Correspondence

Partitioning Techniques for Large-Grained Parallelism

RAKESH AGRAWAL AND H. V. JAGADISH

Abstract—**We present a model for parallel processing in loosely coupled multiprocessing environments, such as a network of computer workstations, which are amenable to "large-grained" parallelism. The model takes into account the overhead involved in communicating data to and from a remote processor, and can be used to partition optimally a large class of computations. This class consists of computations that can be organized as a one-level tree, and are homogeneous and separable. The optimal partition can be determined for a given number of processors and, if required, the optimal number of processors to use can also be derived. We also present experimental results that validate our model and demonstrate its effectiveness.**

Index Terms—**Loosely coupled multiprocessors, multicomputers, optimization, parallel processing, problem partitioning, processor sharing, workstation networks.**

I. Introduction

A major trend in computing in recent times has been the creation of large networks of computer workstations. It has been speculated [17] that the number of computing cycles installed in computer workstations is an order of magnitude greater than the number installed in mainframes. However, most of these cycles are idle most of the time. There are many applications amenable to parallel processing that can profitably use these idle computing cycles by treating these networks as loosely coupled multicomputers. For this approach to become feasible, it is essential to design tools for partitioning a given problem into pieces that may be executed in parallel and provide system facilities to access the idle computing cycles. In this paper, we focus on problem partitioning. Issues related to system facilities for accessing idle computing cycles have been addressed elsewhere (see, for example, [1], [3], [10], [11], [22], [26], [29]).

Most of the work on problem partitioning has been done in the context of vector/array processors and tightly coupled multiprocessors (see, for example, [14]–[16], [18], [23], [25]). These techniques are oriented toward "fine-grained" parallelism as the creation of parallel threads of execution and communications between them are fairly inexpensive. In loosely coupled multicomputers, the cost of creation and communication among concurrent threads on different nodes is often too high for the fine-grained techniques to be useful. Some work has been reported in the literature that analyzed and partitioned specific applications for parallel processing on loosely coupled processors (see, for example, [4], [6], [7], [21], [28]). However, no systematic methodology or algorithmic procedure has been given for partitioning.

In this paper, we present a technique for *optimally* partitioning problems in "large-grained" parallel processing environments. Our model recognizes that in such environments, processors are connected with a rather slow communication medium, and explicitly takes into account the communication costs in addition to the computation costs. The optimal partition can be determined for a given number of processors and, if required, the optimal number of processors to use can also be derived.

Problems can be partitioned onto multiple processors in two ways. One could split the algorithm into multiple steps and execute one step on each processor. Such an approach, referred to as *algorithm partitioning*, works well for processors that can efficiently be connected in a pipeline. This approach could also work, albeit some cost in terms of contention, when the processors communicate over a common bus. However, partitioning an algorithm requires specific knowledge of the algorithm. Also, a pipelined system would support throughput determined by the largest step in the partition—and there may often be little that a user can do to reduce the size of this step. The other approach, referred to as *data partitioning* and the one followed in this paper, is to partition the data across the processors, but to run the same program on them. Not all problems permit data partitioning, but there are many problems in which a significant computational kernel can be computed independently for different partitions of the data, possibly preceded by some common computation, and possibly succeeded by some postprocessing and collation. For example, consider a large text file that has to be typeset. This file can be split into several pieces (as long as the divisions were made only at paragraph boundaries) and typeset separately, except that a final collation step would be required to handle paging. Examples of other problems that have this structure include separable mathematical programming problems, discrete Fourier transform, fault simulation, and design rule check for layouts.

Using this model, we have developed several parallel applications on the U* system [2]. U* is an experimental testbed consisting of nine AT&T 3B2 computers interconnected with AT&T 3BNET, which is an Ethernet compatible 10 Mbit local area network. Each processor runs a modified UNIX system kernel developed for the NEST project [1], [3], [12], which provides the capability to create transparent remote processes. The purpose of this exercise is twofold: we want to determine the feasibility and effectiveness of "large-grained" parallel processing in loosely coupled environments such as workstation networks, and we want to experimentally validate the theoretical results of our model.

The organization of the rest of the paper is as follows. In Section II, we present our parallel processing model, and derive techniques to optimally partition a given task. In Section III, we present experimental results that verify and demonstrate the effectiveness of our model using matrix multiplication as an example. Finally, in Section IV, we give a summary and some pointers for future research.

II. Model

A. Assumptions

Consider a job that some processor (henceforth called master) wishes to perform. Let this job be divisible into an *initial* phase, a *separable* phase, and a *final* phase. The initial and final phases are executed on the master alone. The separable phase can be split into several tasks executing in parallel on slave processors. Each task is independent—a task does not communicate or synchronize with any other task. During the separable phase, let each task also be divisible in three phases: an *input* phase, a *compute* phase, and an *output* phase. Let the time required for input, compute, and output phases at processor i be a_i, y_i, and b_i, respectively. The input and output times include the time taken to communicate data over the network and also the time required to read/write the data on disk. The input time also includes the time to transfer program code to the slave, if necessary, and other overheads such as the remote process creation cost. Let the time for the initial phase be I, and for the final phase be F. All of I, F, a_i, b_i, and y_i are functions of n, the number of parallel tasks that the separable phase is divided into. I and F are likely to be increasing functions of n, if not constant, while a_i, b_i, and y_i are likely to decrease with n.

Manuscript received February 16, 1988; revised July 14, 1988. A preliminary version of this paper appeared in the Proceedings of the 7th Annual International Phoenix Conference on Computers and Communications, Scottsdale, AZ, Mar. 1988.

The authors are with AT&T Bell Laboratories, Murray Hill, NJ 07974.

IEEE Log Number 8824088.

The work in the separable phase can be divided into tasks in several ways. One naive way would be to divide the work equally among the slaves. We are interested in an optimal partition so that the total execution time for the job is minimized. Find a measure of size for each task that the separable phase is split into, s_i being the size of task i, such that $\Sigma_{i=1}^{n} s_i = 1$. Since we are interested in data partitioning, the size of the input would usually be the measure of size. Then for task i, the computation time is $y(s_i)$, the input time is $a(s_i)$, and the output time is $b(s_i)$. We will require, for reasons that will become clear when we get to the Order Maintenance Theorem below, that the computation time, input time, and output time, all be monotonically nondecreasing functions of this measure of size and further satisfy the property that $f(p) - f(q) \geq f(l) - f(m)$ for any $p > l$ and $p - q \geq l - m$. Intuitively this property says that if one considers the increase of the function f over some interval $[l, m]$, then over any interval $[p, q]$ of the same (or larger) size to the right of $[l, m]$, the increase of the function f must be at least as much. The class of functions that satisfies these properties is very large, and appears to include most functions of practical interest. Specifically, it includes all nondecreasing concave functions (which computation costs usually are), all nondecreasing affine functions (which communication costs are normally modeled as), and even all constant functions.[1]

We shall assume throughout this paper that all communication over the network is point-to-point. If broadcast is available, input that is common to several slaves needs be transmitted only once by the master. This situation can be handled in our model by counting the time spent in the broadcast against the input time for the first slave and not against any of the other slaves.

We also assume that each processor can do only one thing at a time. It can either compute or it can communicate with one other processor. During communication between two processors both processors are kept busy for the entire duration. It may appear that we are ignoring multiprocessing. However, even in the multiprocessing situation, two tasks initiated in parallel will each take as much time to finish as both together would have if performed in sequence, unless one task is able to utilize CPU cycles during which the other is waiting for some external resource to become available to produce results.

B. A Lower Bound

Let $\Sigma_{i=1}^{n} a_i = A$ and $\Sigma_{i=1}^{n} b_i = B$. Usually both disk accesses and communications over the network are modeled as affine functions of the data size consisting of a constant overhead added to a proportional cost. The cost to create a remote process and to transfer the program to a slave would be constant for a program. Therefore, A and B are likely to be constant for a given n irrespective of the manner in which the work is divided and is likely to increase with an increase in n depending upon the value of constant overhead. Let $Y = \Sigma_{i=1}^{n} y_i$. Y is likely to be a function both of n and of the manner in which the work is divided into tasks.

A lower bound for the total time to finish the entire job is $I + A + B + F$. This bound is derived from the activity of the master processor. The master has at the very least to do the initial processing, communicate the input to all the slaves, receive the results from all the slaves, and do the final processing. The bound is an increasing function of n, since I and F are likely to increase with n, and is achieved only when the master is kept completely busy. Unless computation is carried out in parallel with communication, the desired lower bound cannot be achieved for any given n.

C. Optimal Partitioning

Let us first introduce a lemma (see Appendix for a proof) based on the following condition.

Input Precedence Condition: The master should send input data to all slaves before receiving output data from any slave.

[1] A function f is said to be *concave* if $f(p) - f(q) \geq f(l) - f(m)$ for any $p > l$ and $p - q \geq l - m$. A function f is said to be *affine* if it is of the form $bx + c$, where b and c are constants. Note that all affine functions are also concave. An affine function is said to be *linear* if $c = 0$, and *constant* if $b = 0$.

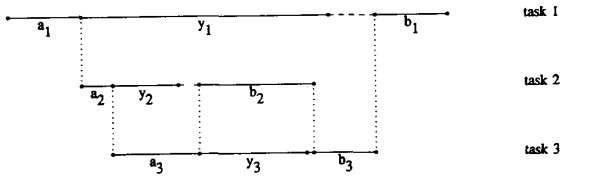

Fig. 1. An example involving three slave processors.

Lemma: Input Precedence Lemma: There exists a minimum execution time partition and schedule that satisfies the input precedence condition. (There may, in addition, exist other minimum execution time schedules that do not satisfy the input precedence condition.)

Let the separable phase commence at time 0, so that slave 1 starts reading input at time 0 and starts computing at time a_1. Slave 2 starts reading input at time a_1 and begins computing at time $a_1 + a_2$, and so on. Slave k can start computing at time $\Sigma_{i=1}^{k} a_i$. All inputs complete at time A. Let the outputs from the slaves be in some order $j_1, j_2, \cdots$ etc., not necessarily the same as the inputs. At each slave, we can write a constraint on T, the total time required for the separable phase, in terms of all the inputs that must occur before the current slave can begin computing, the computation at this slave itself, and all the outputs that must occur from this and later slaves at the end:

$$T \geq \sum_{i=1}^{k} a(s_i) + y(s_k) + \sum_{i=j_k}^{n} b(s_i). \tag{1}$$

An Example: Consider an example with three slaves shown in Fig. 1. Slaves 1, 2, and 3 receive their inputs in order and commence computation, but the output order is 2, 3, 1. Thus, $j_1 = 2$, $j_2 = 3$, and $j_3 = 1$. The constraints can be written as

$$T \geq a_1 + y_1 + b_1$$

$$T \geq a_1 + a_2 + y_2 + b_3 + b_2 + b_1$$

$$T \geq a_1 + a_2 + a_3 + y_3 + b_3 + b_1.$$

We now introduce a theorem that makes one further assumption: that the separable phase can be split into tasks of any arbitrary size, with no restrictions in terms of granularity of the division. Thus, the number of tasks that the separable phase is split into is equal to the number of slaves used in the processing. A proof of correctness is given in the Appendix.

Theorem: Order Maintenance Theorem: Let n be the number of tasks to be executed in parallel on different slaves. For any n, the minimum execution time is achieved for some partition in which the slaves output to the master in the same order as the slaves receive input from the master.

Because of the Order Maintenance Theorem, $j_k = k$ for each $k = 1, \cdots, n$. The constraints on the time T obtained in (1) simplify to

$$T \geq \sum_{i=1}^{k} a(s_i) + y(s_k) + \sum_{i=k}^{n} b(s_i) \tag{2a}$$

for all $k = 1, \cdots, n$, where n is the number of slaves. In addition, T must be greater than the lower bound on time to execute the separable phase, $A + B$:

$$T \geq \sum_{i=1}^{n} a(s_i) + \sum_{i=1}^{n} b(s_i). \tag{2b}$$

Finally, a constraint can be written to normalize the measure of size:

$$\sum_{i=1}^{n} s_i = 1. \tag{2c}$$

So far we assumed that no computation was performed by the master, an assumption that is true in the case that the lower bound is achieved and the master is kept communication bound. However, there usually are constraints on the number of processors available. If there are not enough processors to achieve the lower bound, we have to do the best we can with the number available. In the extra time now available, since the bound is not being met, the master can also perform some computation. Let the size of the task computed at the master be s_0. Then we can replace (2b) and (2c), respectively, with

$$T \geq \sum_{i=1}^{n} a(s_i) + y(s_0) + \sum_{i=1}^{n} b(s_i) \tag{3b}$$

$$\sum_{i=0}^{n} s_i = 1. \tag{3c}$$

Combining these constraints with the n constraints from (2a), we can write the following optimization problem:

$$\min T \quad \text{subject to}$$

$$T \geq \sum_{i=1}^{k} a(s_i) + y(s_k) + \sum_{i=k}^{n} b(s_i)$$

$$\text{for } k = 1, \cdots, \text{n} \quad (n \text{ constraints})$$

$$T \geq \sum_{i=1}^{n} a(s_i) + y(s_0) + \sum_{i=1}^{n} b(s_i)$$

$$\sum_{i=0}^{n} s_i = 1. \tag{4}$$

Such a problem can be solved using well-known techniques (cf. [20]). If $a(\cdot)$, $b(\cdot)$, and $y(\cdot)$ are linear or affine, then (4) becomes a standard linear programming problem, which we can solve following techniques such as in [9].

D. On Choosing n

Our formulation in (4) does not prescribe a value of n directly. However, for a given value of n, the $(n + 1)$th constraint concerns the communication at the master. If this constraint is satisfied exactly by the solution obtained, then the desired lower bound is achieved, while if there is slack, it is not. Moreover, a solution to (4) will also provide the actual value of the best time T that can be achieved for a chosen n.

As discussed before, the lower bound is an increasing function of n. Therefore, one need not consider values of n larger than some value for which the lower bound is achieved. The optimum value of n can be determined by beginning with a small value of n and trying successively larger integers until the lower bound on time is achieved.

A bound can easily be derived on the smallest value of n for which the algorithm executes in time $I + A + B + F$, the lower bound on total execution time derived at the master. The entire computation of the separable part, requiring a total time of Y if performed on a single processor, has to be performed while the master is spending time $A + B$ in communicating data with the slaves. Assuming that the computation is evenly divided among n processors, that each of them can commence computation at the same time that the master starts providing the first slave with input, and that the total computation time added up over the slaves after partitioning is still just Y, one would still need to allow at least Y/n time for computation at the slaves. Therefore, we can require that

$$n \geq \frac{Y}{A+B}. \tag{5}$$

This equation provides a starting point for the values of n that should be tried in the search for the optimal partition. To the extent that A, B, and Y are themselves functions of n, it may not be straightforward to solve the above equation explicitly. However, one only has to obtain a reasonable estimate from this equation to get a starting point.

Note that the lower bound on completion time is generally an increasing function of n. Therefore, a better completion time may sometimes be found for some smaller value of n for which the lower bound cannot be attained. The formulation in (4) can be used to obtain the best time for each n between the limits discussed in this section. The smallest of these times dictates the n of choice, and the corresponding optimal partition.

E. A Brief Resume of the Partitioning Technique

Given a distributed environment with a master–slave model of computation, and an algorithm whose data can arbitrarily be divided, assigned to slaves, and processed with no communication between tasks (except in a final collation at the master), one uses the following procedure to obtain an optimal partitioning:

1. Select a measure of task size, and determine $a(\cdot)$, $b(\cdot)$, and $y(\cdot)$. The first two can usually be obtained directly from known network communication speeds, and an accounting of the information that has to be transferred as a function of task size. $y(\cdot)$ is algorithm dependent, and has to be determined afresh for each new algorithm. But it can be measured on a single processor, by assigning it tasks of several different sizes.
2. Obtain a minimum n from (5).
3. For each value of n beginning from the one obtained from the step 2 and increasing by one each time, set up and minimize T subject to (4). Continue as long as the s_0, the size of the task executed at the master, is nonzero in the partition.
4. The value of n, out of those attempted in step 3, for which T is minimized is the optimum number of processors to have. The solution to (4) for that n gives the optimum partition.

Note that this procedure need be run once for each distributed system and algorithm. Once the solution has been computed, it can be cached, and used directly whenever the same algorithm executes at a later time on the same system.

III. Experimental Verification

In this section, we report on our experience with problem partitioning using the model presented in Section II.

A. Testbed

Experiments were performed on the U* system [2], which is an experimental multicomputer testbed consisting of nine AT&T 3B2/300 computers interconnected with AT&T 3BNET, an Ethernet compatible 10 Mbit local area network. Each processor runs a modified UNIX kernel developed for the NEST project [1], [3], [12]. The basic software building block is the "rexec" primitive, which allows a process to be executed on a remote machine while logically appearing to execute on the originating machine.

Communication between processes on different machines is over one-way communication paths, similar to ones proposed in the DEMOS [5] and the Roscoe [27] projects. In a communication paradigm based on one-way communication paths, the communication path is created by the receiver, who then grants the prospective sender the capability to send messages on this path. Broadcast is not available as a primitive at the application level in distributed systems based on this communication paradigm, even if the communication hardware supports broadcast (see, for example, [8], [13], [24]).

B. A Preliminary Experiment

A major assumption in our model is "one communication at a time;" that is, we assume that only one slave could be communicating with the master at a time. In order to assess the impact of this assumption, we performed a preliminary experiment. One master and eight slaves were involved in this experiment. Each slave was to copy different 0.5 Mbyte of data from the master in chunks of 128 kbyte blocks. The time required for eight sequential remote reads was

TABLE I
PARALLEL REMOTE COPYING

stagger (seconds)	total read time (seconds)
0	205.2
15	203.7
16	190.5
17	181.5
18	164.6
20	164.3

155.4. Table I shows the time required for eight parallel reads. A positive stagger represents the time lag between initiating two successive reads. When the stagger was zero, all the eight reads were initiated simultaneously.

Eight concurrent reads (stagger = 0) perform much worse than the eight serial reads, and only when stagger nearly serializes the parallel reads, do they approach the performance of serial reads. The explanation as to why unsynchronized communication takes longer than sequential communication lies in the fact that in a local area network, in the absence of specialized communication processors, most of the time in sending a message is spent in CPU processing rather than on the network itself [19], and it is more efficient to process two tasks in serial rather than in parallel on the same CPU. This experiment justifies our "one at a time" assumption from the performance standpoint.

C. Problem Selection

We initially partitioned a compute-intensive application that required almost no communication. The problem involved determining the Ramanujam number (smallest N such that $N = a^3 + b^3$, $N = c^3 + d^3$, and $a \neq b \neq c \neq d$, the answer being 1729 with $a = 1$, $b = 12$, $c = 9$, and $d = 10$). The sequential execution of a straightforward (rather dumb) five-nested loop algorithm that exhaustively searched the number space starting from 1 took 53361.9 s on an AT&T 3B2/300 computer. When the same algorithm was executed on U* with eight processors searching the vertically partitioned (modulo 8) number space in parallel, the execution time was 6855.4, which is almost eightfold the speedup. Although the speedups are perfect, we did not pursue partitioning of such problems as they appear to be rather easy and hence uninteresting. Instead, we decided to concentrate on problems that require a significant amount of both computation and communication.

The problem we chose for our study was matrix multiplication. This problem has some nice properties. It is a homogeneous separable problem whose computation time is relatively insensitive to actual data values, and fits our framework well. In our parallel implementation, each slave reads selected rows of one matrix and all of the second matrix,[2] computes part of the result matrix, and writes it back to the master. The results reported in this paper are for the multiplication of two 100 × 100 square integer matrices.

D. Naive Approach

A naive approach to parallel processing would be to create the desired number of slave processes (as many as the number of available processors), equally divide the input data, and let the slaves work on the problem without any synchronization for accessing data from the master. The experimental results of this naive parallel matrix multiplication have been summarized in Fig. 2 and Table IV.

E. Semi-Naive Approach

Drawing upon the results of the preliminary experiment reported in Section III-B, which suggested that one-at-a-time serial reads from the master perform better than parallel reads, one would expect an improvement in the above naive approach if data accesses were serialized. The input data are still divided equally among the slaves, but through appropriate signaling, only one slave reads or writes data at a time. The experimental results have been summarized in Fig. 2

[2] One can possibly use a different unit of partition, such as a square submatrix rather than a set of rows. The choice of unit partition is problem dependent, and has not been addressed in this paper.

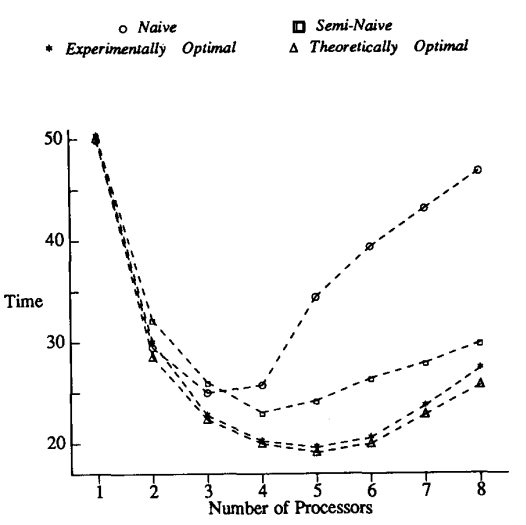

Fig. 2. Execution times.

TABLE II
ESTIMATES of a, b, AND y AS FUNCTIONS OF s. (ALL UNITS IN SECONDS)

Parameter	Estimate
Input time	$a = 1.21 + 1.05\ s$
Output time	$b = 0.10 + 1.59\ s$
Computation time	$y = 44.52\ s$

and Table IV. The time required to complete the multiplication is now significantly less than the naive approach, especially as the number of processors goes up. Also, using this approach, more processors can be profitably employed. For a small number of slave processors, the naive approach performs marginally better as it does not incur any cost for synchronization.

F. Optimal Approach

As in the semi-naive approach, in the optimal approach, the data flow between the master and the slaves is synchronized. However, instead of partitioning the data equally, the model presented in Section II is used to determine data distribution among slave processors.

1) Determination of Parameters: We modeled the communication costs and computation costs as affine functions in the size of input data. A number of experiments (not reported here) were performed to validate this assumption and estimate the values of the coefficients and intercepts. We emphasize that our model can handle more complex functions for the computation and communication costs, but we chose them to be affine since intuitive explanations suggested that these functions should be affine for our problem and, when we actually estimated the coefficients, the fit was found to be very good. Table II summarizes the values of a, b, and y as functions of s, which is taken to be the number of rows of the matrix. Table III shows the optimal values of the partitions for different numbers of processors, obtained by solving the corresponding linear program.

2) Execution Time: Fig. 2 and Table IV show the theoretically optimal time required for matrix multiplication for different numbers of processors using the optimal partition given in Table III. The time required decreases until five processors are used. For more than five processors, the computation becomes communication bound and the time required starts increasing. Although not shown in Fig. 2, this trend continues as more processors are used.

Fig. 2 and Table IV also show the experimentally determined execution times. The execution times obtained using optimal task sizes are consistently better than the times obtained using either the naive or semi-naive approach. There is an excellent match between the theoretically predicted results and the experimentally determined results. (The theoretical estimates are somewhat lower which is partly

TABLE III
OPTIMAL PARTITION

Number of Processors	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
1	1.0000							
2	0.5262	0.4738						
3	0.3878	0.3335	0.2787					
4	0.3329	0.2781	0.2226	0.1664				
5	0.3116	0.2564	0.2007	0.1442	0.0871			
6	0.2725	0.2725	0.2169	0.1607	0.0701	0.0073		
7	0.1829	0.1829	0.1829	0.1829	0.1625	0.1056	0.0003	
8	0.1658	0.1658	0.1658	0.1658	0.1658	0.1658	0.0036	0.0016

TABLE IV
EXECUTION TIMES

Processors	Execution Time (seconds)			
	Naive	Semi-Naive	Optimal	
	Experimental	Experimental	Theoretical	Experimental
1	49.91	50.32	50.06	50.32
2	29.52	32.23	28.55	29.98
3	24.98	26.01	22.34	22.75
4	25.75	23.05	19.94	20.22
5	34.39	24.18	19.07	19.55
6	39.36	26.41	19.92	20.52
7	43.12	27.92	22.80	23.67
8	46.76	29.88	25.68	27.34

explained by the initial startup cost and partly by the rounding errors as each slave multiplies an integral number of rows.) We were also able to accurately predict the optimal number of processors.

IV. Summary and Future Work

With the advent of a large number of computer networks and the opportunity they provide for large-grained parallelism, the need for a formal and rigorous treatment of the partitioning problem has acquired considerable importance. We have presented a model for parallel processing in a loosely coupled multiprocessing environment that takes into account the overhead involved in communicating data to and from a remote processor, and derived techniques to optimally partition a given task. The optimal partition can be determined for a given number of processors and, if required, the optimal number of processors to use can also be derived. We have focused on data partitioning rather than algorithm partitioning (the same instruction stream executes different parts of data). Our model can be used to partition computations that can be organized as a one-level tree, and are homogeneous and separable. We have also presented experimental results that verify and demonstrate the effectiveness of our model.

The model presented in this paper can be extended in many ways. First of all, the model assumes that the cost functions are deterministic. This assumption is quite reasonable for numeric computations. However, for nonnumeric computations (such as sorting), the computation times are often data dependent. The model can be extended to allow for variances in the cost functions. The model also assumes that the input data can be arbitrarily partitioned. In general, this assumption is not true and input can only be partitioned at specific partitioning points (for example, at a record boundary in the case of sorting). If the number of partitioning points is much greater than the number of partitions (as in the case of multiplication of large matrices), the perturbation would not make significant difference. If the number of possible partitioning points is not large, but the partitioning points are equally spaced, integer programming can be used to determine partitions instead of linear programming. The case of a small number of partitioning points that are not equally spaced (for example, functions of different sizes in a parallel compilation) may also be handled by using extensions of integer programming. Finally, the model can be extended to work for computations that are organized as multilevel trees.

Appendix

Order Maintenance Theorem

Assumptions:

1) The job is divisible into an *initial* phase, a *separable* phase, and a *final* phase. The separable phase can be split into several tasks that execute in parallel. Each task is independent—a task does not communicate or synchronize with other tasks.

2) Each task is also divisible into an *input* phase, a *compute* phase, and an *output* phase. A task needs no further input once the compute phase begins, and the output phase begins only after the compute phase has finished.

3) There is some measure of size for each task that the separable phase is split into, s_i being the size of piece i, such that $\Sigma_{i=1}^{n} s_i = 1$. The computation time, input time, and output time, all are monotonically nondecreasing functions of this measure of size and further satisfy the property $f(p) - f(q) \geq f(l) - f(m)$ that for any $p > l$ and $p - q \geq l - m$. We will call such functions *interesting*.

4) The separable phase can be split into tasks of any arbitrary size, with no restrictions in terms of granularity of the division.

5) Each processor can do only one thing at a time. It can either compute or it can communicate with exactly one other processor. During communication between two processors, both processors are kept busy for the entire duration.

Input Precedence Condition: The master should send input data to all slaves before receiving output data from any slave.

Lemma: Input Precedence Lemma: There exists a minimum execution time partition and schedule that satisfies the input precedence condition. (There may, in addition, exist other minimum execution time schedules that do not satisfy the input precedence condition.)

Proof: Given an arbitrary order of input to the slaves and output from the slaves, rearrange the order such that input is given to all slaves prior to obtaining output from any slave, but without altering the relative order in which any pair of slaves receive input or any pair of slaves transmit output. Such a reordering is always possible with no additional delays introduced since inputs are being made available earlier and outputs are being required later so that no slave processor has less time available to complete its computation with the reordering. Therefore, the total time required after reordering is no more than that required before.

Now suppose that there exists a minimum execution time partition and schedule that does not satisfy the necessary input precedence condition, then it can have a reordering performed to satisfy the input precedence condition, such as the one described in the previous paragraph, with no increase in execution time. Therefore, there exists a minimum execution time schedule and partition that satisfies the input precedence condition. As such, we need only consider schedules that satisfy the input precedence condition in our search for the minimum execution time partition. □

Theorem: Order Maintenance Theorem: Let n be the number of tasks to be executed in parallel on different slaves. For any n, the minimum execution time is achieved for some partition in which the slaves output to the master in the same order as the slaves receive input from the master.

Proof: Consider a pair of tasks, i and $i + 1$, that are initiated consecutively but whose outputs are taken in reverse order. We wish to show that these tasks can be reorganized such that the outputs are in the same order as the inputs, and that this reorganization does not adversely affect the execution of the other tasks being performed in parallel.

Define points on the time line, t_1 at which task i is initiated, t_2 and t_3 at which the outputs of the two tasks commence, respectively, and t_4 at which this pair of tasks is complete (see Fig. 3). Task i begins computation and task $i + 1$ commences input at time $t_1 + a(s_i)$. Task

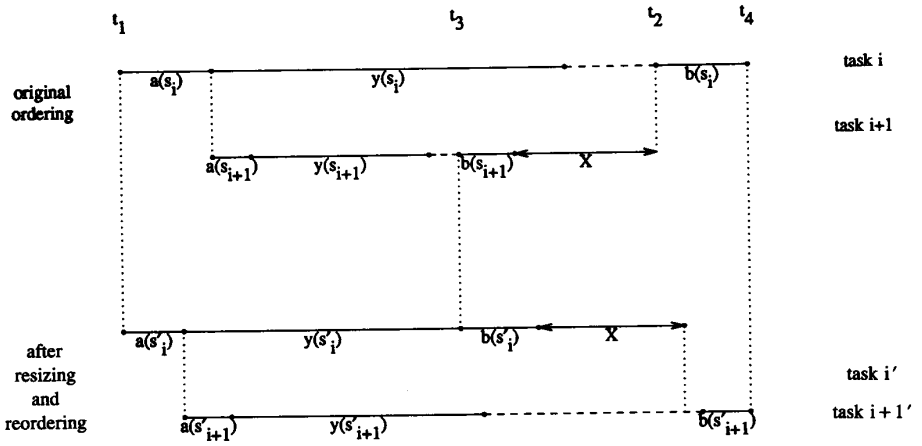

Fig. 3. Task size adjustment and output reordering.

i completes computation at time $t_1 + a(s_i) + y(s_i)$, and task $i + 1$ at $t_1 + a(s_i) + a(s_{i+1}) + y(s_{i+1})$. The times for the outputs can be related as

$$t_4 = t_2 + b(s_i)$$
$$t_2 = t_3 + b(s_{i+1}) + X \quad \text{(A.1)}$$

where X is the time for any intervening outputs and slack. These two equations can be combined to give

$$t_4 = t_3 + b(s_i) + b(s_{i+1}) + X. \quad \text{(A.2)}$$

Also, the outputs cannot be produced before the inputs have been provided and enough time allowed for computation. For each processor a constraint can be written:

$$t_3 \geq t_1 + a(s_i) + a(s_{i+1}) + y(s_{i+1}) \quad \text{(A.3)}$$
$$t_2 \geq t_1 + a(s_i) + y(s_i). \quad \text{(A.4)}$$

Let us reverse the order of the outputs to conform with the input order. Two cases arise. In the first case, the computation of task i completes before t_3. That is

$$t_3 \geq t_1 + a(s_i) + y(s_i).$$

The output of task i can commence at t_3 and complete at $t_3 + b(s_i)$. The output of task $i + 1$ can now commence. (Since task $i + 1$ was ready to output at t_3, we know that the output is available.) This output completes at time $t_3 + b(s_i) + b(s_{i+1})$. Any intervening outputs (in the original order) can now commence. (There can be no inputs during this time because of the input precedence lemma.) That these intervening outputs can commence is guaranteed since they would have commenced at time $t_3 + b(s_{i+1})$ in the original order. The total time taken is no more than the time for the original order.

The second and more difficult case occurs when the computation of task i cannot be completed before time t_3 (see Fig. 3). We reduce the size of task i to s_i' which satisfies

$$t_3 = t_1 + a(s_i') + y(s_i'). \quad \text{(A.5)}$$

The remainder of the task i is added on to task $i + 1$ causing it to grow in size to s_{i+1}', where

$$s_i - s_i' = s_{i+1}' - s_{i+1}. \quad \text{(A.6)}$$

From (A.3) and (A.5), one can obtain

$$a(s_i') + y(s_i') \geq a(s_i) + a(s_{i+1}) + y(s_{i+1}) \geq a(s_{i+1}) + y(s_{i+1}).$$

Since $a(\cdot)$ and $y(\cdot)$ are both monotonically nondecreasing functions of their arguments, so is the sum of the two functions. Therefore, we have

$$s_i' \geq s_{i+1}. \quad \text{(A.7)}$$

Combining this information with (A.6), and using the definition of *interesting* functions, we can write

$$f(s_i) - f(s_i') = f(s_{i+1}') - f(s_{i+1}) \quad \text{(A.8)}$$

where $f(\cdot)$ is any *interesting* function, such as $a(\cdot)$, $b(\cdot)$, or $y(\cdot)$.

Now, one can make the following derivation starting from a rearrangement of (A.5):

$$\begin{aligned} y(s_i') &= t_3 - t_1 - a(s_i') \\ &\geq t_3 - t_1 - a(s_i) && \text{since } s_i \geq s_i' \\ &\geq a(s_{i+1}) + y(s_{i+1}) && \text{from (A.3).} \end{aligned} \quad \text{(A.9)}$$

Combining (A.1), (A.4), and (A.5), we can write

$$b(s_{i+1}) + X \geq (a(s_i) + y(s_i)) - (a(s_i') + y(s_i'))$$
$$\geq (a(s_{i+1}') + y(s_{i+1}')) - (a(s_{i+1}) + y(s_{i+1}))$$

by applying (A.8) twice. From (A.7), since $b(\cdot)$ is a monotonically nondecreasing function, we can rewrite

$$b(s_i') + X \geq (a(s_{i+1}') + y(s_{i+1}')) - (a(s_{i+1}) + y(s_{i+1})).$$

Adding $t_1 + a(s_i')$ to both sides of this equation, and also adding (A.9) to both sides, we obtain

$$t_1 + a(s_i') + b(s_i') + y(s_i') + X \geq t_1 + a(s_i') + a(s_{i+1}') + y(s_{i+1}'). \quad \text{(A.10)}$$

The left-hand side of this equation is the time at which the output of task $i + 1$ is scheduled to begin. The right-hand side of the equation represents the time at which the computation of task $i + 1$ completes. The inequality assures us that in spite of the size enhancement, the new task $i + 1$ will have completed computation in time for it to output the results.

The intervening time X between the two outputs of tasks i and $i + 1$ may consist of some other task outputs. (There can be no inputs during this time because of the input precedence lemma.) Since the

time slot X has been retained intact, sufficient time has been allowed for all of these outputs to complete even in the new schedule. Furthermore, this time slot now begins at time $t_3 + b(s_i')$ rather than at $t_3 + b(s_{i+1})$ as in the original schedule. Since $b(s_i') \geq b(s_{i+1})$, as we just saw, we are guaranteed that this time slot is now scheduled no earlier than it was in the original schedule. Therefore, any computations that could have completed in time to make use of this time slot for output can still do so.

Thus far, we have shown that we can reverse the order of two outputs that are out of sequence and still obtain a valid schedule. We now have to show that this new schedule completes no later than the original schedule. Rearranging the terms in (A.8) with $b(\cdot)$ as the *interesting* function in question, one can write

$$b(s_i) + b(s_{i+1}) \geq b(s_i') + b(s_{i+1}').$$

Adding (A.2) to this equation,

$$t_4 \geq t_3 + b(s_i') + b(s_{i+1}') + X. \tag{A.11}$$

In other words, the revised schedule completes no later than the original completion time t_4.

We have just shown that by switching the order of the outputs (and adjusting the sizes of the pieces if necessary) it is possible to complete both tasks no later than they were being completed previously. The only restriction placed on the tasks was that their inputs be consecutive. Through such pairwise flips, one has to arrange the order of the outputs to correspond with the order of the inputs. The pairwise flips come in two flavors. In one, the two output positions are interchanged with no effect on intervening outputs (case II above). In the other (case I above), the second output takes the place of the first output while the first output and all intervening outputs move down one position. The following simple algorithm suffices.

Look at the task that is output first in the given schedule. Let this be task k. The output of task $k - 1$ definitely succeeds that of task k. Do a flip between this pair. Now task $k - 1$ is output first. Flip this task with $k - 2$. Continue such pairwise flips until task 1 is the one that is output first. Since we only look at the task that is positioned earliest after the flip, it is immaterial which flavor of the flip is performed.

Now look at the task that is output second. Perform a similar set of pairwise flips until task 2 is in that position. This second set of flips is guaranteed not to affect task 1. Continue performing such sets of flips on the third, fourth, etc., positions until all the outputs are in order. □

A Counterexample

The Order Maintenance Theorem may, at first glance, appear "obvious" to the reader. The counterexample below shows that it does not hold in general.

Consider the cost functions ($0 \leq s_i \leq 1$):

$$a(s_i) = b(s_i) = 10 \times s_i$$

$$y(s_i) = \begin{cases} 10 \times s_i & 0 \leq s_i < 0.2 \\ -10 + 60 \times s_i & 0.2 \leq s_i < 0.5 \\ 10 + 20 \times s_i & 0.5 \leq s_i \leq 1. \end{cases}$$

Note that $y(\cdot)$ does not satisfy our definition of an *interesting* function given with Assumption 3 in this Appendix. (For example, if $p = 0.6$, $q = 0.5$, $l = 0.5$, and $m = 0.4$, so that $p > l$ and $p - q \geq l - m$, but $y(p) - y(q) < y(l) - y(m)$.) Let us assume that there are two slave processors. If the output ordering constraint dictated by the order maintenance theorem is observed, the optimal execution would be obtained when the input is partitioned equally, and the time required for the separable phase would be 35 time units as shown in Fig. 4. However, if the output ordering is not required to be maintained, it is possible to do better as also shown in Fig. 4, where only 34 time units are required for the alternative sequence of execution.

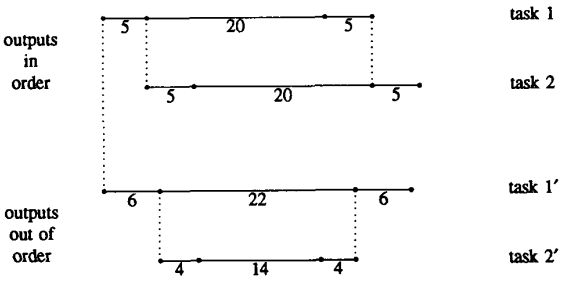

Fig. 4. Order maintenance is not always good.

ACKNOWLEDGMENT

A. Alston developed the software for matrix multiplication on U* and performed an initial set of experiments.

REFERENCES

[1] R. Agrawal and A. K. Ezzat, "Processor sharing in NEST: A network of computer workstations," in *Proc. IEEE 1st Int.Conf. Computer Workstations*, San Jose, CA, Nov. 1985, pp. 198–208.

[2] P. Agrawal and R. Agrawal, "Software implementation of a recursive fault tolerance algorithm on a network of computers," in *Proc. 13th Int. Symp. Comput. Architecture*, Tokyo, Japan, June 1986, pp. 65–72.

[3] R. Agrawal and A. K. Ezzat, "Location independent remote execution in NEST," *IEEE Trans. Software Eng.*, vol. 13, pp. 905–912, Aug. 1987.

[4] E. H. Baalbergen, "Parallel and distributed compilations in loosely-coupled systems: A case study," presented at IEEE Workshop Large Grained Parallelism, Providence, RI, Oct. 1986.

[5] F. Baskett, J. H. Howard, and J. T. Montague, "Task communication in DEMOS," in *Proc. ACM-SIGOPS 6th Symp. Oper. Syst. Principles*, Nov. 1977, pp. 23–31.

[6] M. Beck, D. Bitton, and W. K. Wilkinson, "Sorting large files on a backend multiprocessor," in *Proc. 1986 Int. Conf. Parallel Processing*, St. Charles, IL, Aug. 1986.

[7] H. J. Boehm and W. Zwaenepoel, "Parallel attribute evaluation on a local network," presented at IEEE Workshop Large Grained Parallelism, Providence, RI, Oct. 1986.

[8] W. A. Burnette, R. H. Canaday, and D. H. Fishman, "MAX: A distributed system," Tech. Memo. AT&T Bell Lab., Murray Hill, NJ, 1983.

[9] G. B. Dantzig, *Linear Programming and Extensions*. Princeton, NJ: Princeton Univ. Press, 1963.

[10] D. J. DeWitt, R. Finkel, and M. Solomon, "The CRYSTAL multicomputer: Design and implementation experience," *IEEE Trans. Software Eng.*, vol. 13, Aug. 1987.

[11] F. Douglis and J. Ousterhout, "Process migration in the Sprite operating system," in *Proc. 7th Int. Conf. Distributed Comput. Syst.*, Berlin, West Germany, Sept. 1987, pp. 18–25.

[12] A. K. Ezzat and R. Agrawal, "Making oneself known in a distributed world," in *Proc. 1985 Int. Conf. Parallel Processing*, St. Charles, IL, Aug. 1985, pp. 139–142.

[13] D. H. Fishman, M. Y. Lai, and W. K. Wilkinson, "Overview of the Jasmin database machine," in *Proc. 1984 ACM SIGMOD Int. Conf. Management Data*, Boston, MA, June 1984, pp. 234–239.

[14] E. J. Gilbert, "Algorithm partitioning tools for a high-performance multiprocessor," Ph.D. dissertation, STAN-CS-83-946, Comput. Sci. Dep., Stanford Univ., Stanford, CA, Feb. 1983.

[15] H. V. Jagadish, "Techniques for the design of parallel and pipelined VLSI systems for numerical computations," Ph.D. dissertation, Inform. Syst. Lab., Stanford Univ., 1985.

[16] H. V. Jagadish, S. K. Rao, and T. Kailath, "Multiprocessor architectures for iterative algorithms," *Proc. IEEE*, Aug. 1987.

[17] L. Kleinrock, "Distributed systems," *IEEE Computer*, vol. 18, pp. 90–103, Nov. 1985.

[18] D. Kuck, R. Kuhn, B. Leasure, and M. Wolfe, "The structure of an advanced retargettable vectorizer," in *Proc. COMPSAC '80*, 1980.

[19] T. J. Leblanc, "The design and performance of high-level language primitives for distributed programming," Ph.D. dissertation, Comput. Sci. Dep., Univ. Wisconsin, Madison, 1982.

[20] O. L. Mangasarian, *Non-Linear Programming*. New York: McGraw-Hill, 1969.

[21] T. A. Marsland, M. Olafsson, and J. Schaeffer, "Multiprocessor tree-search experiments," in *Advances in Computer Chess 4*, D. Beal, Ed. Oxford, England: Pergamon, 1985, pp. 37–51.
[22] D. A. Nichols, "Using idle workstations in a shared computing environment," in *Proc. ACM-SIGOPS 11th Symp. Oper. Syst. Principles*, Nov. 1987, pp. 5–12.
[23] S. K. Rao, "Regular iterative algorithms and their implementation on processor arrays," Ph.D. dissertation, Inform. Syst. Lab., Stanford Univ., 1985.
[24] G. R. Sager, J. A. Melber, and K. T. Fong, "The Oryx/Pecos operating system," *AT&T Tech. J.*, vol. 64, pp. 251–268, Jan. 1985.
[25] S. Sahni, "Scheduling multipipeline and multiprocessor computers," *IEEE Trans. Comput.*, vol. C-33, pp. 637–645, July 1984.
[26] J. F. Shoch and J. A. Hupp, "The "Worm" Programs—Early experience with a distributed computation," *Commun. ACM*, vol. 25, pp. 172–180, Mar. 1982.
[27] M. H. Solomon and R. A. Finkel, "The Roscoe distributed operating system," in *Proc. ACM-SIGOPS 7th Symp. Oper. Syst. Principles*, Dec. 1979, pp. 108–114.
[28] M. Stumm and D. Cheriton, "Distributed parallel computations under V," presented at *IEEE Workshop Large Grained Parallelism*, Providence, RI, Oct. 1986.
[29] M. Theimer, K. Lantz, and D. Cheriton, "Preemptable remote execution facilities for the V-System," in *Proc. ACM-SIGOPS 10th Symp. Oper. Syst. Principles*, Orcas Island, WA, Dec. 1985, pp. 2–12.

Circuit Simulation on Shared-Memory Multiprocessors

P. SADAYAPPAN AND V. VISVANATHAN

Abstract—**This paper reports the parallelization on a shared-memory vector multiprocessor, of the computationally intensive components of a circuit simulator—matrix assembly (including device model evaluation), and unstructured sparse linear system solution. A theoretical model is used for predicting the performance of the lock-synchronized parallel matrix assembly and compared to experimental measurements. Alternate approaches to efficient sparse matrix solution are contrasted, highlighting the impact of the matrix representation/access strategy on achievable performance and a new medium-grained approach with superior realized performance is reported. The techniques developed have been incorporated into a prototype parallel implementation of the production circuit simulator ADVICE on the Alliant FX/8 multiprocessor.**

Index Terms—**Multiprocessors, parallel circuit simulation, parallel sparse factorization, sparse matrix, supercomputers.**

I. Introduction

Circuit simulators such as SPICE2 [17], ASTAP [26], and ADVICE [18] are widely used in VLSI design to accurately model and predict the electrical behavior of circuits before fabrication. The size of circuits that can currently be simulated at this detailed level is limited to a few thousand devices, whereas a complete VLSI chip routinely contains hundreds of thousands of devices. Thus, orders of magnitude of performance improvement will be required before full-chip circuit simulation of even small designs is to become feasible. The recent commerical availability of parallel computers holds promise for speeding up circuit simulation. However, the effective use of these machines is not necessarily a simple matter; the problem of identifying and exploiting the inherent parallelism in circuit simulation must be solved. This problem is made difficult by the fact that the key feature of a circuit simulator—a strongly coupled set of equations is solved at each time point—that gives it its robustness is also the one that makes its parallelization difficult. Nevertheless, a number of researchers have begun the development of parallel circuit simulators [4], [5], [14], [27].

There are three basic types of analyses that are carried out in a circuit simulator: ac, dc, and transient analysis, of which transient analysis is the most time consuming. Hence, the main focus of this work is the standard transient analysis algorithm in the simulator ADVICE [18] which is in production use at AT&T. In this algorithm, an implicit integration scheme is used to discretize the differential equations that describe the transient behavior of a circuit. This results in a set of nonlinear algebraic equations that have to be solved to determine the voltages and currents at each time point. The solution technique used is a modified Newton–Raphson method. This is an iterative scheme which consists of repeated assembly and solution of a nonsymmetric sparse system of linear equations. We refer to the former step as the *LOAD* operation and to the latter as *SOLVE*. Since these two steps consume a very large fraction of the CPU time for transient analysis, we will concentrate on techniques for parallelizing them. These techniques have been implemented in a prototype parallel version of ADVICE on the Alliant FX/8 multiprocessor.

The paper is organized as follows. We begin in Section II by outlining a framework for evaluating the performance of our parallel implementations. In Section III, we briefly describe the salient computational aspects of the transient analysis algorithm, and its primary components *LOAD* and *SOLVE* that are the focus of our parallelization effort. In Section IV, we discuss techniques for exploiting concurrency in the *LOAD* operation. We first develop a theoretical prediction of the performance of our approach and then compare it to experimental measurements on circuits of various sizes. In Section V we deal with the *SOLVE* step, for which the issues that have to be addressed for efficient parallelization are very different from that for *LOAD*. In sparse-matrix solution, the matrix representation scheme and element access strategy have a significant impact on the performance of the code. We therefore develop a medium-grained parallel solver that exploits both concurrency and vectorization and is efficient with respect to element access. We demonstrate its superiority to alternate approaches on some representative circuit matrices. The framework introduced in Section II is used to provide insight into the superior performance of our approach. We end the section by briefly describing some interesting further ideas on parallel *SOLVE* that we are currently implementing. Finally, in Section VI we provide a summary of our work.

II. A Framework for Performance Evaluation

The parallel implementation of an algorithm on a computer with N processors will in the ideal case provide a speedup of N over a sequential implementation. In practice, this is rarely achievable, due to a number of factors. These factors can be broadly classified as causes of processor idling or execution overheads. The main causes of processor idling are load imbalance and resource contention. Execution overheads include the overhead that results from modifications to the representation of the algorithm (with respect to that used for efficient sequential execution) that may be necessary to facilitate exploitation of parallelism. As will be seen later, this representation-modification overhead has little effect on the performance of parallel *LOAD* but has a nontrivial influence on the parallelization of *SOLVE*. In addition, the cost of scheduling and synchronizing concurrent tasks (called concurrency-operations overhead) and of setting up vector operations are also execution overheads.

In order to analyze the effect of the above-mentioned factors on the observed performance, we find it convenient to introduce three

Manuscript received March 5, 1988; revised July 19, 1988.
P. Sadayappan is with the Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210.
V. Visvanathan is with AT&T Bell Laboratories, Murray Hill, NJ 07974.
IEEE Log Number 8824089.