v \geq \frac{p}{\sqrt{p'.(d-1)}}$. If $q > \frac{p.\sqrt{d-1}}{2\sqrt{p'}}$ then a lower bound of $\Omega(\frac{p}{\sqrt{p'}})$ follows immediately. Otherwise $k > \frac{p}{2\sqrt{p'.(d-1)}}$ and so v lies on at least $k > \frac{p}{2\sqrt{p'.(d-1)}}$ routes to vertex i for $C_v \geq \frac{p}{\sqrt{p'.(d-1)}}$ choices of destination i. Thus there is a combination of requests which results in $\frac{p}{2\sqrt{p'.(d-1)}}$ packets being routed through vertex v; furthermore, each packet contains a different data item, which precludes the amalgamation of packets (assuming the universal machine has the same word-size as the machine being simulated). Vertex v thus forms a bottleneck, giving us a time-loss of $\Omega(\frac{p}{\sqrt{p'}})$ for each step of the simulation. This gives us a delay of $\Omega(\frac{P(n)}{\sqrt{P'(n)}})$ for an oblivious simulation. This lower bound very quickly becomes trivial, in fact when $P'(n) = \Omega(P(n)^2)$ it gives us no information at all. In this case, theorem 7.1.2 gives us a lower bound of $\Omega(\log P(n))$. □

The proof of theorem 7.3.1 was motivated by theorem 1 of Borodin and Hopcroft [8], where they prove the same result using essentially the same methods in the special case where $P'(n) = P(n)$ and all dedicated processors are assumed distinct. Lang [39] gives a matching upper bound for the case where $P'(n) = P(n)$ and the data-transfers form a permutation (which, in particular, means that there can be no read or write conflicts in the machine to be simulated). By extending his technique we can derive a general upper bound which asymptotically matches the lower bound of theorem 7.3.1.

Theorem 7.3.2 Suppose $P(n) \le P'(n)$. There is a $P'(n)$ processor universal network based on the shuffle-exchange which can carry out a source-oblivious simulation of a $P(n)$ processor network with delay $O(\frac{P(n)}{\sqrt{P'(n)}} + \log P(n))$.

Proof. Fix n, and let $m = \lceil \log P(n) \rceil$, $m' = \lceil \log P'(n) \rceil - m$. We will describe our algorithm on an $(m+m')$–cube. Processor $i.2^{m'}$ will simulate process i, $0 \le i < P(n)$. The simulation of a single step proceeds as follows. Suppose process i, $0 \le i < P(n)$ wishes to read the communication register of some process j_i, $0 \le j_i < P(n)$. Then each processor $i.2^{m'}$, $0 \le i < P(n)$ makes up a request packet (j_i, i). These packets are routed to the respective processors $j_i.2^{m'}$, with multiple requests being combined as necessary. The requests are fulfilled and routed back to their sources along the same paths. Once read-requests have been dealt with in this manner, write-requests are handled analogously with a single routing.

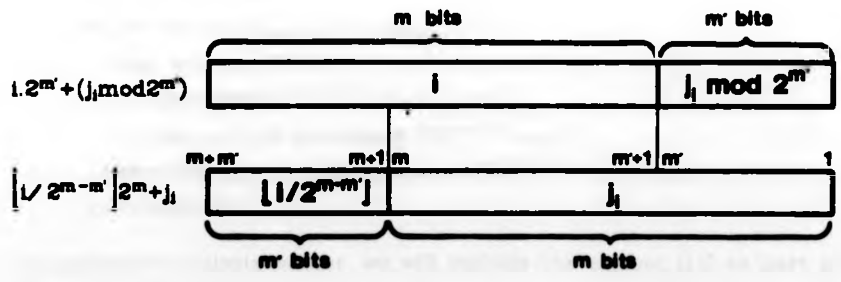
The routing of read-requests is broken up into three parts. In each part we assume that the packets (j_i, i) are held in variables (j, i) of the requisite processor. Processors not in possession of a packet are deemed to hold the null packet (null,null). A *collision* is said to occur if two packets are resident in the same processor.

(a) Route the packets from $i.2^{m'}$ to $i.2^{m'}+(j_i \bmod 2^{m'})$. This can be done quite easily using the following algorithm.

```
for b:=1 to m' do
  if $PID_b = (j$ of processor $PID^{(b)})_b$
    then $(j,i) := (j,i)$ of processor $PID^{(b)}$
    else $(j,i) := (null,null)$
```

There can be no collisions during this stage of the routing, since a packet from processor i remains in processors $i.2^{m'}+x$ for some $0 \le x < 2^{m'}$.

(b) Route the packets from $i.2^{m'}+(j_i \bmod 2^{m'})$ to $\lfloor i/2^{m-m'} \rfloor .2^m+j_i$.



m bits
m' bits
$i.2^{m'}+(j_i \bmod 2^{m'})$
i
$j_i \bmod 2^{m'}$
$m+m'$
$m+1$
m
$m'+1$
m'
1
$\lfloor i/2^{m-m'} \rfloor 2^m+j_i$
$\lfloor i/2^{m-m'} \rfloor$
j_i
m' bits
m bits

This involves changing bits $m'+1$ through m of the current location of each packet, which were previously the low-order bits (bits 1 through $m-m'$) of i, into the high-order bits (bits $m'+1$ through m) of j.

To simplify our presentation, let us assume at first that there are no read-conflicts, so $j_i \ne j_{i'}$ provided $i \ne i'$. We will return to the problem of read-conflicts later. Each processor has a queue, with unit-cost operations enqueue(x,y) (which places packet (x,y) at the tail of the queue), and dequeue (which removes the packet from head of the queue, and returns its value as a result). An attempt to dequeue an empty queue returns the null packet.

This stage of the algorithm consists of $m-m'$ phases. During the k^{th} phase, $1 \le k \le m-m'$, we move each packet (j_i,i) so that bit $m'+k$ of its current location is the same as bit $m'+k$ of $\lfloor i/2^{m-m'} \rfloor .2^m+j_i$. This is sufficient to move packet (j_i,i) from $i.2^{m'}+(j_i \bmod 2^{m'})$ to $\lfloor i/2^{m-m'} \rfloor .2^m+j_i$. Many collisions will occur - this is why each processor has a queue. In order to move every packet in the system in this manner, we must completely flush the queues at each phase. Let m_k be the maximum number of items in each queue at the start of phase k, $1 \le k < m-m'$, to be determined later.

```
initialize queue to empty queue
for k:=1 to m−m' do
  for t:=1 to $m_k$ do
    if (j ≠ null) and ($j_{m'+k}$ = $PID_{m'+k}$)
      then enqueue(j,i)
    if (j of processor $PID^{(m'+k)}$ ≠ null) and
        ($PID_{m'+k}$ = (j of processor $PID^{(m'+k)}$)$_{m'+k}$)
      then enqueue((j,i) of processor $PID^{(m'+k)}$)
    (j,i):=dequeue
```

To make our analysis easier, we will include the packet (j,i) as part of the queue, since we have used variables (j,i) as a dummy head-of-queue in the algorithm. At the beginning of phase 1, the queues are empty, so $m_1 = 1$. After phase k has terminated, a request from process i to process j will find itself in the queue of processor $\lfloor i/2^k \rfloor .2^{m'+k}+(j \bmod 2^{m'+k})$. Each request comes from a different source, and is bound for a different destination.

Thus if two different requests, one from process i to process j, and another from process i' to process j' end up in the same queue at the end of the k^{th} phase, then $i \neq i'$, $j \neq j'$ and $\lfloor i/2^k \rfloor . 2^{m'+k} + (j \bmod 2^{m'+k}) = \lfloor i'/2^k \rfloor . 2^{m'+k} + (j' \bmod 2^{m'+k})$.

How many different choices of i and i' are there? Since $i \neq i'$ and yet $\lfloor i/2^k \rfloor = \lfloor i'/2^k \rfloor$, we are forced to assume that i and i' differ in the last k bits. Thus there are at most 2^k choices for the source, and so $m_k \leq 2^k$. Similarly, since $j \neq j'$ and yet $j \bmod 2^{m'+k} = j' \bmod 2^{m'+k}$ we are forced to conclude that j and j' differ in the leading $m-m'+k$ bits. Thus there are at most $2^{m-m'-k}$ choices for the destination, and so $m_k \leq 2^{m-m'-k}$. Putting both of these together, we conclude that $m_k \leq \min(2^k, 2^{m-m'-k})$.

Thus the algorithm will work if we set $m_k = \min(2^k, 2^{m-m'-k})$. The delay is proportional to

$$\sum_{k=0}^{m-m'-1} m_k = \sum_{k=0}^{m-m'-1} \min(2^k, 2^{m-m'-k}).$$

If $m-m'$ is even the latter is equal to $2^{(m-m'+3)/2}-3$, and if it is odd it is equal to $3.2^{(m-m')/2}-3$. Thus the delay is $O(\sqrt{2^{m-m'}})$.

The case where read-conflicts are allowed is a little more complicated. As well as a queue, arm each processor with a stack, and the usual stack operations. Instead of enqueuing a packet, first check to see whether the queue already contains a packet bound for the same destination. If so, the newly arrived packet is relegated to the stack. This ensures that only packets with different destinations are put in any individual queue, which preserves the invariants necessary for the above timing analysis.

When, much later, the fulfilled request is routed back along the same path, the stack is checked before it is entered on the queue, and any requests for the same data item are fulfilled. By also stacking the time at which a duplicated request was received, the processor can tell when to unstack and despatch each fulfilled request in the return routing.

(c) Thirdly and finally, route the packets from processor $\lfloor i/2^{m-m'} \rfloor .2^m+j_i$ to $2^{m'}.j_i$. This is done in two parts. First route it from $\lfloor i/2^{m-m'} \rfloor .2^m+j_i$ to $j_i.2^{m'}+(j_i \bmod 2^m)$, and then from there to $j_i.2^{m'}$. These two parts correspond to the two for-loops below.

```
for b:=m+m' downto m'+1 do
  if $PID_b = (j$ of processor $PID^{(b)})_b$
    then $(j,i):=(j,i)$ of processor $PID^{(b)}$
for b:=m' downto 1 do
  if $PID_b = 0$
    then $(j,i):=(j,i)$ of processor $PID^{(b)}$
```

Since at all times there are at least m bits of j present in the location of each packet (j,i), and at the end of stage (b) above there are no two distinct packets bound for the same destination, there are no collisions.

Thus by applying the algorithms of parts (a), (b) and (c) consecutively, we can route the request packets in time

$$O(\sqrt{2^{m-m'}}+m'+m) = O(\frac{P(n)}{\sqrt{P'(n)}}+\log P(n))$$

on an (m+m')–cube. Part (a) is a simple-ascend class algorithm, and part (c) is simple-descend. Thus by the use of theorems 4.1.1 and 4.1.2 they can be realized on the cube-connected-cycles or shuffle-exchange interconnection patterns without asymptotic time-loss, using P'(n) processors.

The implementation of (b) needs special care however. It would be simple ascend class except for the fact that m_k data transfers occur along dimensions m'+k, instead of the usual 1. A careful analysis of the proof of theorem 4.1.1 shows that this is easy to handle in the shuffle-exchange case. This does not appear to be the case with theorem 4.1.2 however, due to the pipelining technique used. In the implementation of part (b) on the shuffle-exchange, processes must be moved around at the end of each phase. It takes time proportional to m^k to move the queue at the start of the k^{th} phase, giving asymptotically the same delay as above.

We have not yet described how the fulfilled requests are to be routed back to their sources along the same paths. We simply reverse the above algorithm, by making each ascending loop descend, and vice-versa. In the case where conflicts are allowed, the stacks need not be moved from processor to processor in the simulation of theorem 4.1.1. They are simply implemented as an array at each processor, and elements to be stacked are stored in the processor that the process is currently residing in. Since the algorithm returns the fulfilled requests in a mirror-image of the original routing, each process will be back in the correct place when it wishes to remove a particular item from its stack. $\square$

Chapter 8 Conclusion

We have presented a complexity theory of parallel computation based on a network model, and have argued that this model is a good one, from both a practical and a theoretical point of view. The concept of a universal network is central to our arguments. We have found a practical universal machine which can efficiently simulate the more general model. Thus the user of a practical universal machine is free to program in a high-level language whose virtual architecture corresponds to, and the theoretician is provided with a motive for studying, the more abstract models.

We have seen various kinds of universal machine. A literal simulation is often more efficient than a strongly-literal one, in the sense that slightly less processors are needed (this is tied in strongly with our non-standard definition of space in section 2.1). On the other hand, the number of processors can be reduced even further, and the simulation made strongly-literal, if the machines being simulated are restricted-access networks. Upper bounds on the time required for these simulations can be asymptotically matched by lower-bounds. The situation is quite the opposite, however, in the non-literal case.

We have seen that networks with a large word-size and number of processors are very powerful, even when those processors have a modest instruction-set. In particular, any computable function can be computed in constant time if sufficiently many processors are present. Furthermore, an arbitrary polynomial speed-up of a sequential machine is possible on a network with "reasonable" resources, although an exponential speed-up is probably not.

The choice of a unit-cost measure of time, although controversial in sequential models, can be defended in the parallel case. We have seen a diversely-motivated collection of evidence in favour of the unit-cost hypothesis.

(a) Networks with a unit-cost measure of time are "reasonable" in the sense that they obey the parallel computation thesis, provided a $T(n)$ time-bounded network has instructions which can be simulated by a Turing machine using space $T(n)^{O(1)}$. (Section 3.3).

(b) To ensure that individual processors behave like log-cost sequential machines, replace Turing machine space by deterministic Turing machine time in part (a) above. (Section 3.3).

(c) Networks with a unit-cost measure of time are "reasonable" in the sense that they obey the extended parallel computation thesis, provided an $S(n)$ space-bounded network with word-size $W(n)$ has instructions which can be computed by a deterministic Turing machine using space $(W(n).S(n))^{O(1)}$ and $T(n)^{O(1)}$ reversals. (Section 5.3).

(d) In practice, the average user would probably prefer to own a universal network, rather than go to the expense of fabricating special-purpose networks for each application. In this case, it is valid to use unit-cost charging for a $P(n)$ processor machine whose local instructions take time $O(\log P(n))$ on the universal machine. (Section 7.1).

Thus the unit-cost hypothesis holds for a wide range of instruction-sets (not just the commonly-used arithmetic instruction-sets proposed in section 2.1), including a large class of high-arity machines considered in chapter 6.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley (1974).

2. M. Ajtai, J. Komlós, and E. Szemeredi, "An O(n.log n) sorting network", *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, (Apr. 1983).

3. R. Aleliunas, "Randomized parallel communication", *Proceedings of the ACM Symposium on the Principles of Distributed Computing*, (August 1982).

4. K. E. Batcher, "Sorting networks and their applications", *Proceedings AFIPS Spring Joint Computer Conference* **32** pp. 307-314 (April 1968).

5. P. Beame, "Random routing in constant-degree networks", Technical Report 161/82, Dept. of Computer Science, University of Toronto (1982).

6. N. Blum, "A note on the 'parallel computation thesis'", *Information Processing Letters* **17** pp. 203-205 (1983).

7. A. Borodin, "On relating time and space to size and depth", *SIAM Journal on Computing* **6**(4) pp. 733-744 (Dec. 1977).

8. A. Borodin and J. E. Hopcroft, "Routing, merging and sorting on parallel models of computation", *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, (May 1982).

9. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, "Alternation", *Journal of the ACM* **28**(1)(Jan. 1981).

10. A. K. Chandra, S. J. Fortune, and R. Lipton, "Unbounded fan-in circuits and associative functions", *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, (April 1983).

11. A. K. Chandra, S. Fortune, and R. Lipton, "Lower bounds for constant depth circuits for prefix problems", *Proceedings of the 10th ICALP, Springer-Verlag Lecture Notes in Computer Science* 154(July 1983).

12. S. A. Cook and R. A. Reckhow, "Time-bounded random access machines", *Journal of Computer and System Sciences* 7(4) pp. 354-375 (1973).

13. S. A. Cook, "Deterministic CFL's are accepted simultaneously in polynomial time and log squared space", *Proceedings of the 11th Annual ACM Symposium on Theory of Computing*, (Apr. 1979).

14. S. A. Cook, "Towards a complexity theory of synchronous parallel computation", *L'Enseignement Mathematique* 30(1980).

15. S. A. Cook and C. Dwork, "Bounds on the time for parallel RAMs to compute simple functions", *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pp. 231-233 (May 1982).

16. P. W. Dymond, "Simultaneous resource bounds and parallel computations", Ph. D thesis, issued as Technical Report TR145/80, Dept. of Computer Science, University of Toronto (Aug. 1980).

17. P. W. Dymond and S. A. Cook, "Hardware complexity and parallel computation", *Proceedings of the 21st Annual IEEE Symposium on Foundations of Computer Science*, (Oct. 1980).

18. P. W. Dymond, "Speedup of multi-tape Turing machines by synchronous parallel machines", *Invited address at the special session on theoretical computer science, meeting 792*, American Mathematical society, (Nov. 1981).

19. M. Flynn, "Very high-speed computing systems", *Proceedings of the IEEE* 54 pp. 1901-1909 (Dec. 1966).

20. S. Fortune and J. Wyllie, "Parallelism in random access machines", *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pp. 114-118 (1978).

21. Z. Galil and W. J. Paul, "An efficient general purpose parallel computer", *Journal of the ACM* 30(2) pp. 360-387 (Apr. 1983).

22. M. R. Garey and D. S. Johnson, *Computers and intractability: a guide to the theory of NP-completeness*, W. H. Freeman (1979).

23 L. M. Goldschlager, "Synchronous parallel computation", Ph. D. Thesis, issued as TR-114, Dept. of Computer Science, University of Toronto (December 1977).

24. L. M. Goldschlager, "The monotone and planar circuit value problems are log space complete for P", *SIGACT News* 9(2)(1977).

25. L. M. Goldschlager, "Epsilon-productions in context-free grammars", Technical Report TR13, Dept. of Computer Science, University of Queensland (Apr. 1980).

26. L. M. Goldschlager, R. A. Shaw, and J. Staples, "The maximum flow problem is log space complete for P", Technical Report TR28, Dept. of Computer Science, University of Queensland (June 1981).

27. L. M. Goldschlager, "A universal interconnection pattern for parallel computers", *Journal of the ACM* 29(4) pp. 1073-1086 (Oct. 1982).

28. L. M. Goldschlager and I. Parberry, "On the construction of parallel computers from various bases of boolean functions", Theory of Computation Report No. 48, Department of Computer Science, University of Warwick (March 1983).

29. L. M. Goldschlager and A. M. Lister, *Computer science: a modern introduction*, Prentice-Hall (1983).

30. J. Hartmanis and J. Simon, "On the power of multiplication in random access machines", *Proceedings of the 15th Annual IEEE Symposium on Switching and Automata Theory*, pp. 13-23 (1974).

31. F. Meyer auf der Heide, "Efficiency of universal parallel computers", *Acta Informatica* **19** pp. 269-296 (1983).

32. F. Meyer auf der Heide, "Infinite cube-connected cycles", *Information Processing Letters* **16** pp. 1-2 (Jan. 1983).

33. F. Meyer auf der Heide, "Efficient simulations among several models of parallel computers", Interner Bericht 2/83, Fachbereich Informatik, Universitat Frankfurt (1983).

34. N. D. Jones, Y. E. Lien, and W. T. Laaser, "New problems complete for nondeterministic log space", Technical Report TR-75-1, Dept. of Computer Science, Kansas University (Apr. 1975).

35. N. D. Jones and W. T. Laaser, "Complete problems for deterministic polynomial time", *Theoretical Computer Science* **3** pp. 105-117 (1977).

36. R. M. Karp and R. J. Lipton, "Turing machines that take advice", *Symposium uber logik und algorithmik" in honour of Ernst Specker, L'Enseignment Mathematique* **30**(Feb. 1980).

37. R. E. Ladner, "The circuit value problem is log space complete for P", *SIGACT News* **7**(1) pp. 18-20 (1975).

38. R. E. Ladner and M. J. Fischer, "Parallel prefix computation", *Journal of the ACM* **27**(4) pp. 831-838 (October 1980).

39. T. Lang, "Interconnections between processors and memory modules using the shuffle-exchange network", *IEEE Transactions on Computers* C-**25**(5)(May 1976).

40. F. T. Leighton, "Problem P44", *Bulletin of the European Association for Theoretical Computer Science*, (22) p. 110 (February 1984).

41. T. Leighton, "Tight bounds on the complexity of parallel sorting", *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, (April-May 1984).

42. G. F. Lev, N. Pippenger, and L. G. Valiant, "A fast parallel algorithm for routing in permutation networks", *IEEE Transactions on Computers* **C-30**(2)(Feb. 1981).

43. L. G. L. T. Meertens, "Recurrent ultracomputers are not log n - fast", Technical Report IW118/79, Dept. of Computer Science, Mathematisch Centrum (Sept. 1979).

44. G. Miranker, L. Tang, and C. K. Wong, "A zero-time VLSI sorter", *IBM Journal of Research and Development* **27**(2) pp. 140-148 (Mar. 1983).

45. D. Nassimi and S. Sahni, "Data broadcasting in SIMD computers", *IEEE Transactions on Computers* **C-30**(2) pp. 101-106 (Feb. 1981).

46. D. Nassimi and S. Sahni, "Parallel algorithms to set up the Benes permutation network", *IEEE Transactions on Computers* **C-31**(2)(Feb. 1982).

47. D. Nassimi and S. Sahni, "Parallel permutation and sorting algorithms and a new generalized connection network", *Journal of the ACM* **29**(3) pp. 642-667 (July 1982).

48. D. C. Opferman and N. T. Tsao-Wu, "On a class of rearrangable switching networks", *Bell Systems Technical Journal* **50** pp. 1579-1618 (1971).

49. J. Orenstein, T. H. Merrett, and L. Devroye, "Linear sorting with O(log n) processors", *BIT* **23** pp. 170-180 (1983).

50. I. Parberry, "Some practical simulations of impractical parallel computers", Theory of Computation Report No. 58, Dept. of Computer Science, University of Warwick (December 1983).

51. I. Parberry, "Some processor-saving theorems for synchronous parallel computers", Theory of Computation Report No. 53, Dept. of Computer Science, University of Warwick (October 1983).

52. I. Parberry, "On the power of parallel machines with high-arity instruction sets", Theory of Computation Report No. 57, Dept. of Computer Science, University of Warwick (November 1983, Updated February 1984).

53. N. Pippenger, "On simultaneous resource bounds", *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, (Oct. 1979).

54. V. Pratt and L. J. Stockmeyer, "A characterization of the power of vector machines", *Journal of Computer and System Sciences* **12** pp. 198-221 (1976).

55. F. P. Preparata and J. Vuillemin, "The cube-connected cycles: a versatile network for parallel computation", *Communications of the ACM* **24**(5) pp. 300-309 (May 1981).

56. M. J. Quinn and N. Deo, "Parallel algorithms and data structures in graph theory", Technical Report CS-82-098, Computer Science Department, Washington State University (Oct. 1982, Revised June 1983).

57. J. Reif and L. Valiant, "A logarithmic time sort for linear size networks", *Proceedings of the 15th Annual ACM Symposium on Theory of Computing*, pp. 10-16 (Apr. 1983).

58. R. Reischuk, "A lower time-bound for parallel random-access machines without simultaneous writes", Research Report RJ3431, IBM Research, San Jose (Mar. 1982).

59. W. L. Ruzzo, "On uniform circuit complexity", *Journal of Computer and System Sciences* **22**(3) pp. 365-383 (June 1981).

60. W. J. Savitch, "Parallel random access machines with powerful instruction sets", *Mathematical Systems Theory* **15** pp. 191-210 (1982).

61. A. Schorr, "Physical parallel devices are not much faster than sequential ones", *Information Processing Letters* **17** pp. 103-106 (August 1983).

62. J. T. Schwartz, "Ultracomputers", *ACM Transactions on Programming Languages and Systems* **2**(4) pp. 484-521 (Oct. 1980).

63. J. C. Sheperdson and H. E. Sturgis, "Computability of recursive functions", *Journal of the ACM* **10**(2) pp. 217-255 (1963).

64. Y. Shiloach and U. Vishkin, "Finding the maximum, sorting and merging in a parallel computation model", *Journal of Algorithms* **2** pp. 88-102 (1981).

65. H. Simon, "A tight Ω(log log n)-bound on the time for parallel RAM's to compute nondegenerated Boolean functions", *Information and Control* **55** pp. 102-107 (1982).

66. H. S. Stone, "Parallel processing with the perfect shuffle", *IEEE Transactions on Computers* **C-20**(2) pp. 153-161 (Feb. 1971).

67. E. Upfal, "Efficient schemes for parallel communication", *Proceedings of the ACM Symposium on the Principles of Distributed Computing*, (1982).

68. E. Upfal, "A probabilistic relation between desireable and feasible models for parallel computation", *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, (April-May 1984).

69. L. G. Valiant and G. J. Brebner, "Universal schemes for parallel communication", *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, pp. 263-277 (1981).

70. L. G. Valiant, "A scheme for fast parallel communication", *SIAM Journal on Computing* 11 pp. 350-361 (1982).

71. U. Vishkin, "A parallel-design space distributed implementation space (PDDI) general purpose computer", Research Report RC9541, IBM Thomas Watson Research Centre, Yorkown Heights (1982).

72. U. Vishkin, "Implementation of simultaneous memory address accesses in models that forbid it", *Journal of Algorithms* 4(1) pp. 45-50 (Mar. 1983).

73. U. Vishkin, "Synchronous parallel computation - a survey", Technical Report #71, Dept. of Computer Science, Courant Institute, New York University (April 1983).

74. U. Vishkin and A. Wigderson, "Trade-offs between depth and width in parallel computation", *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, (November 1983).

75. A. Waksman, "A permutation network", *Journal of the ACM* 15(1) pp. 159-163 (Jan. 1968).

76. Y. Wallach, "Alternating sequential/parallel processing", *Springer-Verlag Lecture Notes in Computer Science* 127(1982).