THE COMPLEXITY OF PARALLEL COMPUTATIONS

by

James C. Wyllie

Ph.D. Thesis

TR 79-387

THE COMPLEXITY OF PARALLEL COMPUTATIONS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment for the Degree of

Doctor of Philosophy

by

James Christopher Wyllie

August 1979

THE COMPLEXITY OF PARALLEL COMPUTATIONS

James Christopher Wyllie, Ph.D.
Cornell University 1979

Recent advances in microelectronics have brought closer to feasibility the construction of computers containing thousands (or more) of processing elements. This thesis addresses the question of effective utilization of such processing power. We study the computational complexity of synchronous parallel computations using a model of computation based on random access machines operating in parallel and sharing a common memory, the P-RAM. Two main areas within the field of parallel computational complexity are investigated. First, we explore the power of the P-RAM model viewed as an abstract computing device. Later, we study techniques for developing efficient algorithms for parallel computers.

We are able to give concise characterizations of the power of deterministic and nondeterministic P-RAMs in terms of the more widely known space and time complexity classes for multi-tape Turing machines. Roughly speaking, time-bounded deterministic P-RAMs are equivalent in power to (can accept the same sets as) space-bounded Turing machines, where the time and space bounds differ by at most a polynomial. In the context of comparing models of computation, we consider such polynomial differences in resources to be insignificant. Ad-

ding the feature of nondeterminism to the time-bounded P-RAM changes its power to that of a nondeterministic Turing machine with an exponentially higher running time.

The later sections of the thesis examine algorithm design techniques for parallel computers. We first develop efficient procedures for some common operations on linked lists and arrays. Given this background, we introduce three techniques that permit the design of parallel algorithms that are efficient in terms of both their time and processor requirements. We illustrate the use of these techniques by presenting time and processor efficient parallel algorithms for three problems, in each case improving upon the best previously known parallel resource bounds. We show how to compute minimum string edit distances, using the technique of pairwise function composition. We describe an algorithm for the off-line MIN that organizes its computation in the form of a complete binary tree. Finally, we present an algorithm for undirected graph connectivity that relies on redundancy in its representation of the input graph.

To my parents

Acknowledgments

I would like to thank my thesis advisor, John Hopcroft, for his encouragement of the research that led to this thesis. Without his support, there can be no doubt that this thesis never would have been written.

The Computer Science Department of Cornell University provides a fertile research environment, and I would like to thank its members, both students and faculty, for their valuable ideas and insights that have immensely broadened my education. Particular thanks go to Steve Fortune, who had the patience to read an early draft of this thesis and suggest many improvements.

Finally, I must thank Dianne Deters (now Dianne Wyllie) for her unfailing encouragement, support, patience, and love throughout my years at Cornell.

Table of Contents

Chapter 1

Introduction

The speed of serial computers has increased enormously over the past decade. Unfortunately, due to ultimate physical limitations, this increase cannot continue indefinitely. However, by performing some or all of a desired computation in parallel, greater execution speed can be obtained. One need only look at the proceedings of the most recent computer architecture conference [S4] to detect the degree of interest in parallelism among computer designers; over half the papers are concerned with some aspect of parallelism.

This thesis approaches the study of parallelism from a different direction, namely computational complexity. As used here, the term computational complexity will mean the quantitative study of the computational resources required for the solution of some problem or class of problems. The two computational resources that figure most prominently in classical complexity theory are time and space. In extending the classical theory to accomodate parallelism, other resources must be considered, including the degree of parallelism, the degree of communication among the processors cooperating in the solution of the problem, and so on. The advantage of the computational complexity viewpoint is that its results will not be made obsolete by the next advance in computer hardware; the asymptotic analyses here are approximately valid over a wide range of computer performance.

The goal, then, of our study of parallel computational complexity is to provide parallel analogs for some of the major results of sequential computational complexity and to organize these results in a manner that will permit them to be applied towards the design and programming of parallel computers. Specific areas of study might include techniques for deriving upper and lower bounds for particular problems, complete problems for various resource classes, and the fundamental relationships between the resource requirements of sequential versus parallel implementations of algorithms for problems of interest. The remaining chapters of the thesis will address some of these questions, providing partial answers to some of them.

In order to study the complexity of parallel computations, a specific model must be agreed upon. All of the results to follow will be stated relative to this model, in order to facilitate comparisons. There are a number of axes along which parallel models can be classified. The discussion of most of these is deferred to Chapter 2, but the choices between synchronous versus asynchronous and bounded versus unbounded parallel models are crucial enough that they will be justified here in the introductory section.

By a synchronous model we mean one in which there is a known, well-defined relationship among the processing speeds and instruction timings of the processors executing the pieces of the parallel computation. An example of a synchronous parallel computer would be an array of microcomputers driven by a common clock signal. An asynchronous model is one in which there is no such relationship among the execution speeds, where in fact processors can proceed at differing and time-varying rates. The notion of the "running time" of an algorithm on an asynchronous model is not well-defined, because of this inherent variability. Asynchronous models form the basis for the formal study of topics in operating systems: mutual exclusion, deadlock avoidance, determinacy, etc., where the goal is to prove theorems about the behavior of a system of cooperating sequential processes regardless of their relative execution speeds. For a description of the most important formal asynchronous parallel models, as well as extensive bibliographies, see [M1] or [P1]. Since our interests here lie in quantitative rather than qualitative aspects of parallelism, our model of parallel computation will be synchronous instead of asynchronous.

The choice between bounded and unbounded parallelism is somewhat more difficult. Certainly any parallel computer that will ever be built can have only a fixed number of processing elements. However, just as the random access machine model for sequential computers assumes registers of unbounded length, so will we assume unbounded parallelism. The justification for unboundedly long registers in RAMs is that over a reasonable range of input sizes, the RAM approximates a real computer. Similarly, one may expect that over a reasonable range of input sizes the parallel model will fairly closely approximate a parallel computer containing a finite number, say 10^5 or 10^6, processors. Also, it should be noted that a parallel model with parallelism limited to a constant k is indistinguishable asymptotically from a sequential model that is k times faster than each of the parallel processors.

The remainder of the thesis studies the computational complexity of synchronous, unboundedly parallel computers. It is organized as follows. Chapter 2 describes the particular parallel model, called the P-RAM, that will be used throughout the thesis. Justifications for the attributes chosen for the P-RAM are presented, as well as arguments that demonstrate its robustness, or insensitivity to minor changes, with respect to the resource requirements of programs that run on the model. A parallel programming language is also presented; it allows more compact descriptions of the parallel algorithms to follow. Chapter 3 gives a concise characterization of the power of time-bounded parallel computers in terms of the more familiar Turing machine time and space complexity classes. Also discussed are restrictions to the basic P-RAM model, and the effect of these restrictions on the power of the device, once again expressed relative to the standard complexity classes. Chapter 4 describes some of the basic techniques of P-RAM programming useful in the construction of efficient parallel algorithms. The complexities of these techniques are analyzed with respect to both time and the degree of parallelism required. Chapter 5 uses the techniques of Chapter 4 to give efficient parallel algorithms for several combinatorial problems, in several cases improving on the best previously known parallel resource bounds. Finally, the last chapter summarizes the important concepts introduced by the thesis and poses several unanswered questions.

Chapter 2
Models of Parallel Computation

In this chapter we introduce the parallel machine model that will be employed throughout the remainder of the thesis. Section 2.1 surveys existing parallel computers. The basic properties of our parallel model are described in Section 2.2, then in Section 2.3 we show that a variety of changes to the model, some of them radical, do not substantially affect the resource requirements of programs that run on the model. In the last section of the chapter, we introduce the parallel programming language that is used to present the computational structures and parallel algorithms of Chapters 4 and 5. This final section also shows how to implement the statements of the programming language on the parallel model and how much time and other resources should be charged to the various constructs of the language.

2.1 Existing parallel computers

Current computers exhibiting some of the features of parallelism fall into two main categories: the so-called "pipelined" vector machines, exemplified by the Cray-1 [R1], and true multiprocessors, for example the ILLIAC IV [B1] and Cm* [S6]. We shall outline the characteristics of each of these machines before presenting the parallel model to be used throughout the thesis.

The Cray-1 is the fastest computer currently available. It is a parallel machine in only a certain limited sense, however. The architecture of the Cray-1 is intended to optimize the essentially serial processing of vectors of data, where the identical operation is to be performed on corresponding elements of a pair of vectors. Each of the execution units that carry out the arithmetic instructions of the machine is pipelined; that is, additional sets of operands may be fed into the execution unit before the result of the first set of operands has become available. In this way, parallelism at the lowest level of hardware occurs, since several partially completed operations on elements of the vectors reside in the execution unit at the same time. Once an initial startup period has elapsed, answers can be extracted from the execution unit faster than the unit is capable of producing an answer to any individual operation. The length of the pipeline is bounded by a small constant. Thus, from a theoretical point of view, the Cray-1 is merely a sequential computer sped up by this small constant; hence it is unsuitable as a model for a more general study of the power of parallelism.

The ILLIAC IV comes closer to the parallel model desired here. There are 64 processing elements arranged in the form of a rectangular grid, where each processing element may communicate with its four neighbors on the grid. Each processing element is a computer in its own right, having a CPU and some memory. Instructions are broadcast from a control processor, then executed simultaneously by all processing elements, each using its local memory as the source of its operands. The problem with ILLIAC is communication; for a processing element to obtain data from the memory of some other processing element, the data must be explicitly routed from one processing element to the other. Imagining an ILLIAC style computer having n processing elements, this data transfer could require time $O(n^{1/2})$ in the worst case. Indeed, in [G1] lower bounds of this order for this model are shown for several simple problems, including matrix multiplication. As will be seen, exponentially better running times can be obtained using the P-RAM model to described later.

The Cm* machine provides the next step towards a general purpose parallel computer. In this machine, there are a (potentially large) number of processor - memory pairs, where each pair is a commercially available microcomputer. The pairs are grouped into clusters, and the clusters into superclusters, with interprocessor communication taking place over buses connecting the members of each level of the hierarchy. Logically, each processor views the entire memory of the machine homogeneously, thus may refer to any memory location in any cluster. Physically, references to remote memory locations are implemented by a series of messages passed over the buses. To take advantage of the (hoped for) locality of reference within processors, the machine is asynchronous. Therefore, it is not slowed down to the speed of the slowest memory reference. As a model of parallel computation for the purposes of this thesis, Cm* suffers due to its asynchronous operation. As mentioned in the introduction, we shall be interested in deriving worst case asymptotic running times for parallel algorithms, thus asynchronous models are not appropriate.

2.2 The parallel random access machine model

The model used in this thesis to compare the power of parallel computations to sequential Turing machine computations and to derive efficient parallel algorithms is the parallel random access machine (P-RAM). The P-RAM model is intended to capture the ideas of synchronous computation from ILLIAC IV, while at the same time allowing the elegant logical memory structure of Cm*. Called by other names or not defined explicitly at all, the P-RAM model has been used by other authors for a variety of problems [A3], [C1], [C3], [C5], [H2], [H4], [H5], [K2], [M2], [P3].

A P-RAM consists of an unbounded set of processors P_0, P_1, ..., an unbounded global memory, a set of input registers, and a finite program. Each processor has an accumulator, an unbounded local memory, a program counter, and a flag indicating whether or not the processor is running. All memory locations are capable of holding arbitrary non-negative integers. The program consists of a sequence of possibly labelled instructions chosen from the list in Figure 2.2.1 below. A program is nondeterministic if some label occurs more than

Instruction		Function
LOAD	operand	Transfer operand to/from the accumulator from/to memory.
STORE	operand	
ADD	operand	Add/proper subtract the value of the operand to/from the accumulator.
SUB	operand	
JUMP	label	Unconditional branch to label.
JZERO	label	Branch to label if accumulator is zero.
READ	operand	See text.
FORK	label	See text.
HALT		See text.

Figure 2.2.1 The P-RAM instruction set.

once, deterministic otherwise. Each operand may be a literal, an address, or an indirect address. Each processor may access either global memory or its local memory, but not the local memory of any other processor. Indirect addressing may be through one of these memories to access the other.

Initially the input to the P-RAM is placed in the input registers, one bit per register; all memory is cleared; the length of the input is placed in the accumulator of P_0; and P_0 is started at the first instruction of the program. At each step in the computation, each running processor executes the instruction given by its program counter in one unit of time, then advances its program counter by one unless the instruction causes a jump.

A READ instruction uses the value of the operand to specify one of the input registers; the contents of the selected register is placed in the accumulator. A FORK label instruction executed by processor P_i selects an inactive processor P_j, clears P_j's local memory, copies P_i's accumulator into P_j's accumulator, and starts P_j running at label. A HALT instruction causes a processor to stop running.

Simultaneous reads of a location in global memory are allowed, but if two processors try to write into the same memory location simultaneously, the P-RAM immediately halts and rejects its input. Several processors may read a location while one processor writes into it; all reads are completed before the value of the location is changed.

Execution continues until P_0 executes a HALT instruction (or two processors attempt to write into the same location simultaneously). If the P-RAM is acting as an acceptor, then the input is accepted only if there is some computation in which P_0 halts with a one in its accumulator; the time required to accept the input is the minimum over all such computations of the number of instructions executed by P_0. If the P-RAM is acting as a transducer, the output will be placed in designated global memory locations and the running time is measured as above. The number of processors required to accept or tranform an input is the maximum number of processors that were active (started by a FORK and not yet stopped by a HALT instruction) at any instant of time during the P-RAM's computation.

Several details of the model deserve further comment. A great deal has appeared in the literature about processor interconnection patterns that improve on the ILLIAC IV grid [S5] [W3], but a totally satisfactory pattern has not yet been found, and indeed may not exist. Thus, we avoid a potential problem by positing a global memory. In Chapter 3, some of these interconnection schemes will be discussed and models using explicit interprocessor communication rather than a global memory will be shown to possess many of the same characteristics as P-RAMs.

Given a global memory, we could restrict the local memory of each processor to only a constant number of locations by mapping the local memory of each processor into global memory. However, in Chapter 3 we shall be interested in what happens when the size of global memory is restricted, hence we allow each processor unbounded local memory. Also, we choose to use special registers to hold the input in order to be able to discuss sublinear running times. If, say, the input were encoded in binary and placed in the accumulator of P_0, then it would take at least linear time (in the length of the input) to unpack the input.

In Chapters 4 and 5, however, we shall be dealing with objects such as graphs and matrices where a more compact input representation is reasonable. Therefore, in those chapters, we shall permit the input registers to hold integers as large as the size of the input object and express resource bounds as a function of the number of input registers rather than as a function of the sum of their lengths. For example, the input representation for a graph of size n would be a sequence of integers, each no longer than log n bits. Since the single bit per register input form of such an object can certainly be packed into longer registers in time proportional to the length of the longer registers, an additive term of log n should be charged to algorithms manipulating such objects. But since all algorithms to be presented require at least log n time, the time to pack the input is insignificant. Unfortunately, the processor cost to perform this packing is non-trivial; our algorithm to compute the sum of n numbers in time log n using $\frac{n}{\log n}$ processors does not have the resources to examine $n \cdot \log n$ separate bits. In that sense then, the P-RAM model as used to develop efficient algorithms differs from that used to compare the power of parallel computers and Turing machines.

2.3 Variations on the P-RAM model

There are several simple modifications to the model described in the previous section that do not significantly affect either resource requirements or programming. For example, the input registers could be eliminated by storing the input in designated global memory locations. The instruction set of the P-RAM may be changed somewhat without major disturbances; some of the addressing modes are redundant, for example. Also, features found in other parallel models can be simulated easily on the P-RAM.

Two examples of such simulations will be given: processor numbers and channel registers. To implement these or other changes in an arbitrary P-RAM program the idea of interleaving will be employed. When we say "operation x is interleaved with instruction execution," we mean that each instruction i is to be replaced by a block of instructions having constant execution time that performs operation x followed by instruction i. Since the same block has been prefixed to each instruction, the modified program will possess the same timing characteristics as the unmodified version, slowed down by a constant factor. In particular, the presence or absence of simultaneous writes into a global memory location is unaffected by this program transformation. When only some instructions are to be preceded by some new code, all other instructions must be prefixed by the appropriate number of null instructions in order to preserve this property of synchrony.

The parallel models of [G3] and others include a special instruction that allows a processor to obtain its processor number. These processor numbers are used to indicate to a processor which element of the input it is to process, to arbitrate memory references, etc. We show how to simulate this facility on a P-RAM as follows. Each processor will have a register in its local memory, pn, devoted to holding the processor number. Interleaved with the execution of all instructions, the processor will double the value in its pn register. FORK instructions in the original program are augmented by code which passes to the child processor the value $2 \cdot pn + 1$ to be used as an initial processor number. These two operations insure that at every instant of time, all pn registers will contain different values. The argument passing can be done at

FORKs by passing through the accumulator the address of a block in global memory containing the necessary parameters.

The following scheme suffices to manage space in global memory. In general, assume that a processor has free storage consisting of every k^{th} location starting at address m. Upon executing a FORK, the processor can assign to its child every $2k^{th}$ location beginning at address m+k and change its own allocation to be every $2k^{th}$ location beginning at address m, insuring disjoint global memory work areas for all processors. All of the above changes can be composed together, yielding a program that executes more slowly by only a constant factor.

In [S2] a parallel model similar to the P-RAM is presented in which, during the execution of the equivalent of a k-way FORK instruction, a fixed number of parameters may be passed to the child processors using channel registers. The parent may then wait until its children have completed processing and read their results out of the same channel registers. P-RAM simulation of channel registers is easy; argument passing through a block of storage in global memory as described above works nicely. To simulate the wait of the parent, a busy wait may be performed, repeatedly testing for completion of the (constant number of) children.

Not all modifications to the P-RAM model can be handled quite so easily. Another parallel machine model is the SIMDAG [G3], which is similar to the P-RAM model in that it is based on the RAM model for sequential computations [C4] and has a global memory accessible by all processors. However, the

SIMDAG differs from the P-RAM in two major respects. First, the two models use different conventions for resolving write conflicts in global memory. The P-RAM disallows such conflicts entirely, while the SIMDAG allows the lowest numbered processor to update the contents of a global register in the event of a write conflict. This discrepency allows a SIMDAG to determine in constant time whether a Boolean vector contains any ones, by assigning a processor to each element of the vector and having the processor assigned to a particular element store a one in some common global memory location if its vector element is non-zero. A P-RAM algorithm for the same problem requires that answers be fanned together pairwise in the form of a complete binary tree to prevent an input Boolean vector that has more than a single one from causing a write conflict in global memory. This fan-in procedure takes time proportional to the logarithm of the length of the vector.

To simulate an arbitrary SIMDAG program on a P-RAM, each write into global memory by the SIMDAG must be simulated by a fan-in tree of depth the log of the number of currently active processors, which can be as large as $\log 2^t = t$ at time t. Thus, in the worst case P-RAM time will be the square of SIMDAG time. However, in [G3] the notion of a "charged SIMDAG" is presented, in which the instructions of the SIMDAG are charged at an amount reflecting the cost of their implementation at a more primitive level, rather than at simply unit time per operation. It turns out that the charged SIMDAG cost in the worst case will also be the square of the simple SIMDAG running time, so the P-RAM prohibition of global memory write conflicts appears reasonable in this light.

The second major difference between the P-RAM and SIMDAG models is that the "S" in SIMDAG stands for "single instruction stream," which means that as in the ILLIAC IV there is a central unit that broadcasts instructions to all of the processors. In contrast, each processor of a P-RAM has its own program counter and may thus execute a portion of the program different from that being executed by other processors. In the notation of [F1], the P-RAM is therefore a "MIMD," or "multiple instruction stream, multiple data stream" computer. It might appear that allowing multiple instruction streams adds power to the model, but the discussion below shows that this is not the case.

There are two issues to be dealt with. First, the differing strategies for resolving write conflicts in global memory must be reconciled. Later, we must show how to simulate multiple instruction streams using a single instruction stream. It is necessary for the SIMDAG to be able to detect when more than one processor attempts a write into the same global memory location. Since the processors in the SIMDAG can determine their processor numbers, by using the subroutine in Figure 2.3.1 to perform each write, global memory conflicts can be detected. The conflict signalling can be done by writing into a specified register in global memory.

```
procedure store (x, a):
/* store value x into global memory location a */
  pn := processor number;
  store pn into a;
  if value of a is not pn
    then signal that a write conflict has occurred
    else store x into a
  fi
end
```

Figure 2.3.1 SIMDAG write conflict detection.

It is not yet clear that the SIMDAG is capable of executing the write conflict detection program, since as presented so far there is no way for some processors to execute the then clause while others execute the else clause of the if statement. In existing SIMD machines [B1] [T3], conditionals are accomplished by having comparison instructions set one or more mask bits within each processor, then broadcasting instructions whose execution by a given processor is conditioned by the setting of its mask bits. Under such a scheme, the SIMD code broadcast to all processors to implement the conditional

```
begin
  if A
    then B
    else C
  fi
end
```

Figure 2.3.2 Conditional statement.

statement in Figure 2.3.2 might be that shown in Figure 2.3.3.

```
begin
  evaluate A, setting mask bit if true;
  execute B - occurs only within those processors whose
       mask bit is set;
  reverse mask bit - this is always executed by all pro-
       cessors, regardless of their mask bit settings;
  execute C;
  set all mask bits
end
```

Figure 2.3.3 SIMD code to implement conditional statement.

Nested conditionals can be handled with the aid of instructions to push and pop the mask bits.

The SIMDAG contains only a much simpler mechanism for dealing with conditionals. The only parallel instruction whose outcome is conditioned on the value of some register is one which stores a value into global memory only if the contents of some specified local register is greater than zero. The register specified can be treated as a mask, with execution masked off if the contents of the register is less than or equal to zero. Each instruction in, say, the <u>then</u> clause of an <u>if</u> statement can be processed as follows. Since no instructions have side effects, the effect of any instruction will be to compute some value v and store it into an address a. If a is in global memory the conditional store instruction can be used directly. To simulate the effect of masking if a is in a processor's local memory, compute the value v and then execute the program in Figure 2.3.4, where there is a temp register in global memory for each processor and where the parenthesized statement is executed only if the mask register

```
begin
  temp  := contents of a;
  (temp := v);
  copy temp into address a;
end
```

Figure 2.3.4 SIMDAG simulation of mask registers.

is greater than zero. Nested conditionals can be simulated by a statically allocated stack of mask registers.

Given this method of performing nested conditional statements on a SIMDAG, the single instruction stream simulation of multiple instruction streams is straightforward. The accumulator and program counter registers of each processor of the P-RAM are simulated by registers in global memory of the SIMDAG. An initial segment of the SIMDAG program will write a copy of the P-RAM program into global memory in some easily decoded form. The remainder of the SIMDAG program will be an implementation of the fetch-execute cycle of the P-RAM, a portion of which is shown in Figure 2.3.5.

The only instruction whose simulation is at all tricky is FORK, and that too can be handled with little difficulty. By maintaining a pn register as described earlier, each processor p at each instant of time will have a "buddy" processor, which will be awakened if p needs to simulate a FORK at that particular instant. Since the simulated accumulators and program counters are kept in the global memory of the SIMDAG, the newly activated processor can be initialized with the proper values. The running time of the SIMDAG simulation is just a

```
begin
  while true do
    for all processors do
      I := instruction of the program pointed to by the
           program counter;
      if I = ADD with operand in local memory
        then simulate the ADD instruction
        else if I = ADD with operand in global memory
               then simulate the instruction
               else
                 .
                 .
                 .
             fi
      fi
    od
  od
end
```

Figure 2.3.5 SIMDAG simulation of P-RAM.

constant (albeit a very large one) times the running time of the P-RAM for the same program. The preceding discussion has proven Theorem 2.3.1, which is stated below.

<u>Theorem</u> 2.3.1 Any P-RAM program may be simulated on a SIMDAG with only a constant factor slowdown.

The importance of Theorem 2.3.1 is not so much that a SIMDAG can simulate a P-RAM, but that any reasonable single instruction stream model capable of executing conditional statements can simulate a multiple instruction stream model with only a constant factor loss in speed. This result is of interest only to theoreticians, not computer architects, since the constant in question will be on the order of the size of the instruction set of the multiple instruction stream machine, which can be in the hundreds for modern computers.

2.4 Parallel pidgin Algol

The proofs of the theorems in Chapter 3 will be expressed in terms of the low-level machine instructions of the P-RAM. Such programs at the machine language level are not appropriate as a vehicle for describing the computational structures and parallel algorithms of Chapters 4 and 5. The purpose of this section is to describe a high level programming language, parallel pidgin Algol, which will be used in later chapters to express all of the parallel algorithms introduced. A formal description of the language would be a major undertaking and will not be attempted; instead we hope only that our informal language specification will be sufficient to allow the reader to understand the basic concepts behind the algorithms that follow. Since we shall often be interested in the running times of parallel pidgin Algol programs, we shall describe how one might implement each statement of the language in terms of P-RAM instructions.

A parallel pidgin Algol program is of the form <u>begin</u> statement list <u>end</u>, where a statement list is a list of statements chosen from the list below and separated by semicolons. Clauses enclosed in brackets [ ] are optional. Variables will not normally be declared, their definitions either being clear from context or described by text preceding the program. Allocation of storage in global memory has been discussed previously, and allocation within a processor's local memory can usually be accomplished by some simple static mapping policy.

1. variable := expression
2. _if_ condition _then_ statement list [_else_ statement list] _fi_
3. _while_ condition _do_ statement list _od_
4. _for_ variable := initial value _to_ final value [_by_ stepsize] _do_ statement list _od_
5. _procedure_ name (formal parameter list): statement list _end_
6. procedure name (actual parameter list)
7. _assign_ processor specification
8. _for_ processor specification _do_ statement list _od_
9. any other well-defined statement

Statement types 1 through 6 are essentially the same (with minor syntax changes) as the corresponding definitions of sequential pidgin Algol statements given in [A1]. The costs assigned to these statements will be discussed later.

Statement type 7 is the means by which parallel pidgin Algol programs access the FORK instruction of the P-RAM. An example of an _assign_ statement might be

assign a processor to each element of array A.

The P-RAM implementation of the above _assign_ statement would be a FORK instruction in a loop, with the loop controlled in such a way that the processor encountering the _assign_ would execute a FORK, then the two active processors would also execute FORKs, etc., until a total of |A| processors were active.

We assume that processor numbers are passed at each FORK, so that processors can know to which element of A they have been assigned. The time to execute the assign is the number of executions of the FORK loop, or the depth of the tree of processor activations, in this case $O(\lceil \log |A| \rceil)$. If $|A|$ is not a power of two, the FORK tree is complicated by the fact that some processors stop FORKing one iteration before others. We shall typically ignore such details, since the program changes required to implement them are relatively straightforward, although messy.

Statement type 8 is the companion to type 7. All of the specified processors execute the statement within the scope of the do od brackets. There are no notions of complicated run-time scheduling involved here; enough processors will be assumed so that each node of a graph, or element of an array, etc., may have its own processor. An example of this statement type is:

for each processor p do A(p) := A(p) + 1 od

Assuming that this statement followed the assign statement above, it would have the effect of incrementing each element of the array A by one. Note the use of the dummy variable p; the expression A(p) refers to the element of the array A assigned to processor p. Depending on whether or not elements of the array A need to be referenced by more than one processor, the parallel pidgin Algol compiler would allocate storage for A in either global or local memory.

Square brackets are used for standard array references, so that B[A(p)] would refer to "the element of array B (which is implicitly global) given by the element of array A assigned to processor p." Before discussing the timing characteristics of "for processors do S od" statements, it is necessary to examine the implementation of some simpler statements.

Consider the execution of the statement in Figure 2.4.1 by two processors simultaneously, where x and y are global

```
for processors p do
  if B(p)
    then x := f(y);
    else y := g(x);
  fi
od
```

Figure 2.4.1 Example of a race.

memory locations. The Boolean, B, depends on values local to each processor, thus may be true for one processor and false for the other. Hence a race can develop between the processors: depending on the relative times required to evaluate f and g, either the old or new value of, say, x may be used by the else clause. The code generated is a legitimate P-RAM program; however, for our purposes this program is not very useful because it becomes hard to argue about its precise behavior. If a statement such as that above were to occur in a parallel pidgin Algol program, we shall adopt the convention that the old values of both x and y must be used in the computation, as if the program had been written as in Figure 2.4.2

```
for processors p do
  if B(p)
    then tx := f(y)
    else ty := g(x)
  fi
  wait for both processors to reach here;
  if B(p)
    then x := tx
    else y := ty
  fi
od
```

Figure 2.4.2 Transformation that removes a race.

Transformations of this form possibly cannot be given in general so that the running time of the program is affected only by a constant factor. However, all of the parallel pidgin Algol programs in this thesis will have simple enough transformations that the order of their running times will remain unchanged when race conditions are removed.

The transformed program in Figure 2.4.2 above points out another characteristic needed to prove properties of parallel pidgin Algol programs: synchronization. The wait between the two conditionals is necessary if the use of the old values of both x and y is to be guaranteed. Although the P-RAM model itself allows far more opportunities for parallelism, we shall restrict our parallel programs as follows:

> If each processor in a set P begins to execute a statement S at the same time, all processors in P which do not halt within S will complete their executions of S simultaneously.

Given this restriction, the time to execute any parallel pidgin Algol statement S can be seen to be the maximum over all processors of the execution time of each processor on S plus the time to resynchronize processors at the conclusion of S. Often the resynchronization can be accomplished by a few judiciously placed null instructions, for example in padding out the <u>then</u> or <u>else</u> branch of an <u>if</u> statement, provided the execution times of the alternate paths can be computed at compile time. The general case requires that the number of processors entering and leaving S be matched, necessitating a log (number of processors) delay to count the processors entering and leaving S.

An example should help to clarify these concepts. The program in Figure 2.4.3 is a simple-minded program to set an

```
begin
  assign a processor to each element of A;
  for each processor p do
    while A(p) > 0 do
      A(p) := A(p) - 1
    od
  od
end
```

Figure 2.4.3 Unsynchronized array initialization program.

array A of non-negative integers to all zeros. The running time of the "<u>for</u> each processor" statement in this case is $O(M + \log |A|)$, where $M = \max_i \{A[i]\}$. The log |A| term reflects the time required to synchronize the processors at the conclusion of the statement. A similar program for the same purpose is shown in Figure 2.4.4, assuming that M is somehow known ahead of time. The "for each processor" statement now

```
begin
  assign a processor to each element of A;
  for each processor p do
    for i:= 1 to M do
      if A(p) > 0
        then A(p) := A(p) - 1
        else idle
      fi
    od
  od
end
```

Figure 2.4.4 Synchronized array initialization program.

has a running time of just O(M), since synchronization at the termination of the for loop is guaranteed, in contrast to the while loop above where synchronization has to be explicitly tested.

Note the "idle" statement in the else clause of the second program. Since the time to execute the then clause is known, synchronization within the if statement can be accomplished by this simple device. As with the transformations designed to prevent race conditions mentioned earlier, the full generality of processor synchronization will seldom be required in the parallel pidgin Algol algorithms to follow in later chapters.

Chapter 3

The Computational Power of Parallel Computers

The main purpose of this chapter is to examine the power of the P-RAM model when it is viewed as an abstract model of computation. A secondary purpose is to examine some alternative parallel models to further investigate the appropriateness of the attributes chosen for the P-RAM model. Section 3.1 contains the main results of the chapter; they relate time complexity classes for deterministic and nondeterministic P-RAMs to Turing machine time and space complexity classes. These results originally appeared in [F2]. Section 3.2 investigates the effects of limiting the memory resources available to the P-RAM. Finally, in Section 3.3, parallel models that replace the global memory of the P-RAM by explicit interprocessor communication through a connection network are introduced. Results very similar to those in Section 3.1 are derived for several network topologies, supporting the contention that the P-RAM model is a reasonable basis for the study of parallel computational complexity. The reader is referred to [A1] for the definitions of Turing machine time and space complexity classes.

3.1 Deterministic and nondeterministic P-RAMs

This section is devoted to proving two theorems that completely characterize the power of deterministic and nondeterministic P-RAMs in terms of Turing machine time and space complexity classes. We say that a language L is in the class deterministic (nondeterministic) polynomial-time-P-RAM if there is a deterministic (nondeterministic) P-RAM M such that for all words x of length n, x is in L if and only if x is accepted by M and requires time at most T(n). Our two main theorems are presented below. Throughout this chapter, resource bounds unqualified by P-RAM refer to Turing machine computations.

<u>Theorem</u> 3.1.1 (deterministic P-RAMs) For $T(n) \geq \log n$,

$$\bigcup_{k=1}^{\infty} T(n)^k\text{-time-P-RAM} = \bigcup_{k=1}^{\infty} T(n)^k\text{-space.}$$

<u>Theorem</u> 3.1.2 (nondeterministic P-RAMs) For $T(n) \geq \log n$,

$$\bigcup_{c>0} \text{nondet-}cT(n)\text{-time-P-RAM} = \bigcup_{c>0} \text{nondet-}2^{cT(n)}\text{-time.}$$

For particular familiar complexity classes, these theorems say that deterministic P-RAMs can accept within polynomial time exactly the sets that Turing machines can accept within polynomial space and that nondeterministic P-RAMs can accept within polynomial time exactly the sets that nondeterministic Turing machines can accept using exponential time. The proof of Theorem 3.1.1 is given in Section 3.1.1, and the simulations establishing the truth of Theorem 3.1.2 are given in Section 3.1.2.

Before giving the proofs of the theorems, it is appropriate to compare these results to others that have been obtained for nontraditional machines modelling some aspects of parallelism. Several authors have constructed models for which deterministic and nondeterministic polynomial time are equivalent and are equal to PSPACE on a Turing machine. These include Hartmanis and Simon [H1] and Pratt and Stockmeyer [P2], who use RAMs augmented by instructions that can manipulate exponentially long numbers at unit cost; Kozen [K3] and Chandra and Stockmeyer [C2], who use alternating Turing machines (nondeterminism is subsumed by alternation, hence adds no power); and Savitch and Stimson [S2], who use parallel RAMs without global memory. In [G2], Goldschlager shows that the SIMDAG model described in Chapter 2 can accept in polynomial time exactly the sets in PSPACE, but he does not study the power of nondeterminism for that device. By our simulation presented in Chapter 2 and the theorems above, it can be seen that, to within a polynomial, the P-RAM and the SIMDAG possess the same power.

While we do not know that nondeterminism is more powerful than determinism on our model, settling the question would decide PSPACE = nondeterministic exponential time. Savitch [S3] has also proven a result similar to Theorem 3.1.2 for nondeterministic parallel RAMs without global memory augmented by list processing instructions that can manipulate exponential amounts of information in unit time. The power of the deterministic version of his machines is still open.

It is appropriate to note at this point that although we use the uniform cost criterion [C4] throughout this section, our results still hold (at most squaring the simulation cost) if the logarithmic cost criterion is used; this is in contrast to the results for the RAMs mentioned above that rely on the unit time manipulation of large numbers. Similarly, we could charge memory accesses at a cost proportional to the logarithm of the size of global memory with only a polynomial increase in computing time.

3.1.1 Proof of Theorem 3.1.1

The proof of Theorem 3.1.1 is here divided into two lemmas, one for each direction of simulation.

Lemma 3.1.1.1 Let L be accepted by a deterministic $T(n)$ space-bounded Turing machine M, for $T(n) \geq \log n$. Then L is accepted by a deterministic $c \cdot T(n)$ time-bounded P-RAM P, for some constant c.

Proof We shall first present a simulation of M by P that assumes that the value of $T(n)$ is available, then show how to remove this assumption. Given $T(n)$, P will construct a directed graph where each node represents a configuration of M. The number of configurations of M is bounded by $2^{d \cdot T(n)}$

for some d depending on M, hence so is the number of nodes. Leaving each node will be a single edge to the node of its successor configuration. Accepting configurations of M are their own successors. Thus there is a path from the initial configuration node to an accepting configuration node if and only if M accepts its input within T(n) space.

To build the graph, P first initiates $2^{d \cdot T(n)}$ processors in O(T(n)) steps, each holding a different integer, representing each possible configuration of M. Each processor then, in O(T(n)) time, unpacks its configuration integer into local memory, computes the successor configuration, and packs the result into a single integer. The graph is then stored in global memory, the address of a configuration node being the integer representing the configuration.

To find the endpoint of the path from the initial configuration, the following is executed iteratively. Each processor finds the successor of the successor of its configuration node, then stores that as its successor in global memory. After i iterations, each node has its descendant at distance 2^i stored as its successor. Since the terminal configuration is at distance at most $2^{d \cdot T(n)}$ from the initial configuration, only $d \cdot T(n)$ iterations are needed. As each iteration takes constant time, and the rest of the simulation O(T(n)), the total time is $c \cdot T(n)$, for some c.

To avoid the constructibility assumption about T(n), modify the above simulation as follows: P_0 will start processors one at a time that will execute the procedure above assuming $T(n) = 1,2,\ldots$. When any of these simulations succeed, P_0 will be notified and will accept. The order of the running time is unchanged, since it requires only $O(T(n))$ time to start the simulation that uses the correct guess for $T(n)$. The individual simulations must also be modified so that they each use disjoint locations in global storage to store their configuration graphs. This can be accomplished by increasing each address by the assumed value of $2^{T(n)}$. Details are left to the reader. $\square$

<u>Lemma</u> 3.1.1.2 Let L be accepted by a deterministic $T(n)$ time-bounded P-RAM. Then L is accepted by a $O(T(n)^2)$ space-bounded Turing machine.

<u>Proof</u> We shall first construct a nondeterministic $T(n)$ space-bounded Turing machine M that accepts L, then show how to make M deterministic without increasing the space bound.

In order to determine whether the P-RAM accepts its input, M needs to know the contents of P_0's accumulator when it halts and needs to verify that no two writes occur simultaneously into the same global memory location. The simulation is based on a recursive procedure ACC that checks the contents of a processor's accumulator at a particular time. By applying the procedure to P_0, we can determine if the P-RAM accepts. ACC is similar to the procedure FIND in [H1].

ACC will check that at time t, processor P_j executed the i^{th} instruction of its program, leaving c in its accumulator.

In order to check this, ACC needs to know

i) the instruction executed by P_j at time t-1 and the ensuing contents of its accumulator, and

ii) the contents of the memory location(s) referenced by instruction i.

ACC can nondeterministically guess (i) and recursively verify it. To determine (ii), for each memory location m referenced, ACC guesses that m was last written by some processor P_k at time $t' < t$. ACC can recursively verify that P_k did a STORE of the proper contents into m at time t'. ACC must also check that no other processor writes into m at any time between t' and t. It can do this by guessing the instructions executed by each processor at each such time, recursively verifying them, and verifying that none of the instructions change m. Checking that two writes do not occur into the same memory location simultaneously can be done in a similar fashion. For each time step and each pair of processors, ACC nondeterministically guesses the instructions executed, recursively verifies them, and checks that the two instructions were not writes into the same location.

The correctness of the simulation follows from the determinism of the P-RAM. In general, each instruction executed by the P-RAM will be guessed and verified many times by ACC. However, since the P-RAM is deterministic, at any step for each running processor there is exactly one instruction that can be executed; thus all verified guesses of that instruction must be identical.

To analyze the space requirements, note that there can be at most $2^{T(n)}$ processors running after $T(n)$ steps, so writing down a processor number takes $T(n)$ space. Since addition and subtraction are the only arithmetic operators, numbers can increase in length by at most one bit at each step. Thus, writing down the contents of an accumulator takes at most $T(n) + \log n = O(T(n))$ space. Writing down a time step takes $\log T(n)$ space and the program counter requires only constant space. Hence the arguments to a recursive call of ACC can be written down in $O(T(n))$ space. Cycling through time steps and processor numbers to verify that a memory location was not overwritten also takes only $O(T(n))$ space, hence the total space requirement at each level is $O(T(n))$. As the total depth of recursion is $T(n)$, the total space required is $O(T(n)^2)$.

Note that the simulation can be performed directly by a deterministic Turing machine. At each step in the simulation outlined above where a nondeterministic guess was made, M can deterministically cycle through all possible outcomes until the correct one is found. This requires no additional space. ▯

3.1.2 Proof of Theorem 3.1.2

As in the previous section, the proof of Theorem 3.1.2 is given by the two lemmas that follow.

<u>Lemma</u> 3.1.2.1 Let L be accepted by a nondeterministic $T(n)$ time-bounded Turing machine M. Then L can be accepted by a nondeterministic $c \cdot \log T(n)$ time-bounded P-RAM P, for some constant c,

<u>Proof</u> For an input of size n accepted by M, the length of an accepting sequence of configurations is at most $T(n)^2$. In $d \cdot \log T(n)$ steps, for some d, enough processors can be activated so that each can guess one symbol of the computation. The first n processors check that the initial configuration corresponds to the configuration of M on its input; each of the remaining processors verifies that its symbol follows from the corresponding symbol and the ones on either side of it in the preceding configuration. The processors for the last configuration must also check that it is an accepting configuration. In another $d' \cdot \log T(n)$ steps, for some d', the information that the computation is correct can be bubbled up to P_0 through the tree of activated processors, and P_0 can accept.

T(n) does not have to be constructible, since the length of the accepting computation can be guessed nondeterministically. □

<u>Lemma</u> 3.1.2.2 Let L be accepted by a nondeterministic $T(n)$ time-bounded P-RAM, $T(n) \geq \log n$. Then L is accepted by a nondeterministic $2^{c \cdot T(n)}$ time-bounded Turing machine, for some constant c.

<u>Proof</u> The brute force simulation of the P-RAM suffices. There can be at most $2^{T(n)}$ processors activated, each of which can access at most T(n) memory locations with addresses in the range 0 to $n \cdot 2^{T(n)}$. The contents and address of each memory location can be of size at most $T(n) + \log n = O(T(n))$, since numbers at most double at each step and P_0's accumulator originally contains n. Hence the total tape required is $2^{T(n)} \cdot T(n)^2$. The Turing machine can simulate one step of one processor in some constant number of scans of its tape; $2^{T(n)}$ processors with T(n) steps each takes time $2^{2 \cdot T(n)} \cdot T(n)^3 < 2^{c \cdot T(n)}$, for some c. $\square$

3.2 Memory limited computations

Since the proofs of Theorems 3.1.1 and 3.1.2 require numbers of memory locations and processors that are exponential in the parallel running time, it is natural to ask what happens when these resources are restricted to a polynomial in the running time. In the case of processors, such a restriction is not very interesting, since a polynomial number of processors running for polynomial time can be simulated by a single processor running for polynomial time. However, when the size of the global memory of the P-RAM is limited to a polynomial in the parallel running time, non-trivial results can be obtained.

We can characterize the power of nondeterministic P-RAMs with restricted storage and can partially characterize the power of similarly constrained, but deterministic, P-RAMs. The theorems below are stated for the familiar classes NP and PSPACE, but as with the earlier theorems in this chapter, similar results hold for higher and lower complexity classes. For the remainder of this section, all P-RAMs are assumed to possess only a polynomial number of global storage locations.

Theorem 3.2.1 The class of sets accepted by nondeterministic polynomial time-bounded, polynomial global storage-bounded P-RAMs is identically PSPACE.

Proof The key to the P-RAM's simulation of a PSPACE bounded Turing machine M is a recursive subroutine TEST(i,j,t) that verifies that M's configuration j follows from configuration i within d^t steps, for some constant d depending on M. TEST works by guessing the configuration k midway between i and j, then FORKing to execute TEST(i,k,t-1) and TEST(k,j,t-1) in parallel. The middle configuration k can be guessed using a processor's local memory, and parameters can be packed into the accumulator prior to the FORK, so no global storage is required to perform all of the parallel subroutine calls.

There is not sufficient global storage for each call of TEST to bubble its result back to its father as in Lemma 3.1.2.1, so an alternate strategy must be employed. When a processor determines that TEST(i,j,t) is false, it forces the P-RAM to reject by forking to create two sons, each of which does a store into global memory location zero. If TEST(i,j,t) is found to be true, the processor merely halts. The root processor P_0 computes D(n), an upper bound on the depth of recursion and calls TEST (initial configuration, final configuration, D(n)). It then waits long enough for D(n) levels of recursive calls to TEST to complete, and accepts. If P_0 accepts, then there exists a sequence of guesses of middle configurations k such that no processor finds a mistake or attempts a recursive call deeper than D(n), so M must accept its input.

D(n) may be taken to be a polynomial since M is PSPACE bounded. The processing at each call to TEST, exclusive of recursive calls, is proportional to the length of a configuration, a polynomial, so the P-RAM runs for time at most some polynomial in the length of the input.

To simulate a global memory limited, nondeterministic polynomial time-bounded P-RAM within NPSPACE (hence PSPACE), first observe that although exponentially many processors may be active at once, only polynomially many of them may write into global memory on any step of the computation. Thus, a nondeterministic Turing machine may guess and write down within polynomial space the entire contents of global memory after each step of the computation, along with documentation of which processors modified which storage locations at each step. The Turing machine can then traverse the tree of activated processors implied by FORK instructions and verify that all instructions executed are consistent with the guessed global memory contents and conversely. If this implied tree is traversed so that only the local memories of the processors directly on the path back up to P_0 are stored at any time, the simulation can be carried out within polynomial space. □

Theorem 3.2.2 The class of sets accepted by deterministic polynomial time-bounded, polynomial global storage-bounded P-RAMs contains the class co-NP.

Proof It is sufficient to show how to accept the complement of some NP-complete set deterministically on a memory limited P-RAM. We show how to accept the set of Boolean formulas that are not satisfiable. In c_1n time, 2^n processors can be activated, each holding a different integer in the range 0 to 2^n-1. In an additional c_2n time each processor can unpack its integer into its local memory and determine if the assignment of truth values represented by the bits of the integer cause the input formula to be satisfied. Processors finding a satisfying assignment force the P-RAM to reject as in Theorem 3.2.1. If no processor has forced rejection after $(c_1+c_2)n$ steps, P_0 accepts the input. □

Notice that the set of satisfiable Boolean formulas cannot be accepted by trying all truth values in parallel and setting a flag in global memory if any satisfying assignment is found, because there may be more than one such satisfying assignment, resulting in several processors trying to set the global flag simultaneously. If an NP-complete set could be found such that each member of the set had exactly one "certificate" (or only polynomially many), then memory limited deterministic polynomial time P-RAMs could accept at least NP union co-NP.

3.3 Network connected parallel computers

The main theorems of the previous sections use the global memory of the P-RAM model as a means of communication between arbitrary pairs of processors. The question naturally arises whether similar results can be obtained for a more restrictive form of interprocessor communication, such as that used by the ILLIAC IV. Recall that on this machine, processors may share data only with their neighbors on a rectangular grid and even then only when the data are explicitly routed from one processor to another. A generalized model of parallel computation based on ILLIAC IV may be realized by replacing the global memory of the P-RAM by a graph, where processors are located at the vertices and communication occurs only along edges. This motivates the definition of a network RAM, given below.

A network connected RAM (N-RAM) consists of a set of input registers, a communication network, and an unbounded set of processors. The input registers and instructions for reading them are the same as in the P-RAM model. The processors are also very similar to the processors of the P-RAM in that they each possess an unbounded local memory, an accumulator, and a program counter, and that they operate synchronously.

Acceptance or rejection of an input will be handled as before by the single designated processor that is active initially. Differences between the two models lie in their instruction sets and in the absence of global memory on an N-RAM.

Instead of a global memory, an N-RAM has two new instructions that implement interprocessor communication. For a word to be sent from one processor to another, one processor must execute SEND( ) while the other simultaneously executes RECEIVE( ). The parameter to SEND and RECEIVE specifies one of the possible communication links attached to the processor that executes the instruction. The structure of the communication network is implicit in the correspondence of these parameter pairs. In order for a message to be sent along a link, there must be a SEND executed at one end of it and a RECEIVE at the other end. An unmatched SEND or RECEIVE behaves as a null instruction. A corollary of this protocol is that a processor may communicate with only one other processor at a time; there is no "broadcast" facility. We shall assume that all communication links are bidirectional.

The meaning of the FORK instruction must also be clarified in the context of the N-RAM model. Execution of a FORK instruction on an N-RAM causes a processor adjacent to the processor executing the FORK to be activated and the number of the communication link that connects them to be made available to both parent and child for use in subsequent SEND and RECEIVE instructions. As a consequence, the number of neighbors of each processor must be unbounded, to permit a FORK to be executed at each unit of time.

For our applications, it will usually be necessary that processors be able to infer the link numbers that connect them to certain of their neighbors. For this reason, we shall use as the underlying communication network of the N-RAM only graphs having a very regular structure. The graph to be used throughout the next several subsections will be the binary n-cube, defined as follows. The 2^n vertices of an n-cube are labelled with binary vectors of length n and there are edges between those vertices whose labels differ in only one position. We shall take n to be the running time of the N-RAM program to allow the maximum number of FORKs. Using the n-cube connection pattern, FORKs executed at time t will activate processors whose labels differ from their activating processor only in the t^{th} bit position.

We shall also assume that the N-RAM may access processor numbers as well as individual bits of processor numbers. This last facility may be simulated by passing processor numbers as in Section 2.2 and unpacking them in time proportional to their length, possibly squaring the N-RAM running time. Another squaring of the running time results if the execution of SEND-RECEIVE pairs is charged at a cost proportional to the length of the value transferred instead of at unit cost. Since the applications we shall present for N-RAMs are fairly insensitive to polynomial changes in running time, the result of these squarings is only a slight weakening of our theorem that relates the power of nondeterministic N-RAMs to the power of nondeterministic Turing machines.

The remainder of this section presents N-RAM analogs for Theorems 3.1.1 and 3.1.2.

3.3.1 Deterministic N-RAMs

Running times and complexity classes are defined for N-RAMs in the same way as they were for P-RAMs earlier, so for deterministic N-RAMs, a theorem corresponding exactly to 3.1.1 can be derived. One of the simulations is not as tight as in the proof of the earlier theorem, but since a union over all polynomial-bounded complexity classes is involved, the statement of the theorem, given below, remains unchanged except for the substitution of N-RAM for P-RAM.

<u>Theorem</u> 3.3.1 (deterministic N-RAMs) For $T(n) \geq \log n$,

$$\bigcup_{k=1}^{\infty} T(n)^k\text{-time-N-RAM} = \bigcup_{k=1}^{\infty} T(n)^k\text{-space.}$$

<u>Proof</u> One direction of the proof is nearly trivial. The deterministic $T(n)^2$ space simulation of a deterministic $T(n)$ time-bounded N-RAM is nearly identical to the proof of Lemma 3.1.1.2. In the case of an N-RAM, the recursive procedure ACC can in fact be somewhat simpler, since write conflicts in a global memory do not have to be detected.

To simulate a deterministic space-bounded Turing machine within deterministic $O(T(n)^2)$ parallel time on an N-RAM, the recursive procedure TEST of Theorem 3.2.1 will be employed, with exhaustive testing replacing nondeterministic guessing of the middle configuration. As before, the depth of recursion of calls to TEST is $T(n)$, but to start up processors that try all possible $2^{T(n)}$ middle configurations requires an additional time $T(n)$ at each level of recursion. Constructibility assumptions about $T(n)$ are avoided by running the simulation in parallel for $T(n) = 1,2,\ldots$. □

Notice that the n-cube connection network is not really needed here, since the computation tree may be embedded in the n-cube of the N-RAM in the obvious manner, and the only communication links ever used are the links connecting parent and child processors at a FORK. Also notice that the technique of this proof does not provide an improvement of the power of memory limited P-RAMs given by Theorem 3.2.2 because although all of the calls to TEST can be made by the P-RAM, there is insufficient space in global memory to return the answers back up the tree.

3.3.2 Nondeterministic N-RAMs

For nondeterministic N-RAMs using the n-cube connection pattern, an exact analog of Theorem 3.1.2 can be given. The theorem is stated below.

Theorem 3.3.2 (nondeterministic N-RAMs) For $T(n) \geq \log n$,

$$\bigcup_{c>0} \text{nondet-}cT(n)\text{-time-N-RAM} = \bigcup_{c>0} \text{nondet-}2^{cT(n)}\text{-time}$$

Proof Once again, the Turing machine simulation of an N-RAM program follows the proof given earlier. The brute force simulation in Lemma 3.1.2.2 works for N-RAMs as well as for P-RAMs.

To simulate a nondeterministic $2^{T(n)}$ time-bounded Turing machine M using a nondeterministic $O(T(n))$ time-bounded N-RAM, a modification of the simulation used in the proof of Lemma 3.1.2.1 suffices. Recall that the four steps of this simulation were

i) $2^{2T(n)}$ processors, connected as a tree, are initiated,

ii) each processor guesses one symbol of an accepting computation of M,

iii) each processor verifies that its symbol follows from the corresponding symbol and the symbols adjacent to it in the preceding configuration, and

iv) the information that the guessed computation is correct is bubbled up to the root of the processor tree, which accepts the input.

Stages (i), (ii), and (iv) can be performed straightforwardly within $O(T(n))$ time using the techniques described earlier. The following lemma aids in the implementation of part (iii) of the simulation.

<u>Lemma</u> 3.3.2.1 Suppose each of the 2^N processors of a nondeterministic N-cube connected N-RAM wishes to send a message to some other processor whose number is known. Subject to the constraint that no processor send or receive more than one message, all messages can be delivered to their destinations within time O(N).

The proof of Lemma 3.3.2.1 will be postponed for now.

Given the lemma, part (iii) of the desired simulation can be performed by some constant number of message passing iterations as follows. The N-RAM nondeterministically guesses T(n) and makes the value available to each processor during step (i). Each processor p sends messages first to processor p-T(n)-1, then to p-T(n), and finally to p-T(n)+1 requesting a copy of the guessed symbol of the computation stored in those processors. Once the return messages have been received, p can determine whether its guessed symbol follows from the corresponding symbols in the previous configuration and go on to part (iv). Generalized message passing is not required to verify that the initial and final configurations are correct, since communication along the edges of the processor tree is sufficient. The total simulation time is O(T(n)), as claimed. ▯

<u>Proof</u> of Lemma 3.3.2.1 The proof that messages can be nondeterministically routed on an N-cube in O(N) time consists of three parts. We shall

i) show that messages can be nondeterministically routed quickly from the inputs to the outputs of a certain graph G,

ii) give a transformation of G that preserves its message passing characteristics, and

iii) demonstrate that the transformed graph is a subgraph of the binary N-cube.

The graph G will be chosen so that (i) will be obvious.

A 2^N-permutation network is a directed graph with 2^N input vertices and 2^N output vertices such that for each of the $2^N!$ permutations of inputs to outputs there exists a set of vertex-disjoint paths from inputs to outputs realizing the permutation. In [W2] it is shown how to construct a 2^N permutation network with the additional properties that

i) each vertex has in and out degrees bounded by two, and

ii) the network contains $O(N \cdot 2^N)$ vertices, arranged as $O(N)$ levels of 2^N vertices each. The only edges connect vertices of each level to vertices of the next level.

The construction in [W2] uses many copies of a graph known as a switch, shown in Figure 3.3.2.1. A 2^1-permuter is a single switch. A 2^N-permuter is built recursively from two 2^{N-1}-permuters as shown in Figure 3.3.2.2. The correctness of the construction is given in [W2], along with an algorithm for finding the disjoint paths realizing any given permutation.

Figure 3.3.2.1 A switch and its schematic representation.

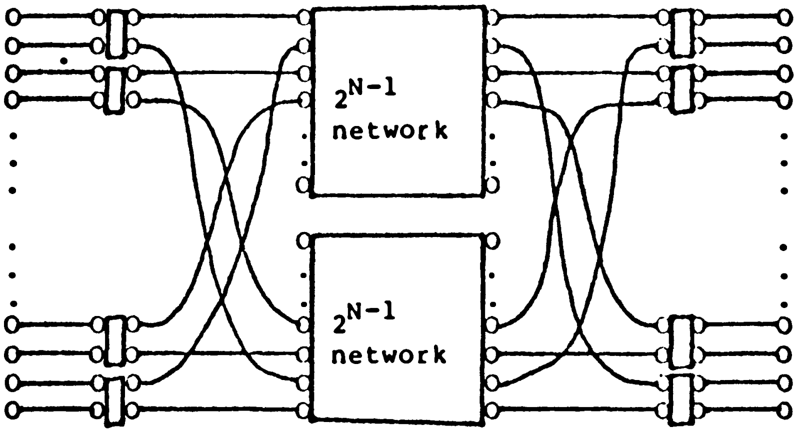

Figure 3.3.2.2 Construction of a 2^N permutation network.

Since our routing algorithm is to be nondeterministic, we require only the existence of the sets of disjoint paths, and do not have to find them explicitly. By identifying the inputs with the outputs of the permutation network, its message passing abilities are clear. It remains to show how to embed the network in an N-cube.

Part (ii) of the proof of Lemma 3.3.2.1 is motivated by the following observations. The vertices of a 2^N permutation network are arranged as 2^N rows of O(N) vertices each. Shifting a set of messages through the network can be done in O(N) steps by simultaneously moving all messages from one level to the next at each step. Thus, the levels of the permutation network correspond to units of time. If the vertices in each row of the permutation network were collapsed together, all of the interprocessor connections required to realize any permutation would still exist.

In Figure 3.3.2.2, to minimize the number of edges in the collapsed network, as many of the edges as possible connecting the outputs of the first tier of switches to the inputs of the smaller permutation networks were drawn horizontally. Such horizontal edges occurring in a path connecting an input to an output correspond in the collapsed network to a processor that does not reroute a message at a particular time step. Another feature of the way the permutation network was drawn above is symmetry: every directed edge in the collapsed network is accompanied by a directed edge of the opposite orientation, thus both directed edges can be replaced by a single undirected edge. Simultaneous passing of messages in both directions along an undirected edge can be modelled by serializing the messages. Figure 3.3.2.3 shows collapsed versions of a switch and of a 2^N-permutation network.

Finally, the collapsed representation of the permutation network can easily be seen to be a subgraph of a binary N-cube. In Figure 3.3.2.3, a 2^N input permutation network is built from two 2^{N-1} input networks plus edges connecting every other vertex of the smaller networks. An N-cube is built identically, except that every corresponding pair of vertices of the smaller cubes is joined by an edge, rather than every

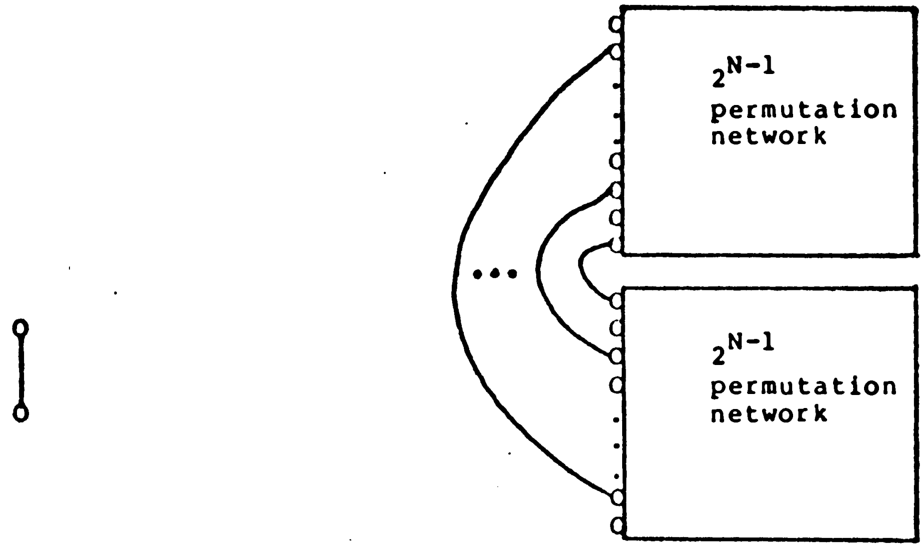

Figure 3.3.2.3 Collapsed switch and permutation network.

other pair.

We have shown that in some sense, a binary N-cube contains a 2^N-permutation network. Thus, by using at most half of the available communication links, a nondeterministic N-RAM can perform message passing (or sorting) on an N-cube in the desired O(N) time. ▯

Although Lemma 3.3.2.1 is sufficient to prove Theorem 3.3.2, it is instructive to consider a deterministic version of the lemma. We shall describe a deterministic algorithm capable of routing 2^N messages on an N-cube within $O(N^3)$ time. The deterministic message routing algorithm will employ the technique of divide and conquer. Initially each processor contains in its local memory at most one message, whose destination bears no relation to the number of the processor holding the message. On each of N iterations, messages will be passed to an adjacent processor so that there is agreement between one more bit of the destination address and the number of the processor holding the message. The algorithm is given in Figure 3.3.2.4.

As shown in the figure, after messages are swapped some

```
begin
  for i := 1 to N do
    for each processor p do
      if bit i of the message destination does not match
           bit i of p
        then send message to the processor connected to p
             along dimension i
      fi
    od
    /* Destinations and processor numbers now match in
         bit positions 1,2,...,i.  By shuffling messages
         around in the N-i cube obtained by varying the
         last N-i bits of processor numbers, make sure
         that no processor holds more than one message */
  od
end
```

Figure 3.3.2.4 Deterministic message routing in an N-cube.

processors may end up holding two messages, so a balancing procedure must be performed before the next iteration may commence. There are at most 2^{N-i} messages situated at the vertices of each N-i cube, which may be redistributed deterministically in time $O((N-i)^2)$ (or nondeterministically in time $O(N-i)$) by the divide and conquer algorithm of Figure 3.3.2.5. It remains only to show that redistribution among lower dimensioned subcubes at each step of 3.3.2.5 can always be performed (deterministically) without causing any processor to

```
begin
  for j := i+1 to N do
    divide each N-j cube into two N-j-1 cubes along the
           j^th dimension and balance the number of messages
           held in each smaller cube
  od
end
```

Figure 3.3.2.5 Deterministic message redistribution.

hold in excess of two messages at once.

Call the subcubes that differ in their j^{th} dimension L and R, containing n_L and n_R messages, respectively, and $\frac{n_L + n_R}{2}$ vertices each. Consider the edges connecting the two subcubes. There are only nine types of edges, depending on the number of messages initially at the left and right end-points of the edge. Let x_{ij}, $i,j = 0,1,2$, be the number of each kind of edge, where the left endpoint processor holds i messages and the right endpoint processor holds j messages. We have

$$n_L = x_{10} + 2 \cdot x_{20} + x_{11} + 2 \cdot x_{21} + 2 \cdot x_{22} + x_{12} \text{ and}$$

$$n_R = x_{01} + 2 \cdot x_{02} + x_{11} + 2 \cdot x_{12} + 2 \cdot x_{22} + x_{21}.$$

Without loss of generality, suppose $n_L \geq n_R$. Then the maximum number of messages that can be moved from L to R without forcing a processor to hold more than two messages at once is

$$n_{L \to R} = x_{10} + 2 \cdot x_{20} + x_{11} + x_{21}$$

By simple arithmetic, it can be verified that

$$n_L - n_{L \to R} \leq n_R + n_{L \to R}$$

thus the required redistribution is always possible.

Using nondeterminism, this redistribution can be performed in constant time. To redistribute the messages deterministically the values n_L and n_R must be computed and used to "steer" the redistribution, using the techniques to be presented in Section 4.1.5. These computations require an additional $O(N-i)$ time, for a total of $O(N^3)$ for the entire deterministic message routing algorithm.

Several corollaries readily follow from the above proofs. Results identical to Theorems 3.3.1 and 3.3.2 can be derived for N-RAMs using connection patterns other than the binary n-cube. The collapsed form of the permutation network described above is clearly one of these and so is a collapsed form of the "butterfly" graph employed by the Fast Fourier Transform [A1], since the latter connects in successive stages vertices whose labels differ in each bit position.

Chapter 4

Computational Structures

The designer of a parallel algorithm is confronted with additional choices that do not face the developer of a sequential algorithm. In addition to the ordinary concerns about the method and data structures to be used, the author of a parallel program must also worry about the assignment of processors to the various objects in the program. For example, a parallel algorithm to add n numbers together uses trivial data structures, a few scalar variables per processor, but the order in which these processors operate on their data can become quite involved, as will be shown in Section 4.1.1. We use the term computational structures to mean the combination of data structures, algorithms that manipulate them, and processor assignments.

4.1 Basic Computational Structures

In the study of parallel algorithms, there are several computational structures that occur so frequently that it is worthwhile to factor their analysis into a separate section. These structures include:

i) computing the sum (or any other associative function) of n elements,

ii) counting the number of elements in a linked list,

iii) converting a list of elements from a linked representation to an array representation,

iv) deleting blocks of elements from a linked list,

v) compressing a sparse array, and

vi) initializing a block of storage to a given value.

4.1.1 Computing the sum of n numbers

Perhaps the most basic non-trivial parallel operation is exemplified by computing the sum of n numbers. Since the addition operator is associative, we are free to organize the computation in a manner yielding the smallest parallel running time. Assuming first that n is a power of 2, we can achieve a time bound of log n using $\frac{n}{2}$ processors as follows: Each processor will compute the sum of two array elements in unit time. At the second step, each of $\frac{n}{4}$ processors will compute the sum of two sums from step 1, etc., until one processor computes the complete sum at step log n. If n is not a power of 2, $\lfloor\frac{n}{2}\rfloor$ processors suffice to compute the sum in time $\lceil \log n\rceil$, by an obvious generalization. Any associative operation could have been used instead of addition; min, union, or, etc. will be used in later algorithms. A parallel pidgin Algol program to express this algorithm would involve processors examining bits of their processor numbers to determine whether they are active on a given step; such programming details are not of interest and will not be given.

Another way of viewing the computation of the sum above would be to consider it as a binary tree of minimum height with the n elements to be summed at the leave . . internal nodes marked with the addition operator. This viewpoint allows us to derive more general resource bounds. Suppose $p \le \frac{n}{2}$ processors are available. Then to compute the bottom level sums would require at most $\lceil \frac{n}{2p} \rceil$ time, the next higher level would require at most $\lceil \frac{n}{4p} \rceil$, etc., for a total time T satisfying

$$T \le \lceil \frac{n}{2p} \rceil + \lceil \frac{n}{4p} \rceil + \ldots + \lceil \frac{n}{2^{\lceil \log n \rceil} p} \rceil$$

$$\le (1+\frac{n}{2p}) + (1+\frac{n}{4p}) + \ldots + (1+\frac{n}{2^{\lceil \log n \rceil} p})$$

$$= \lceil \log n \rceil + \frac{n}{p} \cdot (\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{2^{\lceil \log n \rceil}})$$

$$\le c_1 \cdot \log n + c_2 \cdot \frac{n}{p}$$

$$= O(\log n + \frac{n}{p})$$

Notice that T is O(log n) for $p = \frac{n}{\log n}$, so a running time of the same order as the simple parallel addition program can be obtained with a logarithmic factor fewer processors.

4.1.2 Counting the number of elements in a linked list

To count the elements in a list, we shall employ a technique called doubling, which will appear in some form in most of the algorithms to follow. The essential idea is that the processor associated with each element of the list will maintain a pointer to an element farther along the list, and at each step will "double" the distance its pointer spans by setting its pointer to the value of the pointer belonging to the element pointed to on the previous iteration.

Algorithm 4.1.2

Given a list L, linked together via next fields, with end of list indicated by a null next field, compute |L|. It will be convenient to treat the head of the list as the zeroth element of L. The program is given in Figure 4.1.2.1.

```
begin
  assign a processor to each element of L
  assign the processor named head to the list header
  for each processor p (including head) do
    far(p)  := next(p);
    span(p) := 1;
    while far(head) is not null do
      if far(p) is not null
        then span(p) := span(p) + span(far(p));
             far(p)  := far(far(p))
      fi
    od
  od
end
```

Figure 4.1.2.1 Finding the length of a linked list.

The correctness of the algorithm follows from the observations that

i) the following is an invariant of the while loop:

$\forall p$, $far(p) = next^{span(p)}(p)$, and

ii) far(head) will eventually become null.

When far(head) is null, span(head) will be the number of links in L, including the final null link, so $|L| = span(head) - 1$. To analyze the running time of the algorithm, notice that span for each processor is computed as the sum of two spans from the previous iteration, except possibly for the last, when the null link at the end of L is discovered. Thus, span(head) doubles on each iteration, so the running time is therefore $\lceil \log(|L| + 1) \rceil = O(\log |L|)$.

As in Section 4.1.1, a natural question to ask is whether the above algorithm can be made more efficient, i.e. whether it can be made to run in the same time using fewer processors. Here the answer appears to be no, at least for the following simple modification. Suppose the $p < |L| + 1$ available processors are assigned to the list elements such that each list element executes the body of the <u>while</u> loop once before any element is allowed to execute the body again. The modified program is still correct; the same invariant holds, but it requires time $\lceil \frac{|L|}{p} \rceil$ to insure that each processor at least doubled its span (although some of the processors might more than double their spans, depending on the actual assignment of processors to elements). The total running time becomes $O(\frac{|L|}{p} \cdot \log |L|)$. Since p is included multiplicitively rather than additively in the running time, no gain in efficiency results.

4.1.3 Converting from a linked list to an array

Given a linked list L of elements, the problem of storing the elements contiguously in an array A while preserving the order of L is equivalent to the problem of determining, for each x on L, the position of x in the list. Once the position numbers are known, they can be used as subscripts into A to produce the array in constant time using $|L|$ processors. To compute the position numbers a simple modification of the previous algorithm suffices. Note that in Algorithm 4.1.2 the distance from an element of a linked list to the end of the list is computed for all list elements, not just the list header. From the computed distances and $|L|$, the position of each element in the list can be determined. The running time and processor requirements are the same as for Algorithm 4.1.2, namely $O(\log |L|)$ and $|L|$, respectively.

Observe that the inverse operation, converting from an array to a linked list, can be done trivially in constant time by assigning a processor to each array element to form the links, provided that n processors have previously been initiated. We shall frequently make such an assumption, since for many of our algorithms the processor startup cost is incurred only once, at the beginning of the algorithm.

4.1.4 Deleting elements from a linked list

Suppose we are given a linked list that has a processor assigned to each element of the list. Simultaneously, all processors decide whether to delete their element from the list, and some elements are marked as deleted. Linking around the deleted segments of the list is nontrivial, since there may have been a large block of consecutive elements all deleted at the same time. The following algorithm uses a doubling strategy to locate the boundaries of a block of deleted elements and performs the relinking that deletes the appropriate elements from the list.

<u>Algorithm</u> 4.1.4

Given a list L, circularly doubly linked by next and last pointers and given a predicate <u>deleted</u> that tells whether an element is to be removed, relink L. The list head will be the zeroth element of L, with deleted(head) = <u>false</u>. We identify the processor associated with an element and the element itself where no confusion can result. The program is shown in Figure 4.1.4.1 below.

For deleted list elements p, far(p) will always point to a deleted element farther along the list. Let $b \leq |L|$ be the size of the largest block of consecutive deleted elements in L, and let p_0 be the processor associated with the leftmost deleted element in a deleted block of size b. Within $\lceil \log b \rceil - 1$ iterations of the <u>while</u> loop, the doubling of far will have made far(p_0) point to the rightmost deleted element in the block, so p_0 can link around the deleted segment of L. The following two final observations confirm the correctness

```
begin
  for each processor p (including head) do
    if deleted (p)
      then
        far(p) := if deleted(next(p))
              then next(p)
              else p
        while far(far(p)) ≠ far(p) do
          far(p) := far(far(p))
        od
        if not deleted(last(p)) and
          not deleted(next(far(p)))
          then p is the leftmost in a block of deleted
               elements and far(p) is the rightmost
               deleted element in the block; p may link
               together the undeleted elements last(p)
               and next(far(p)), bypassing the deleted
               block
        fi
    fi
  od
end
```

Figure 4.1.4.1 Deleting elements from a linked list.

of Algorithm 4.1.4:

i) simultaneous deletions of blocks on both sides of an undeleted element e do not interfere with one another, since the two deletions change different pointer fields in e, and

ii) the processor that deletes a segment is determined uniquely since only processors for deleted elements are active and only one of these in each block may have its left neighbor undeleted.

The running time of this algorithm appears to be O(log b), where b was defined previously, and indeed, this is the running time before resynchronization costs are included. However, the program in Figure 4.1.4.1 provides an example of the need for synchronization, as discussed in Section 2.3. As written, processors will complete the while loop at different times and the processors associated with undeleted list elements must wait a variable amount of time before they may continue. Rewriting the algorithm as shown in Figure 4.1.4.2

```
begin
  for each processor (including head) do
    if deleted(p)
      then
        far(p) := if deleted(next(p))
                    then next(p)
                    else p
    fi
    for i := 1 to ⌈log |L|⌉ do
      if deleted(p)
        then far(p) := far(far(p))
      fi
    od
    if not deleted(last(p)) and
      not deleted(next(far(p)))
      then p links last(p) and next(far(p)) together
             around the deleted segment
    fi
  od
end
```

Figure 4.1.4.2 Synchronized list element deletion program.

provides solutions to both of these problems. By replacing the while loop by a for loop with a fixed bound, synchronization is guaranteed and this program may be implemented on a P-RAM by padding out else branches by a constant number of null instructions. The running time is $O(\log b + \log |L|) = O(\log |L|)$. Henceforth, program transformations similar to the preceding will not be given explicitly.

4.1.5 Compressing a sparse array

Given an array A of size n where only the elements $A[i_1]$, $A[i_2]$, ..., $A[i_m]$ are non-null, we wish to produce an array B such that for $1 \leq j \leq m$, $B[j] = A[i_j]$. One solution to this problem combines two of the previous algorithms: form a linked list of all the elements of A, then use Algorithm 4.1.4 to delete the null elements, then use Algorithm 4.1.3 to store the compressed list into B. This algorithm has a running time of O(log n) using O(n) processors. The algorithm we present here solves the problem with slightly fewer processors and illustrates a technique that will be useful later.

The crux of the problem is to compute, for each non-null element $A[i_j]$, the value of j. As in Section 4.1.1, the computation will be organized as a complete binary tree whose leaves are the elements of A. Unlike the earlier algorithm, however, information will flow in both directions along the edges of the tree, in a manner similar to the carry lookahead adder circuit.

<u>Algorithm</u> 4.1.5

Given an array A with null and non-null elements, compute j for each non-null element $A[i_j]$ using the program in Figure 4.1.5.1. The computation tree will be made explicit in this presentation, although it would not be in an efficient implementation. We assume that $|A| = n$ is a power of 2. The counts are computed exactly as in Algorithm 4.1.1, with information moving up towards the root. For a subtree rooted at p,

```
begin
  assign a processor to each leaf and internal node of
       the computation tree
  compute count(p) for each node p of the tree as the
       number of non-null elements of A that are descen-
       dants of p;
  first(root) := 1;
  last(root)  := count(root);
  for each processor p do
    for i := 0 to log n do
      if node p is at distance i from the root with
           count(p) > 0
        then
          if p is a left son of its father
            then first(p) := first(father(p));
                 last (p) := first(father(p)) +
                             count(p) - 1
            else first(p) := last(father(p)) -
                             count(p) + 1;
                 last (p) := last(father(p))
          fi
      fi
    od
  od
end
```

Figure 4.1.5.1 Compressing a sparse array.

first(p) and last(p) delimit the range of j values to be assigned to leaves of that subtree. These values are computed by working down the tree towards the leaves. By induction it can be verified that for the non-null leaf i_j, first(i_j) = last(i_j) = j upon termination of the algorithm. As presented, the resource requirements are O(log n) time and O(n) processors, but using the techniques of Section 4.1.1, the processor requirement can be reduced to $O(\frac{n}{\log n})$ without changing the time bound.

4.1.6 Initializing a block of storage

Clearly, an array A of size pT can be initialized in time T by p processors. For some applications, however, this is not sufficient. For example, consider a graph algorithm that requires an adjacency matrix representation of a sparse graph $G = (V,E)$. If $|E|$ is on the order of $|V|$ and a processor is available for each edge, then the naive algorithm requires at least $O(|V|)$ time to initialize the adjacency matrix. A generalization of Exercise 2.12 in [Al] allows us to perform the initialization in constant time. With each processor p that can store information into A is associated a stack in global memory and a pointer giving the extent of the valid stack entries in p's stack.

A value v is entered into A[i] by p_j by pushing (v,i) onto p_j's stack and linking A[i] to the new stack entry. Initialization consists of invalidating all stack entries by setting the local stack pointer for each processor to null and storing the default (initial) value for the array elements in a special global location. The program in Figure 4.1.6.1 may be executed to find the value of A[i]. The procedures for initialization, entering a value, and retrieving a value each require only constant time and as many processors as will ever access or store into A simultaneously.

```
begin
  if A[i] points to a valid stack entry for some proces-
       sor, say j
    then
      if the stack entry points back to A[i]
        then the value is taken from p_j's stack entry
        else use the default value
      fi
    else use the default value
  fi
end
```

Figure 4.1.6.1 Extracting the value of A[i].

4.2 Algorithms using basic computational structures

In this section we present parallel algorithms that solve common problems by the application of the computational structures given in the previous section. We first show how to compute the preorder, inorder, and postorder traversals of a binary tree, then give a parallel implementation of a structure similar to a priority queue.

4.2.1 Tree traversals

Since we are interested here in parallel algorithms, the traversal of a tree will be accomplished by assigning to each node its number in the traversal. The algorithms for preorder, inorder, and postorder are similar, and all depend on the ability to construct a particular representation of the tree quickly. We shall first assume that the tree is already in the proper form, then show how to remove this restriction.

In the desired representation, each node of the tree consists of a block of three pointers, left, right, and father. For any of the traversals, the pointers belonging to a node x

```
left(x) := if x has a left son l
     then address of the left pointer of l
     else address of the right pointer of x

right(x) := if x has a right son r
     then address of the left pointer of r
     else address of the father pointer of x

father(x) := if x is a left son of its father f
     then address of the right pointer of f
     else address of the father pointer of x
```

Figure 4.2.1.1 Data structure for tree traversals.

are defined in Figure 4.2.1.1. In addition, there is a special node head whose pointers are set to make the root of the tree a left son of head, and father(head) is set to null. A tree and its representation are shown in Figure 4.2.1.2. The links of this structure form a singly linked list, beginning with left(head), having the following properties:

i) the left pointer out of a node x is the beginning of a linked list that links through all the pointers of the left subtree of x before returning to the right pointer of x, and

ii) the right pointer out of a node x behaves similarly, traversing the right subtree of x and returning to the father pointer of x.

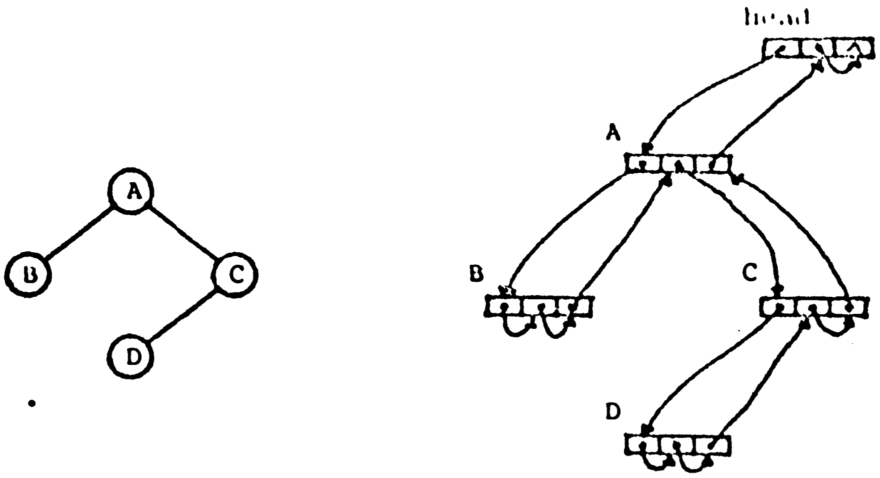

Figure 4.2.1.2 Example of data structure for tree traversals.

These assertions are easily proven by induction on the size of the tree using the definitions of left, right, and father.

The linked list that results defines the tree traversal <root> <left subtree> <root> <right subtree> <root>. By deleting all but the first, second, or third visit to <root>, we obtain preorder, inorder, or postorder traversals of the tree, respectively. Given the structure above for a tree T, the algorithm for a preorder traversal is shown in Figure 4.2.1.3. The algorithms for inorder and postorder traversals are identical except for the choice of nodes to delete. The running time is dominated by the time to run Algorithms 4.1.3 and 4.1.4, for a total of $\lceil \log(3\cdot(|T|+1))\rceil + \lceil \log(|T|+1)\rceil$, which is $O(\log |T|)$, using $O(|T|)$ processors.

The required data structure can be constructed in constant time from any representation of T in which nodes can determine their father and left and right sons in constant time. Suppose, however, that the given representation of T

```
begin
  assign a processor to each pointer field of each node
       of T;
  change the singly linked representation of T to a dou-
       bly linked representation in constant time;
  for all processors, mark as deleted the right and fa-
       ther pointers of each node in T;
  perform Algorithm 4.1.4 to link around deleted segments
       in the list;
  perform a variation of Algorithm 4.1.3 to assign
       numbers to the list of nodes (these are the output
       of the traversal)
end
```

Figure 4.2.1.3 Parallel preorder tree traversal algorithm.

contains only a father pointer for each node. Then the sons of a node are unordered, so we must show how to classify sons as left or right. This subproblem appears to be easy; one might expect to be able to solve it in time O(log 2) using |T| processors. However, the best algorithm we can give uses O(log |T|) time and O(|T|·log |T|) processors. Each node x will store the pair (x, father(x)) in an array, which is then sorted on its second component using the algorithm of [P3] to bring together the pairs belonging to nodes having the same father. In constant time, nodes can determine whether they are left or right sons of their fathers by examining their neighbors in the sorted array and set their links appropriately. The resource bounds follow from the time and processor requirements of the sorting algorithm in [P3].

4.2.2 Parallel pipelined MIN

In [A1], the problems on-line and off-line MIN are introduced. In either of these, a sequence of operations INSERT(i) and MIN is to be processed, where INSERT(i) adds the integer i, $1 \le i \le n$ to a set S and MIN computes the minimum of all elements of S and deletes that element from S. For the on-line version of the problem the answer to a MIN request must be produced before the next operation in the sequence is seen, while an off-line solution allows input of the entire sequence before any answers must be produced. A further possibility exists: reading of the sequence is allowed to continue without waiting for an answer to a MIN request, but answers cannot lag behind inputs by more than time f(n). We shall refer to this as the "pipelined MIN" problem, where the pipeline has size f(n).

For a parallel algorithm it is appropriate to allow multiple INSERTs or MINs to occur simultaneously, so the parallel pipelined MIN problem can be stated as follows. Given n processors, numbered 1 to n, at any time t any number of processors may receive INSERT and/or MIN commands. An INSERT command received by processor i means to insert i into the set S being maintained. No later than time t + f(n) the answers to the MIN commands received at time t must be produced. A MIN command received at time t is allowed to extract a value inserted at time t. For simplicity, we assume that none of the n processors receive more than one INSERT command or more than one MIN command during execution of the sequence of commands.

Example: Suppose n = 5 and f(n) = 2.

	t=1	t=2	t=3	t=4	t=5
input:	I(3) I(2) M(3) I(4)	I(1) M(4) M(2)	M(1) M(5)		
output:			M(3)=2	M(4)=3 M(2)=1	M(1)=4 M(5)=undefined

Figure 4.2.2.1 Parallel pipelined MIN example.

Note that the two MINs received at time t=2 in the example in Figure 4.2.2.1 could have produced either M(4)=3 and M(2)=1 or M(4)=1 and M(2)=3. We adopt the convention that the smaller min will be output by the lower numbered processor.

To solve the parallel pipelined MIN problem with a pipeline length of f(n) = O(log n), we shall use a generalization of the array compression algorithm of Section 4.1.5. As in the earlier algorithm, there will be a tree of processors above the n processors that receive the INSERT and MIN commands. Each processor in the tree will maintain insert_count and min_count arrays: counts of the total number of INSERTs and MINs received below it prior to or at each each unit of time. These counts are computed in the obvious way by propagating subtree counts up the tree towards the root. We shall assume that the count arrays are indexed by time units. Observe that, say, insert_count(root)[t] will not be correctly computed until time t + log n.

Two types of information will flow down the tree towards the leaves. The first of these will assign unique sequence numbers to each of the MIN commands received, by sending (first, last) pairs to the sons of a processor based on their min_count values, as in Algorithm 4.1.5. The sequence numbers will be cumulative.

The second type of information will also be MIN command sequence numbers, but instead of sending this information back to the processors that received the MIN commands, these data will be routed to the processors that received the inserted values that are the answers to the MIN requests. To effect this routing, an additional array, delete_count, is required, to allow each processor p to keep track of how many of the insert_count(p)[t] elements inserted below p prior to or at time t have been deleted by MIN commands. The number of elements in the subtree rooted at p that were inserted at time $\leq$ t and are eligible to be deleted by MIN commands received at time t is given by

insert_count(p)[t] - delete_count(p)[t-1].

For p = root, if this quantity is smaller than

min_count(root)[t] - min_count(root)[t-1],

then some of the MINs received at time t are unsatisfiable.

Using the (first, last) routing algorithm described in Section 4.1.5, the two sets of MIN command sequence numbers may be sent back towards the leaves of the processor tree. Eventually, both the processor that received the MIN command at time t and the processor that inserted the answer to the MIN command will receive the same sequence number. This sequence number can then be used as an index into a global array through which the answer can be passed.

The delay between receipt of a MIN request and output of its answer is the time for messages to travel from the leaves to the root and back to the leaves, or $2 \cdot \log n$. The number of processors required is $2n - 1$, or $O(n)$. The space required per processor is not proportional to the total time the algorithm runs, as it would first appear, but rather is proportional to log n, since at most $2 \cdot \log n$ of the entries in any of the three arrays for each processor can be active at any time, and these can be "folded" into a circular buffer of size $O(\log n)$. Thus, the total space required is bounded by $O(n \cdot \log n)$.

As a final observation, note that the above solution to the pipelined MIN can also be viewed as an $O(\log n)$ implementation of the on-line MIN problem. The memory required for the on-line version of this algorithm is $O(n)$, since only a constant amount of information needs to be kept at each node of the processor tree.

Chapter 5

Parallel Algorithms for Some Representative Problems

In this chapter we develop several useful design techniques for fast parallel algorithms to augment those of Chapter 4 and illustrate their use by constructing parallel algorithms for several problems. We shall introduce the technique of pairwise function composition, which is a generalization of the doubling strategy discussed in the last chapter. In addition, we shall describe data structures that rely on redundancy in their representation of some object, thereby enhancing the speed of a parallel algorithm that manipulates the object. The problems for which we shall develop algorithms are

i) computing string edit distances,

ii) the off-line MIN, and

iii) determining the connectivity of an undirected graph.

5.1 String edit distances

The string-to-string correction problem is the following. Given two strings x and y of symbols from a finite alphabet Σ, find the minimum cost of transforming x into y, where the allowable transformations are inserting a symbol into x, deleting a symbol from x, and replacing a symbol of x by another symbol from Σ. A sequential solution to this problem is given in [W1].

Throughout this section we shall write x as $x_1x_2...x_n$ and y as $y_1y_2...y_m$. Also, the substring $x_ix_{i+1}...x_j$ will be denoted x_{ij}, with the convention that if $j < i$ then x_{ij} is the null string ϵ. We say that the transformation of x to y is at <u>stage</u> (<u>i</u>,<u>j</u>) if the first i symbols of x have been read and the first j symbols of y have been output. Although this definition implicitly assumes a sequential transformation of x to y, we shall be interested in the cost of the transformation, not the actual sequence of edit operations; thus, parallel algorithms to compute the edit cost are meaningful. In [W1] the three editing operations were allowed to have arbitrary costs. We assume here that each operation has unit cost, since the generalization is straightforward.

The solution to the string-to-string correction problem in [W1] is based on dynamic programming. The costs for all possible stages of the transformation are placed in a matrix A[0:n, 0:m], where A[i, j] is the cost of transforming $x_{1,i}$ to $y_{1,j}$. The sequential algorithm, for unit cost edit operations, is given in Figure 5.1.1. At the termination of this algorithm, A[n, m] contains the minimum cost of converting x to y. The min computes whether it is cheaper to construct $y_{1,j}$ from $x_{1,i}$ by

```
begin
  for i := 0 to n do A[i, 0] := i od;
  for j := 0 to m do A[0, j] := j od;
  for i := 1 to n do
    for j := 1 to m do
      d_ij := if x_i = y_j then 0 else 1;
      A[i, j] := min { A[i, j-1] + 1,
                       A[i-1, j] + 1,
                       A[i-1, j-1] + d_ij
                     }
    od
  od
end.
```

Figure 5.1.1 Sequential string-to-string correction.

i) converting $x_{1,i}$ to $y_{1,j-1}$ and then inserting y_j,

ii) deleting x_i from $x_{1,i}$ and then converting $x_{1,i-1}$ to $y_{1,j}$, or

iii) converting $x_{1,i-1}$ to $y_{1,j-1}$ and then changing x_i to y_j.

The running time is obviously O(nm).

A fairly simple modification of the above algorithm yields a parallel algorithm whose running time is O(n+m) and which uses O(max(n, m)) processors. The same matrix A will be computed, but all elements A[i, j] having i+j=k will be filled in simultaneously, for k = 0,1,...,n+m. This is possible because in order to compute A[i, j], only elements of A whose subscript sum is less than i+j are required. The algorithm is given in Figure 5.1.2. This algorithm has the advantage that it could be implemented on a parallel computer without a global memory. Using O(nm) processors connected only to their neighbors on a rectangular grid, as in Illiac IV [Bl], the stated running time could be obtained by storing one element

```
begin
  for k:= 0 to n+m do
    assign a processor to each pair (i, j) such that
         i+j=k (this implicitly assigns values to i and j
         in each processor)
    if i = 0 then A[i, j] := j;
    else if j = 0 then A[i, j] := i;
    else
      d_ij := if x_i = y_j then 0 else 1;
      A[i, j] := min { A[i, j-1] + 1,
                       A[i-1, j] + 1,
                       A[i-1, j-1] + d_ij
                     }
    fi fi
  od
end
```

Figure 5.1.2 Linear parallel time string edit distances.

of the array A in each processor's local memory. However, if global memory is available, as in the P-RAM model, we shall show how to construct a O(log n·log m) time algorithm.

The faster parallel algorithm will compute the matrix A indirectly. The answer must be placed in A[n, m] and there appears at first to be no way to compute this value quickly, because the three surrounding values are not known. The key that makes the algorithm work is that although A[n, m] cannot be computed directly, a <u>function</u> that expresses A[n, m] in terms of other (unknown) elements of A can be computed efficiently. By computing similar functions for each element of A, then composing these functions together, an easily evaluated function yielding A[n, m] in terms of the (known) boundary conditions can be obtained.

Without loss of generality, assume that n+m is odd. The case where n+m is even is similar and will not be given.

Divide the elements of A into classes C_k, $0 \le k \le N$, for $N = \lfloor \frac{n+m}{2} \rfloor$, where A[i, j] is in class C_k if i+j=2k or i+j=2k+1. That is, classes consist of pairs of parallel antidiagonals. Notice that the elements of C_k can be computed if the elements of C_{k-1} are known, for fixed strings x and y. For example, A[n-1, m], a member of class C_N, may be expressed in terms of elements of class C_{N-1} as min [A[n-2, m]+1, A[n-1, m-1]+1, A[n-2, m-1]+$d_{n-1,m}$], where d_{ij} is zero if $x_i = y_j$ and one otherwise. Similarly, A[n, m] may be first expressed as a function of A[n-1, m]*, A[n, m-1]*, and A[n-1, m-1]. The starred terms are members of the same class as A[n, m], namely C_N, so we expand those elements in terms of elements of C_{N-1}, yielding:

```
A[n, m] = min [ 1 + min { A[n-2, m] + 1,
                          A[n-1, m-1] + 1,
                          A[n-2, m-1] + d_{n-1,m}
                        },
                1 + min { A[n-1, m-1] + 1,
                          A[n, m-2] + 1,
                          A[n-1, m-2] + d_{n,m-1}
                        },
                A[n-1, m-1] + d_{n,m}
              }
```

$$
\begin{aligned}
= \min \{ \; & A[n-2, m] + 2, \\
& A[n-2, m-1] + 1 + d_{n-1,m}, \\
& A[n-1, m-1] + \min \{d_{n,m}, 2, 2\}, \\
& A[n-1, m-2] + 1 + d_{n,m-1}, \\
& A[n, m-2] + 2 \\
\} &
\end{aligned}
$$

The form of this function for any member of a class C_k is very regular; it is the minimum for all $A[i, j]$ in C_{k-1} of $A[i, j]$ plus a constant, where the constant for each member of C_{k-1} depends on only the local structure of x and y. In fact, we shall show that the same functional form can be used to express any member of C_k in terms of members of any other class C_j, $j < k$. A compact way of representing the above function is by recording the constants in an array that is indexed by elements of C_{k-1}, with the constant ∞ stored in positions of the array corresponding to unreferenced elements of C_{k-1}. For this representation, we use the symbol $f_{ij}^{AB}(p,q)$ to be the constant depending on x and y such that

$$
\begin{aligned}
A[i, j] = \min \{ \; & \ldots, \\
& A[p, q] + f_{ij}^{AB}(p,q), \\
& \ldots \\
\}. &
\end{aligned}
$$

That is, $f_{ij}^{AB}(p,q)$ is the constant in the expansion of $A[i, j]$ that applies to $A[p, q]$, where $A[i, j]$ is in class C_A and $A[p, q]$ is contained in class C_B. We write f^{AB} to denote the set of all $f_{ij}^{AB}(p,q)$ with (i,j) and (p,q) restricted to the appropriate ranges, and if the classes are understood, we write

$f_{ij}(p,q)$. Another way of viewing $f_{ij}(p,q)$ is that it is the cost of going from stage (p,q) to stage (i,j). The key to the parallel algorithm to compute $A[n, m]$ is contained in the following lemma, which formalizes the above ideas.

<u>Lemma</u> For any classes C_A and C_C, $C < A$, and any $A[i, j]$ in C_A,

$$A[i, j] = \min_{(p,q)\in C_C} \{A[p, q] + f_{ij}^{AC}(p,q)\},$$

where $f_{ij}^{AC}(p,q)$ is defined by

if $A = C + 1$, $f_{ij}^{AC}(p,q)$ is found by expanding $A[i, j]$ in terms of elements of C_C, as was done in the above text to show $f_{nm}(n-2, m) = 2$, $f_{nm}(n-2, m-1) = 1$, etc.

if $A > C + 1$, $\forall B$, $C < B < A$,

$$f_{ij}^{AC}(p,q) = \min_{A[u, v]\in C_B} \{f_{ij}^{AB}(u,v) + f_{uv}^{BC}(p,q)\}$$

<u>Proof</u> The transformation of x to y proceeds one stage at a time, either increasing the length of the substring of x that has been seen, or increasing the length of the substring of y that has been produced, or both. At no time, however, can the sum of the lengths of the substrings increase by more than two from the previous stage. Therefore, beginning at stage (0,0) of the transformation and proceeding towards stage (i,j), at some point a stage (p,q) must be reached that is a member of class C_C, for $0 \le C < A$, since the width of a class is two. Thus, the form of the function for $A[i, j]$ is clearly the minimum over all such stages (p,q) in C_C of the sum of the costs of

getting to stage (p,q) and getting from (p,q) to (i,j). These costs are, respectively, A[p, q] and $f_{ij}^{AC}(p,q)$, as the lemma states.

It remains to show that the $f_{ij}^{AC}(p,q)$ are computed correctly, which we do by induction on A-C. The basis case, A-C=1, was given in the discussion preceding the lemma. For the inductive step, assume the lemma is true for classes separated by less than A-C. The expression $f_{ij}(p,q)$ is the cost of transforming the string $x_{1,i}$ to $y_{1,j}$ given that we have already transformed $x_{1,p}$ to $y_{1,q}$. By reasoning similar to that above, in passing from stage (p,q) to stage (i,j), some stage (u,v) in class C_B must be entered. The value of $f_{ij}(p,q)$ is the minimum over all such (u,v) in C_B of the sum of the costs of going from stage (p,q) to (u,v) and from stage (u,v) to (i,j); this is exactly what the definition of $f_{ij}^{AC}(p,q)$ in the statement of the lemma states. These costs are $f_{uv}^{BC}(p,q)$ and $f_{ij}^{AB}(u,v)$, respectively, which are computed correctly by the inductive hypothesis. $\square$

<u>Algorithm</u> 5.1

Given strings x and y, determine the minimum cost of transforming x to y using the operations defined above. The values $f_{ij}^{AC}(p,q)$ are computed for $A-C = 1,2,4,\ldots,2^{\lceil \log N \rceil}$ by the method suggested by the lemma. The answer will be $f_{nm}^{N,0}(0,0)$, since $A[n, m] = \min\limits_{A[i,\ j] \in C_0} \{A[i,\ j] + f_{nm}^{N,0}(i,j)\} =$

$\min\ [A[0,\ 0]+f_{nm}(0,0),\ A[0,\ 1]+f_{nm}(0,1),\ A[1,\ 0]+f_{nm}(1,0)] =$

min $\{f_{nm}(0,0),\ f_{nm}(0,1)+1,\ f_{nm}(1,0)+1\}$, and the cost $f_{nm}^{N,0}(0,0)$ includes the possibility of going through either of the stages (0,1) or (1,0) in C_0 during the transformation of x to y. The program in Figure 5.1.3 is a sketch of the detailed fast parallel algorithm.

```
begin
  assign a processor to each pair (i,j), 0≤i≤n, 0≤j≤m;
  for (i,j) in class C_A do
    Compute f_ij^{A,A-1}(p,q) for the maximum of five elements
         (p,q) in C_{A-1} that A[i, j] depends on, taking
         into account boundary conditions on A, i, j, p,
         and q;
  od
  for iter := 1 to ⌈log N⌉ do
    C := max (0, A - 2^iter);
    B := max (1, A - 2^(iter-1));
    for each (p,q) in C_C, compute f_ij^AC(p,q) as
            min      [f_ij^AB(u,v) + f_uv^BC(p,q)]
         (u,v)∈C_B
  od
end
```

Figure 5.1.3 Fast parallel string edit distance algorithm.

In a careful implementation of this algorithm, several efficiencies can be realized, in terms of both storage and processor requirements. First, the f^{AB} do not have to be computed for the indicated A and B at every iteration. Since $f_{nm}^{N,0}(0,0)$ is the only number of interest to be output, any f^{AB} that does not contribute to computing $f_{nm}^{N,0}(0,0)$ may be discarded. For example, on the first iteration of the for loop, only $f_{ij}^{k,k-2}$ for $k = n+m, n+m-2, \ldots, 3$ need to be computed, since they are the only values that will be referenced by later iterations. In general, on iteration i, the values of $f^{k,\max(0,k-2^i)}$ need to be computed only for those classes C_k such that n+m-k is a multiple of 2^i. Thus the number of active f^{AB} begins at O(nm) and approximately decreases by a factor of two at each iteration.

Complementing this decrease is the increase in the number of processors needed to find the min in the computation of $f^{AC}_{ij}(p,q)$, as well as the increase in the number of non-∞ values of $f^{AC}_{ij}(p,q)$ as A-C increases. In some portions of A, in the worst case, both of these latter quantities double at each iteration, being ultimately bounded by O(min(n,m)). A crude bound on the number of processors required is thus:

$$\leq nm\cdot(1 + 2^1\cdot 2^1\cdot\frac{1}{2^1} + 2^2\cdot 2^2\cdot\frac{1}{2^2} + \ldots + 2^{\lceil\log\min(n,m)\rceil}\cdot 2^{\lceil\log\min(n,m)\rceil}\cdot\frac{1}{2^{\lceil\log N\rceil}})$$

which is O(nm·min(n,m)). The running time is given by the number of iterations, $\lceil\log\lfloor\frac{n+m}{2}\rfloor\rceil$, or O(log max(n,m)), times the time required to find the minimum inside the loop, or $\lceil\log\min(n,m)\rceil$, for a total time that is O(log n·log m). Since the time for each iteration is dominated by the cost of the minimum, the techniques of Section 4.1.1 can be applied to reduce the number of processors by a factor of log min(n,m) without changing the order of the running time.

The space required to store the results of each iteration is bounded by O(nm), since the reduction in the number of active f^{AB} exactly balances the increase in the non-∞ terms that must be stored for each f_{ij}^{AB}. The non-∞ terms for each f_{ij}^{AB} form a contiguous block within the class C_B, so no storage need be allocated to store ∞ terms. The total space required is dominated by the space used to store temporary results on each iteration while computing the minimums, thus space O(number of processors) may be employed.

5.2 Off-line MIN

In Section 4.2.2, the problems on-line and off-line MIN were introduced and an algorithm was given for a generalization of the on-line version of the problem. In this section we show that if the entire sequence of INSERTs and MINs is made available at once, then all of the answers to MIN commands can be obtained in time $O(\log n)$ using $O(n^2)$ processors (which can be reduced to $O(\frac{n^2}{\log n})$ processors by the usual techniques). We shall assume that the entire sequence Q of INSERT(i) and MIN commands is available in an array located in global memory and that $|Q|$ is a power of two. The output will be placed in an array parallel to the input, with output[i] = j if input[i] is the MIN command whose answer is j. Unsatisfiable MIN commands will have null output values.

The parallel off-line MIN algorithm will operate in a manner reminiscent of the sequential algorithm for the same problem given in [A1]. The updating of a set S of integers is simulated in both cases from the point of view of the INSERT instructions. For each INSERT command, the MIN is located that extracts the value inserted. Whereas the sequential algorithm does this computation iteratively for each INSERT, the parallel algorithm to be described operates on all INSERTs simultaneously. We shall describe here the procedure executed to find which MIN extracts the value k, with the understanding that a copy of this procedure needs to be executed for each k for which there is a command INSERT(k) in Q.

Assuming that the command INSERT(k) appears somewhere in Q, we may write Q as Q_L INSERT(k) Q_R, or $Q = Q_L k Q_R$ for brevity. We use S(Z) to denote the contents of the set S being manipulated by the INSERT and MIN commands as it would appear after the command sequence Z has been processed sequentially. The answers produced for different values of k are interrelated, but the following lemma allow the computations for distinct values of k to proceed independently.

<u>Lemma</u> In determining which MIN in Q extracts the value k, INSERTs of values greater than k may be ignored without affecting the outcome.

<u>Proof</u> The proof is by induction on the number of INSERT(i) commands in Q with $i > k$. If there are no such INSERTs in Q, the lemma is true vacuously. Now suppose the lemma is true for sequences Q containing fewer than p INSERTs of values greater than k and let Q be a command sequence containing exactly p such INSERTs. We may write Q as $Q_1 k Q_2 m_k Q_3$, where k means INSERT(k) and m_k is the MIN that extracts k from S. Let

r be the value of the rightmost INSERT in Q of a value greater than k. There are now three cases, depending on whether r occurs in Q_1, Q_2, or Q_3. In each case, we seek to show that removing from Q some INSERT of a value greater than k leaves the answer to m_k fixed, allowing the inductive hypothesis to be applied to the shorter command sequence.

$r \in Q_3$: An INSERT of any value appearing in Q_3 clearly cannot affect earlier commands in the sequence. It therefore may be removed without changing the answer to m_k.

$r \in Q_2$: No MIN in Q_2 may extract a value greater than k or be unsatisfiable, since the value k is known to be in S until it is deleted by m_k. If the INSERT of r to S were ignored, m_k would still extract k.

$r \in Q_1$: If r is not extracted by a MIN somewhere in Q_1, the same argument as above for r in Q_2 shows that INSERT(r) may be deleted without affecting m_k. If r is extracted from S by a MIN within Q_1, then deleting INSERT(r) from Q_1 will either leave $S(Q_1)$ the same (if some MIN in Q_1 is made unsatisfiable) or decrease $|S(Q_1)|$ by one. In the latter case, $S(Q_1)$ will be lacking some value larger than r that was extracted by a MIN in Q_1. But the presence or absence of values $>r$ (hence $>k$) in $S(Q_1)$ does not affect the outcome of m_k, by the previous argument. ▯

The skeleton of an algorithm shown in Figure 5.2.1 outlines the parallel simulation of a command sequence $Q = Q_L k Q_R$.

```
begin
  delete all INSERT(i) commands from Q that have i>k;
  find the shortest prefix q of Q_R such that
      |S(Q_L kq)| = 0;
  the last command in q is the MIN that deletes k
end
```

Figure 5.2.1 Skeleton off-line MIN algorithm.

By ignoring any INSERTs of values greater than k, only the size of S is important, not the actual members, since all members of S other than k will be less than k and will be equivalent as far as determining which MIN extracts k from S.

The problem of finding which MIN command deletes a particular value k from S has been reduced to the problem of determining $|S(Z)|$ for command sequences Z that are prefixes of Q and contain no INSERTs of values greater than k. Alternatively, one could allow Z to contain INSERTs of large values but not count them when computing $|S(Z)|$. We shall adopt the second approach here. The size of S(Z) may be stated concisely by the following recurrence equation:

$$|S(e)| = 0$$

$$|S(Zc)| = \max\{0, |S(Z)| + \delta_c\}, \text{ where}$$

$$\delta_c = \begin{cases} 1 & \text{if } c \text{ is INSERT}(i) \text{ for } i \leq k \\ 0 & \text{if } c \text{ is INSERT}(i) \text{ for } i > k \\ -1 & \text{if } c \text{ is MIN} \end{cases}$$

For a general discussion of the parallel solution of recurrence problems, see [K2].

The essential structure of this recurrence problem can be captured by replacing each command c in Z by the corresponding δ_c value, using + to indicate +1 and - to indicate -1. Thus, for

$$Z = 2\ 3\ m\ 5\ m\ 6\ m\ 4,$$

we have

$$\delta(Z) = +\ +\ -\ +\ -\ 0\ -\ +,$$

for k = 5.

Because of the form of the recurrence equation involving the δ_c's, there is a compact representation for any string $\delta(Z)$ that summarizes the action of $\delta(Z)$ upon $|S(Z)|$. Whenever a minus follows a plus in such a sequence, both symbols may be cancelled out against one another. Repeated applications of this rule (and deleting zeros) will always reduce a string of pluses, minuses, and zeros to a canonical form: n minuses followed by m pluses, written (n,m). For the example above, we have

$$r[+(+-)(+-)(0)-+] \Rightarrow r[(+-)+] \Rightarrow r[+] = (0,1)$$

Note that (-+) pairs cannot be reduced; applying a MIN and then an INSERT to a set S_1 yields a set S_2 with $|S_2| = 1$ whether $|S_1|$ is 0 or 1.

Our algorithm for computing the effective representation of a string $\delta(Z)$ depends on the associativity of string concatenation and the following rule for the reduction operator r described above. If $r[\delta(x)] = (a,b)$ and $r[\delta(y)] = (c,d)$, then

$$r[\delta(xy)] = r[\delta(x)\delta(y)] = \begin{cases} (a,d) & \text{if } b=c \\ (a+(c-b),d) & \text{if } b<c \\ (a,(b-c)+d) & \text{if } b>c \end{cases}$$

This rule can easily be derived by writing $r[\delta(x)\delta(y)]$ as $-^{a}+^{b}-^{c}+^{d}$ and cancelling (+-) pairs in the interior of the string.

By performing the $r[\cdot]$ calculation in the form of a complete binary tree, the parallel time bound of $O(\log |Z|)$ can be achieved using $O(|Z|)$ processors. The example in Figure

```
Z  =   2     3     m     5     m     6     m     4
       +     +     -     +     -     0     -     +
     (0,1) (0,1) (1,0) (0,1) (1,0) (0,0) (1,0) (0,1)
        \   /       \   /       \   /       \   /
        (0,2)       (1,1)       (1,0)       (1,1)
            \       /               \       /
             (0,2)                   (2,1)
                  \                 /
                         (0,1)
```

Figure 5.2.2 Example of r[ ] calculation.

5.2.2 shows the tree built for the string ++-+-0-+. From the tree in Figure 5.2.2 one could immediately conclude that the effect of the execution of the command sequence Z would be a net increase of one on the size of set S for k=5 (since the root summarizes Z as (0,1)).

The skeleton algorithm given earlier requires knowledge of the size of S at points intermediate in the processing of the command sequence. This information can be extracted from the r[·] tree efficiently by the following procedure.

Algorithm 5.2.1
Suppose the r[·] tree described above has been constructed for some command sequence Q. Each node of the tree is labelled with a (n,m) pair. To obtain the number of minuses at a node, the program uses minus[node] and similarly for pluses. To find $|S(Z)|$ for Z a prefix of Q, walk from the root of the tree to the leaf corresponding to the last command in Z according to the program in Figure 5.2.3. The correctness of this algorithm follows from the construction of the r[·] tree and the fact that inside the while loop size is $|S(\delta(Y))|$ for Y the prefix of Z (and Q) to the left of the subtree rooted at position. The algorithm may be executed by a single processor in time proportional to the depth of the tree, or $O(\log |Q|)$.

The skeleton off-line MIN algorithm given earlier finds the shortest prefix q of Q_R such that $|S(Q_L kq)| = 0$. This operation can be decomposed into stages as follows:

```
begin
  let l be the leaf corresponding to the last command
       in Z;
  size     := 0;
  position := root of r[·] tree;
  while position ≠ l do
    if l is in the left subtree of the tree rooted at po-
         sition
     then position := left son of position;
     else size := max (
                       0,
                       size - minus[left son of position]);
          size := size + plus [left son of position];
          position := right son of position
    fi
  od
  adjust size by +1 or -1 according to the last command
       in Z;
  |S(Z)| is contained in size
end
```

Figure 5.2.3 Computing $|S(Z)|$.

i) build T_1 corresponding to $\delta(Q_L k Q_R)$,

ii) compute $|S(Q_L k)|$ by applying Algorithm 5.2.1 to T_1,

iii) build tree T_2 corresponding to $\delta(Q_R)$, and

iv) use the following procedure to locate the MIN command that deletes k from S.

Algorithm 5.2.2

Suppose that size has been initialized to $|S(Q_L k)|$ and that T_2 corresponding to $\delta(Q_R)$ has been built. At the termination of the algorithm in Figure 5.2.4, position will be the leaf of T_2 corresponding to the MIN command that extracts k from S. This procedure can be seen to be a simple modification of Algorithm 5.2.1 in which the search is guided by the location of the leftmost zero of $|S(Q_L kq)|$ rather than by the location of a

```
begin
  if minus[root] < size
    then k is never extracted
    else
      position := root of T2;
      while position ≠ a leaf do
        if minus[left son of position] ≥ size
          then position := left son of position;
          else size := size
                      - minus[left son of position]
                      + plus [left son of position];
               position := right son of position
        fi
      od
  fi
end
```

Figure 5.2.4 Finding q such that $|S(Q_L kq)| = 0$.

fixed leaf. Variable size keeps track of the size of S after processing every command to the left of the current subtree, given by position. The correctness of Algorithm 5.2.2 can be established easily by induction.

The running time of the parallel off-line MIN algorithm is given by the time to build and then traverse T_1 and T_2. These times are bounded by $O(\log |Q|)$ in each case. The only places where more than a single processor is required are in building T_1 and T_2, which require n processors (or $\frac{n}{\log n}$) to yield the claimed running time. Since the trees built are quite similar for different values of k, it is an interesting open problem to find a way to reuse the tree built for one value of k, or part of it, for other values of k.

5.3 Connectivity of undirected graphs

Much attention has been given by many authors to algorithms for various forms of graph connectivity [A1] [T1]. The problem appears to be fundamental, since the techniques developed to deal with connectivity problems have often been applied in other contexts. For example, the technique of depth first search used in sequential algorithms to test biconnectivity, strong connectivity, etc., also finds application to sequential algorithms for graph planarity [H5], dominators [T2], and so on. In the parallel domain the same claim cannot yet be made, since the techniques to deal with connectivity are still being devised. However, a few arguments can be presented for the importance of the parallel solution to the connectivity problem for undirected graphs.

Unlike the sequential case, parallel algorithms for undirected connectivity can be easily extended to (time) efficient algorithms for higher order connectivity. For example, as has been observed elsewhere [S1], by testing the connectivity of each of the graphs $G - \{v_1\}$, $G - \{v_2\}$, ..., for each vertex v in G, the biconnectivity relation of an undirected graph can be determined in time $O(C(n) + \log n)$, where $C(n)$ is the running time of the undirected connectivity algorithm. Algorithms with the same running time for determining k-connectivity, for any fixed k, can be obtained similarly. As a further argument, from the results of Chapter 3 showing that LOG space is contained in deterministic-LOG-parallel-time and the LOG space reductions in [J1], parallel algorithms running in time O(C(n) + log n) can be constructed for a variety of problems, including testing bipartiteness of graphs, testing satisfiability of certain propositional formulas, and determining whether a graph has a clique cover of size two. In addition, it seems likely that the computational structures used in solving the undirected connectivity problem will once again find wider application.

There have been three major approaches to connectivity algorithms. They are

i) deleting edges until a spanning tree remains,

ii) adding edges until a clique results, and

iii) collapsing vertices together until the entire graph has been compressed to a single vertex.

In any of the approaches, the goal is to verify that there is a path connecting each pair of vertices in the input graph G.

In the sequential case, an efficient algorithm (time O(n + e)) can be obtained by performing a traversal of the graph using a technique such as depth first search. This approach essentially builds a spanning forest of the graph by throwing away edges that provide redundant paths between pairs of vertices. The graph is connected if and only if the spanning forest is a tree. Depth first search does not appear to yield a practical parallel connectivity algorithm, because the best known parallel implementation requires time O(n) in the worst case [A2].

The second approach is used in [A4] to yield a parallel algorithm with a running time of $O(\log^2 n)$ using $O(n^3)$ processors, where n is the number of vertices in the graph. In this algorithm, whenever a path from v_i to v_j is discovered, the edge (v_i, v_j) is added to G. If this transitive closure process succeeds in producing the clique K_n, the original graph is connected.

The algorithm of [H3] uses a combination of the second and third approaches above to give a running time of $O(\log^2 n)$ using $O(n^2)$ processors. In this algorithm, cycles in a graph derived from G are located, the vertices on each cycle are collapsed into a single vertex representing the cycle, and the connectivity of the smaller graph is tested recursively. If the derived graph can be collapsed to a single vertex, G is connected. A transitive closure process also proceeds simultaneously to determine the connected components of the graph in the event it is not connected. The meaning of the phrase "collapsing vertex x into vertex y" is that any edges incident to x become incident to y instead, effectively removing x from the graph. Collapsing together of adjacent vertices obviously preserves the connectivity of the graph.

The algorithm to be described here uses a version of the third approach, vertex collapsing, to achieve the same running time as the two previous parallel algorithms, but uses $O(n+e)$ processors rather than $O(n^2)$ or $O(n^3)$, where e is the number of edges in G. For sparse graphs, including such important classes as planar graphs, this algorithm is more efficient than that of [H3].

Because of the method to be used, an algorithm slightly more general than required will be presented. The algorithm will operate on <u>multigraphs</u>, which are similar to directed graphs except that they are defined in such a way that they may contain multiple loops and/or multiple edges between pairs of vertices. To distinguish multigraphs from directed graphs, we use the terms <u>node</u> and <u>arc</u> to refer to the objects in multigraphs corresponding to vertices and edges in directed graphs. Each edge in the input undirected graph G will be represented by a pair of oppositely oriented arcs joining nodes in the multigraph that correspond to the endpoints of the edge in G. It should be clear that the transformed graph is connected if and only if the input graph is.

The outline of the node-collapsing algorithm is given in Figure 5.3.1. We must show that the algorithm can be made to work within the resource bounds mentioned earlier.

The first issue to dealt with is the form of the input. Since only O(n+e) processors are to be used, an input representation such as an adjacency matrix cannot be employed. The input representation we use is the <u>adjacency list matrix</u>, as defined in [A3]. The representation of an undirected graph G can be viewed conceptually as a matrix A, where row i of A contains the value j for each j such that there is an edge from vertex i to vertex j in G. Since the graph is undirected, each edge will appear twice. In addition, we assume that the number of edges incident to a vertex v is available as degree[v]. The important property of A is that the rows are

```
begin
  while not all nodes have been collapsed together and
        the graph is not known to be disconnected do
    identify sets of nodes that may be collapsed togeth-
         er;
    make all arcs incident from a set of collapsing nodes
         be incident from one representative of the set
         instead;
    update the "incident to" relation by making arcs
         point only to nodes that survived the collapsing
         process;
    eliminate arcs that are self-loops;
  od
  if the graph has been collapsed to a single node
    then the graph is connected
    else it is not
  fi
end
```

Figure 5.3.1 Skeleton connectivity algorithm.

left justified; for each row i, the elements A[i, 1], A[i, 2], ..., A[i, degree[i]] are non-null, and the remaining elements of row i are null. The left justification, together with the degree[ ] values, free the program that manipulates A from having to assign processors to non-existant edges. It is not necessary to use n^2 space to store A, since an array of pointers into a block of storage of size O(e) suffices; however, a later step of the algorithm will require $O(n^2)$ space, so we assume the simpler representation.

Given an undirected graph G in the form above, we now describe the data structure that will be used to represent the multigraph form of G. For each node there will be a circularly doubly linked list with a list header. The elements of the list will be of two types: arc elements and dummy elements. Besides the forward and backward pointers, each list element that represents an arc will contain the number of the node to which the arc is incident (the node it is incident from is given implicitly by the identity of the list on which the arc appears) and a pointer to its brother arc (the oppositely oriented arc representing the same edge). Dummy list elements will contain only a flag distinguishing them as dummies and the forward and backward pointers. Initially, arc and dummy elements will alternate on each list, with the first and last elements dummies. The purpose of dummy elements is to isolate arcs from one another on a list so that simultaneous updates can be made to adjacent arcs. Building this structure in constant time given the adjacency list matrix as input is simple, except for the links between brother arcs. Using an auxiliary matrix of size n^2 and the techniques of Section 4.1.6, each arc can store its address in the auxiliary matrix for its brother to read.

Let us now examine the implementation of the node collapsing operation given the linked arc-list structure described above. Suppose that the first arc list element on the list for node x is an arc incident to node y, and that we wish to collapse x into y. By the definition given previously, we must make all the arcs that were incident from x be incident from y instead. The arcs (x,y) and (y,x) should not appear in the collapsed graph, since the vertex x will no longer be part of the graph. In constant time, a single processor can replace the arc element (y,x) in list y with the entire list (arcs and dummies) belonging to list x, then delete the arc element (x,y). The brother pointer of (x,y) is used to determine where to insert the list and the header of list x is used to find the front and rear of list x.

The lists in the resulting structure will no longer alternate arc and dummy list elements; instead they will alternate arc elements with blocks of dummy elements of size at least one. Notice that the only fields in list y that must be changed by the collapsing operation are the forward and backward links of the dummy elements surrounding the arc element (y,x). This permits, for example, a simultaneous collapse into the arc element adjacent to (y,x) without interference. Suppose that a set of node collapsings is defined by a set S of arc lists as follows. Simultaneously, for each list x in S, if (x,y) is the first arc element on list x, collapse list x into list y. The following lemma characterizes the maximal sets S of simultaneous interference-free collapses supported by the arc-list data structure.

<u>Lemma</u> If the collapses defined by a set S of arc lists are not circular, then they may be performed in constant time using O(|S|) processors, yielding an arc-list structure in which arc list elements are separated from one another by blocks of dummy list elements of size at least one.

<u>Proof</u> The proof is by induction on |S|. For |S| = 1, the discussion above provides the basis case. Suppose the lemma is true for up to k simultaneous collapses, and consider the

collapses specified by a set S of arc lists of size k + 1. Since the collapses are acyclic, there must be some list x that is to collapse into a list y such that no list collapses into x. By the inductive hypothesis, the set of collapses defined by S - {x} can be performed simultaneously. In addition, the arc element (y,x) will not be collapsed into by any of the collapses in S - {x}, because only by collapsing its brother $(\dot{x},y)$ may (y,x) be removed from the list structure. Since the arc element (y,x) is surrounded by dummy elements, the collapse of list x into (y,x) may occur in parallel with the other collapses. Dummy list elements are never deleted, only moved, so arc list elements must remain separated from one another by dummies if they were so separated in the original structure. ▯

Shown below in Figure 5.3.2 is an example of the arc-list structure and a set of collapses. Dummy elements are shown as dots. The parentheses in the final list serve only to delimit the lists that are moved by the collapse operations and are not actually part of the data structure.

Having described the arc-list structure and how to perform the node-collapsing operation on it, only a few fairly simple details remain to complete the algorithm. We must show how to locate a set of arc lists defining a set of acyclic collapses large enough so that the algorithm will run sufficiently fast. To do this we employ the total ordering of the nodes implied by the node numbering. Letting S be the set of

```
          before collapses:
1:   . 4 . 2 .
2:   . 3 . 4 . 1 .
3:   . 2 . 4 .
4:   . 3 . 2 . 1 .

S = {1, 2, 4}
node 1 collapses into node 4 which collapses into node 3;
node 2 also collapses into node 3

          after collapses:
3:   . (. . 4 . 1 .) . (. . 2 . (. . 2 .) .) .
```

Figure 5.3.2 Example of arc-list structure and collapses.

arc lists whose first arc elements define collapses of nodes into lower numbered nodes, and $\overline{S}$ conversely, then one of S or $\overline{S}$ will define a set of collapses that will at least halve the size of the graph. Provided that the size of the first block of dummy elements on each list is of constant size, the set S may be enumerated in constant time and $|S|$ computed in time log n.

To insure that the block size condition holds, at the conclusion of each iteration duplicate dummy elements can identify themselves and delete themselves from their list by applying Algorithm 4.1.4. The "incident to" relation may be updated by recording each node that collapses into another, then applying a doubling procedure to determine, for each node x, upon which list x's arcs ended up after all simultaneous collapses are applied. If, after updating this relation, an arc becomes a loop, it and its associated dummy element delete themselves via Algorithm 4.1.4.

The two applications of Algorithm 4.1.4 are combined in the final program, which summarizes all of the concepts men-

```
begin
  assign processors to each node and arc;
  set C[x] := x for each node processor x;
  build the arc-list structure from the adjacency list
       matrix;
  for i := 1 to ⌈log n⌉ do
    count the number of low to high and high to low col-
         lapses;
    perform the most frequent type of collapse, setting
         C[x] := y if x is collapsed into y;
    for each node processor x do
      for j := 1 to ⌈log n⌉ do
        C[x] := C[C[x]]; od od
    update the head of each arc according to C;
    mark arc list elements deleted if they are loops;
    for each dummy list element d do
      if forward(d) is marked deleted or is a dummy ele-
           ment
        then mark d deleted
      fi
    od
    execute Algorithm 4.1.4 on the arc lists to remove
         all elements marked deleted;
  od
  if the graph has been reduced to a single node
    then it is connected
    else it is not, and each remaining node x represents
         a connected component, with the nodes y in the
         component having C[y] = x
  fi
end
```

Figure 5.3.3 Efficient parallel connectivity algorithm.

tioned above and is shown in Figure 5.3.3. The running time is dominated by several operations inside the main <u>for</u> loop that require O(log n) time, for a total of $O(\log^2 n)$, as claimed. There are O(n+e) processors required and $O(n^2)$ space.

It is interesting to speculate on the possibilities for improving the running time of Algorithm 5.3 to O(log n). One might hope to be able to do this based on the observation that collapsing a set of nodes together can be performed in time proportional to the log of the size of the set. For example, in collapsing together k nodes, the computation of the closure of C and the execution of Algorithm 4.1.4 require at most only $\lceil \log k \rceil$ steps until their answers have been computed. However, as the discussion following the presentation of Algorithm 4.1.4 points out, to remain in synchronization the processors must wait a full $\lceil \log n \rceil$ time. To circumvent this difficulty, the less synchronized algorithm of Figure 5.3.4 suggests itself. The idea here is to allow the arc-list struc-

```
begin
  while not done do
    perform some acyclic set of collapses, setting C[ ]
         appropriately;
    C[x] := C[C[x]];
    update the head of each arc according to C;
    mark list elements deleted as in Algorithm 5.3;
    execute one iteration of Algorithm 4.1.4;
  od
end
```

Figure 5.3.4 Less synchronized connectivity algorithm.

ture to only loosely represent the collapsed graph after each iteration. The much weaker invariant assertion is that if the program is halted at some point, the consistancy of the arc-list structure can be re-established within time O(log n) by one execution of the body of the main <u>for</u> loop in Algorithm

5.3.

There are a number of details to be cleared up concerning this algorithm, but they are moot because the skeleton algorithm suggested above cannot run in time O(log n). It is instructive to consider why not. The problem lies in the first statement inside the loop: selecting nodes to collapse. Since Algorithm 4.1.4 is not run to completion, there may be a large block of elements marked deleted at the beginning of a list, so it is not always possible to find arcs along which to collapse. If it could be shown that this deleted block either could not get too big or could not get big too often, the algorithm might still be shown to run quickly. The following example shows that this is not the case.

At the first step, nodes numbered k+1 collapse into nodes numbered k, for k=1,3,...,n-1. At the second step, nodes numbered k+2 collapse into nodes numbered k, for k=1,5,...,n-3. In the worst case, before any further collapses can occur, four arcs joining nodes k and k+1 with nodes k+2 and k+3, for k=1,5,...,n-3, must be deleted, requiring time log 4. At the next step, nodes numbered k+4 collapse into nodes numbered k, for k=1,9,...,n-7. This set of collapses causes all arcs that originally joined nodes k through k+3 with nodes k+4 through k+7 to simultaneously change from being parallel arcs joining the collapsed nodes k and k+4 to being loops at the collapsed node k. Eliminating these 16 loops in the worst case delays by time log 16 the next set of collapses. In general, the running time will be proportional to

$$\log 4 + \log 16 + \log 64 + \ldots + \log \left(\tfrac{n}{2}\right)^2$$

$$= 2\cdot(1 + 2 + 3 + \ldots + (\log n-1))$$

which is $O(\log^2 n)$, as before.

We see that the difficulty is redundant parallel arcs that slow down the algorithm once they become loops. Attempts to detect and delete parallel arcs when they first occur are faced with a fundamental problem: depending on the sequence of collapses, any two arcs potentially may become parallel. Thus, in testing whether to delete itself each arc must look at every other arc (or every arc with a lower address), resulting in a delay of at least $O(\log e)$. This is too slow to allow an improvement over Algorithm 5.3. The arc-list structure may be viewed as a method of overcoming a similar problem, that of being able to collapse nodes quickly although faced with the possibility of being required to collapse any arbitrary pair of nodes. It is thus still possible that some clever data structure, possibly combined with some preprocessing of the graph, may allow faster parallel arc deletion, and therefore faster parallel algorithms for undirected graph connectivity.

Chapter 6

Conclusions

6.1 Summary

In this thesis we have studied the computational complexity of synchronous parallel computers. Chapter 2 introduced the main parallel model used throughout the thesis, the P-RAM. The P-RAM model is intended to capture many of the features that may be found in the parallel computers of the not too distant future. It is not an entirely realistic model, but we believe it to be fairly reasonable, in the same sense that the RAM model for sequential computation closely approximates existing sequential computers over a moderate range of problem sizes. We showed that the precise characteristics chosen for the P-RAM are unimportant, provided the basic features of unbounded parallelism, synchronous operation, and a global memory are retained. Given this uniform model for parallel computations, we investigated two main areas within the field of parallel computational complexity. In Chapter 3, we explored the power of the P-RAM model viewed as an abstract computing device, and in Chapters 4 and 5 we studied techniques for developing efficient algorithms for parallel computers.

In Chapter 3, we were able to give concise characterizations of the power of deterministic and nondeterministic P-RAMs in terms of the more widely known space and time complexity classes for multi-tape Turing machines. Roughly speaking, time-bounded deterministic P-RAMs are equivalent in power to (can accept the same sets as) space-bounded Turing machines, where the time and space bounds differ by at most a polynomial. In the context of comparing models of computation, we consider such polynomial differences in resource bounds to be insignificant. Adding the feature of nondeterminism to the time-bounded P-RAM changes its power to that of a nondeterministic Turing machine with an exponentially higher running time. This latter result is perhaps somewhat surprising, in light of results by other authors for models expressing some of the aspects of parallelism in which nondeterminism provides no increase in power. Since little is known of the relationships between various deterministic and nondeterministic Turing machine time and space complexity classes, our equivalence results suggest another means through which some of the fundamental problems in computer science might be approached.

Chapter 3 also investigated the power of several variations on the basic P-RAM model. For example, we showed that by limiting the amount of global memory available to the P-RAM for use in interprocessor communication, the classes of sets accepted drop one level in the complexity hierarchy. The last section of Chapter 3 demonstrated that P-RAMs possess essentially the same power as a model that replaces global memory by explicit message passing, where the topology of the message network is given by a binary n-cube.

Chapter 4 played the role that a section on data structures might play in a paper about sequential computational complexity. We argued that for parallel algorithms, something more general than data structures are needed, to account for the assignments of processors to data in a parallel computer. The term computational structures was introduced to denote data structures and processor assignments and the mechanisms for manipulating them. We presented computational structures for a variety of simple subproblems that occur frequently in the design of fast, efficient parallel algorithms. Among the topics considered were tree traversals and several very basic operations on linked lists and arrays.

The purpose of Chapter 5 was to exhibit efficient parallel algorithms for several problems and, more importantly, to define the techniques that were applied to yield the algorithms. Three parallel algorithms were described, each having a running time proportional to at most the square of the logarithm of the problem size and each requiring a number of processors that is a polynomial function of the problem size. The algorithms were: computing minimum string edit distances, computing the off-line MIN, and finding the connected components of an undirected graph. The techniques used were, respectively, pairwise function composition, the structuring of a parallel computation in the form of a complete binary tree, and redundancy in the data structure used to represent an object. We can view pairwise function composition as a parallel version of dynamic programming. Structuring a parallel computation as a balanced tree should properly be called a computational structure; the off-line MIN algorithm, however, uses several different types of data simultaneously moving up and down the tree. Finally, redundancy in a data structure allows an improvement over the best previously known processor bound for the graph connectivity problem. Each of the techniques we described in Chapter 5 has been embedded in parallel algorithms for other problems by other authors; we hope that by extracting these techniques, they may be applied in the future to yield additional efficient parallel algorithms.

6.2 Future research

Our goal in this thesis has been to extend some of the results of sequential computational complexity theory to parallel models of computation. A long list of open problems may be generated by simply enumerating areas of study within computational complexity, since we have studied only a few out of the many possible topics. We shall mention several of these in particular.

This thesis did not consider the entire area of lower bounds on resource requirements for particular problems. For a general model of parallelism and problems of size n, only the bound $\Omega(\log n)$ is known. This lower bound corresponds to the time required to examine all n of the inputs, hence is trivial. Non-trivial lower bounds for the parallel solution of problems using a general model may be as difficult to obtain as bounds bigger than $\Omega(n)$ for sequential algorithms; however, it would still be interesting even to prove lower bounds for restricted parallel models, possibly even demonstrating the optimality of some non-trivial parallel algorithm for such models. A specific example of an interesting problem that may even have practical consequences is the inherent complexity of sorting depending on the number of processors available. See [K1] or [K4] for preliminary work in this area.

In the area of algorithms, a candidate for further work is depth first search. As mentioned in Chapter 5, the numbering assigned to a graph by depth first search permits efficient sequential algorithms for a variety of problems. If this numbering could be computed using parallel time, say, $O(\log^2(n+e))$, then many of these sequential algorithms could almost immediately be converted to efficient parallel algorithms.

A problem related to both algorithms and lower bounds is suggested by the frequency with which the parallel time bound of $O(\log^2 n)$ has appeared in this thesis. A natural question to ask is whether or not it is a coincidence that many of our algorithms have this as their running time. If not, there may be some deeper connection between the algorithms. It may be possible to define the notion of a complete problem for this parallel time complexity class and to find other members of the class. Alternatively, if there is no such deeper connection, it becomes reasonable to search for faster parallel algorithms for these problems.

Finally, the results of Chapter 3 provide alternate means by which the major open problems in computational complexity might be attacked. To cite just one example, if one could prove, as seems reasonable, that a single deterministic processor running for polynomial time was strictly less powerful than exponentially many processors sharing a polynomial-bounded global memory, then one would have shown proper containment of the class P in the class PSPACE, settling an important conjecture in the affirmative.

References

[A1] Aho, A., J. Hopcroft, and J. Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, Reading, Mass., 1974.

[A2] Alton D., and D. Eckstein, Parallel Searching of Non-sparse Graphs, Manuscript from Computer Science Dept., Univ. of Iowa, 1977.

[A3] Arjomandi, E., A Study of Parallelism in Graph Theory, Ph.D. thesis, TR-86, Dept. of Computer Science, Univ. of Toronto, 1975.

[A4] Arjomandi, E. and D. Corneil, "Parallel Computations in Graph Theory," SIAM J. of Computing 7, May 1978, pp. 230-237.

[B1] Barnes, G., M. Richard, M. Kato, D. Kuck, D. Slotnik, and R. Stokes, "The ILLIAC IV Computer," IEEE Transactions on Computers C17, August 1968, pp. 746-757.

[C1] Chandra, A., Maximal Parallelism in Matrix Multiplication, Report RC 6193, IBM Thomas J. Watson Research Center, September 1976.

[C2] Chandra, A. and L. Stockmeyer, "Alternation", Proc. of the 17th Annual Symposium of Foundations of Computer Science, Houston, Texas, Oct. 1976, pp. 98-108.

[C3] Chen, S., "Time and Processor Bounds for Linear Recurrence Systems with Constant Coefficients," Proc. of the International Conference of Parallel Processing, August 1976, pp. 196-205.

[C4] Cook, S. and R. Reckhow, "Time Bounded Random Access Machines," JCSS 7, 1973, pp. 354-375.

[C5] Csanky, L. "Fast Parallel Matrix Inversion Algorithms," SIAM J. of Computing 5, 1976, pp. 618-623.

[F1] Flynn, M., "Very High-Speed Computing Systems," Proc. IEEE 54, December 1966, pp. 1901-1909.

[F2] Fortune, S. and J. Wyllie, "Parallelism in Random Access Machines," Proc. of the Tenth Annual ACM Symposium on the Theory of Computing, San Diego, California, May 1978, pp. 114-118.

[G1] Gentleman, W., "Some Complexity Results for Matrix Computations on Parallel Processors," JACM 25, January 1978, pp.112-115.

[G2] Goldschlager, L., Synchronous Parallel Computation, Ph.D. thesis, TR-114, Dept. of Computer Science, Univ. of Toronto, December 1977.

[G3] Goldschlager, L., "A Unified Approach to Models of Syn- [illegible]

[H1] Hartmanis, J. and J. Simon, "On the Power of Multiplication in Random Access Machines", Proc. of the 15th Annual IEEE Symposium on Switching and Automata Theory, New Orleans, October 1974, pp. 13-23.

[H2] Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review 20, October 1978, pp. 740-777.

[H3] Hirschberg, D., "Parallel Algorithms for the Transitive Closure and the Connected Components Problems," Proc. of the Eighth Annual ACM Symposium on the Theory of Computing, Hershey, Pennsylvania, May 1976, pp. 55-57.

[H4] Hirschberg, D., "Fast Parallel Sorting Algorithms," CACM 21, August 1978, pp. 657-661.

[H5] Hopcroft, J., and R. Tarjan, "Efficient Planarity Testing," JACM 21, 1974, pp. 549-568.

[J1] Jones, N., Y. Lien, and W. Laaser, New Problems Complete for Nondeterministic Log Space, TR-75-1, Dept. of Computer Science, Univ. of Kansas, April 1975.

[K1] Karp, R. and W. Miranker, "Parallel Minimax Search for a Maximum," Journal of Combinatorial Theory 4, January 1968, pp. 19-35.

[K2] Kogge, P., "Parallel Solution of Recurrence Problems," IBM J. of Research and Development 18, March 1974, pp. 138-148.

[K3] Kozen, D., "On Parallelism in Turing Machines," Proc. of the 17th Annual Symposium on Foundations of Compter Science, Houston, Texas, Oct. 1976, pp. 89-97.

[K4] Kung, H., "New Algorithms and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions and Recurrences," JACM 23, April 1976, pp. 252-261.

[M1] Miller, R., "Mathematical Studies of Parallel Computation," Proc. of the First IBM Symposium on Mathematical Foundations of Computer Science, October 1976, pp. 1-23.

[M2] Munro, I. and M. Paterson, "Optimal Algorithms for Parallel Polynomial Evaluation," JCSS 7, 1973, pp. 189-198.

[P1] Peterson, J. and T. Bredt, "A Comparison of Models of [illegible]

[illegible] 221.

[P3] Preparata, F., Parallellism in Sorting, Proc. of the International Conference on Parallel Processing, Bellair, Michigan, August 1977.

[R1] Russell, R., "The Cray-1 Computer System," CACM 21, 1978, pp. 63-72.

[S1] Savage, C. and J. Ja'Ja', Fast, Efficient Parallel Algorithms for Some Graph Problems, Manuscript from Computer Science Dept., Pensylvania State University, 1979.

[S2] Savitch, W. and M. Stimson, "Time Bounded Random Access Machines with Parallel Processing," JACM 26, January 1979, pp. 103-118.

[S3] Savitch, W., Parallel and Nondeterministic Time Complexity Classes, Report 78-CS-012, Dept. of Applied Physics and Information Science, Univ. of California at San Diego.

[S4] SIGARCH Newsletter 7, April 1979, Proc. of the 6th Annual Symposium on Computer Architecture.

[S5] Stone, H., "Parallel Processing with the Perfect Shuffle," IEEE Transactions on Computers C20, February 1971, pp. 153-161.

[S6] Swan, R., S. Fuller, and D. Siewiorek, "Cm* - A modular, multi-processor," AFIPS Conference Proceedings 46, 1977, pp. 637-644.

[T1] Tarjan, R., "Depth First Search and Linear Graph Algorithms," SIAM J. of Computing 1, 1972, pp. 146-160.

[T2] Tarjan, R., "Finding Dominators in Directed Graphs," Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems, 1974, pp.414-418.

[W1] Wagner, R. and M. Fischer, "The String to String Correction Problem," <u>JACM</u> <u>21</u>, January 1974, pp. 168-173.

[W2] Waksman, "A Permutation Network," <u>JACM</u> <u>15</u>, January 1968, pp. 159-163.

[W3] Wen, K., <u>Interprocessor Connections - Capabilities, Exploitation and Effectiveness</u>, Ph.D. thesis, Report No. UIUCDCS-R-76-830, Univ. of Illinois, October 1976.