PARTITIONING TECHNIQUES FOR LARGE-GRAINED PARALLELISM

RAKESH AGRAWAL and *H.V. JAGADISH*

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

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 optimally partition 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.

1. INTRODUCTION

A major trend in computing in recent times has been the creation of large networks of computer workstations. It has been speculated 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 [13]. 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 the problem into pieces that may be executed in parallel.

Most of the work in the past on problem partitioning has been done in the context of vector/array processors and tightly-coupled multiprocessors (see, for example, [10-12, 14, 16, 19]). These techniques are oriented toward "fine-grained" parallelism as the creation of parallel threads of execution and communications between them are fairly inexpensive. Unfortunately, in the loosely-coupled multicomputers context, the costs of creating and communicating among concurrent threads on different nodes are often too high for the fine-grained techniques to be useful. Some work has been reported in literature that analyzed and partitioned specific applications for parallel processing on loosely-coupled processors (see, for example, [5-7, 15, 20]). However, no systematic methodology or algorithmic procedure has been given for partitioning.

In this paper, we present a technique for optimally partitioning problems in the "large-grained" parallel processing environments. Our model recognizes that in such environments processors are connected with rather a 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 communication 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 the 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 algorithms permit data-partitioning, but it appears that there are large classes of 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 post processing and collation. For example, a large dataset can be sorted by splitting it into many small datasets, sorting them individually on separate processors, and then merging the results. For another example, consider a large text file that has to be type-set. This file can be split into several pieces (as long as the divisions were made only at paragraph boundaries) and type-set separately, except that a final collation step would be required to handle paging. Similarly, most matrix operations involve independent computation of each matrix element in the result and hence can easily be split across many processors although considerable overlap of input data is likely. 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 megabit local area network. Each processor runs a modified UNIX system kernel developed for the NEST project [1,4,9] which provides capability to create transparent remote processes. The purpose of this exercise is two fold: 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 2, we present our model of parallel processing and derive techniques to optimally partition a given task. In Section 3, we present experimental results that verify and demonstrate the effectiveness of our model using matrix multiplication as an example. Finally, in Section 4, we give a summary and some pointers for future research. The results are stated in this paper without proof. The proofs, and a more detailed presentation, are contained in [3].

2. MODEL

2.1 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, 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.

The work can be divided into pieces in several ways. One naive way would be to divide the work equally amongst the slaves. We are interested in the optimal partitioning 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 piece i, such that $\sum_{i=1}^{n} s_i = 1$. Since we are interested in data partitioning, the size of input data would usually be the measure of size. Then for piece i, the computation time is $y(s_i)$, input time is $a(s_i)$, and output time is $b(s_i)$. We will require that the computation time, input time and output time, all be monotonically non-decreasing 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 $p - q \geq l - m$. The class of functions that satisfies these properties is very large, and appears to include most functions of practical interest. Specifically, it includes all concave (*cf.* [18]) non-decreasing functions (which computation costs usually are), all non-decreasing affine functions (which communication costs are normally modeled as), and even all constant functions.

Let $\sum_{i=1}^{n} a_i = A$ and $\sum_{i=1}^{n} b_i = B$. 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 $Y = \sum_{i=1}^{n} y_i$. Y is likely to be a function both of n and of the manner in which the work is divided into pieces.

Finally, let us assume that 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. It may appear that multiprocessing has been assumed away. However, even in the multiprocessing situation, a simplifying assumption can be that two tasks occurring in parallel each take as much time to finish as both together would have if performed in sequence.

2.2 A Lower Bound

Under these assumptions, 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, and is achieved only when the master is kept completely busy.

It is easy to see that unless computation is carried out in parallel with communication, the desired lower bound cannot be achieved. Before proceeding further, we introduce the following lemma:

Lemma: Input Precedence Lemma

> The minimum execution time is achieved for some partition with all the inputs fed before any outputs are taken.

Consider now an implementation in which if the separable phase commenced at time 0, 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 computing at time $a_1 + a_2$, and so on. Slave k can start computing at time $\sum_{i=1}^{k} a_i$. All inputs complete at time A. Now the outputs from the slaves are in some order $j_1, j_2, \cdots$ *etc.* If the first output is to begin at time A, as soon as all the inputs are done, then slave j_1 must have finished computation by this time, imposing a constraint:

$$y_{j_1} \le A - \sum_{i=1}^{j_1} a_i$$

This output takes time b_{j_1}, by which time, slave j_2 should have finished computation, giving the constraint:

$$y_{j_2} \le A + b_{j_1} - \sum_{i=1}^{j_2} a_i$$

Proceeding in a similar fashion one obtains the set of constraints:

$$y_{j_k} \le A + \sum_{i=1}^{k-1} b_{j_i} - \sum_{i=1}^{j_k} a_i$$

Remembering that $A = \sum_{i=1}^{n} a_i$, these constraints can be rewritten:

$$y_{j_k} \le \sum_{i=1}^{k-1} b_{j_i} + \sum_{i=j_k+1}^{n} a_i \qquad \text{for all } 1 \le k \le n \qquad (1$$

The set of equations (1) represent the set of constraints that must be satisfied if the lower bound on the execution time is to be met. The set of equations (1) can also be written in matrix form, using $\vec{y}$ to represent the vector of compute times, $\vec{a}$ of input times, and $\vec{b}$ of output times. The ordering is captured in the permutation matrices **I** and **J**. The matrix inequality constraint then is:

$$\mathbf{J}\vec{y} \le \mathbf{L}\mathbf{J}\vec{b} + \mathbf{U}\mathbf{I}\vec{a} \qquad (2$$

where **L** is a strictly lower triangular matrix of all 1's and **U** is a strictly upper triangular matrix of all 1's.

2.3 Optimal Partitioning

Before seeking the optimal number of processors and the corresponding partitioning of the task, we need a theorem that is presented now. This theorem 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.

Theorem: Order Maintenance Theorem

> For any choice of n, the number of tasks to be executed in parallel, the minimum execution time is achieved for some partition with the outputs taken in the same order as the inputs were fed.

Because of the Order Maintenance Theorem, we know that $\mathbf{J} = \mathbf{I}$ in eqn. (2), the matrix inequality that we wish to satisfy. Moreover, by numbering the tasks correctly, these permutation matrices can be set to identity. Thus eqn. (2) can be rewritten as:

$$\vec{y} \le \mathbf{L}\vec{b} + \mathbf{U}\vec{a} \qquad (3$$

which can be rearranged to obtain:

$$(b\mathbf{L} - diag(y,\ y,\ .\ .\ .\ ,\ y) + a\mathbf{U})\vec{s} \ge 0 \qquad (4$$

where a, b, y should be considered operators rather than scalars, and $diag(y,\ y,\ .\ .\ .\ ,\ y)$ is a matrix with each entry on the diagonal being y and all others being zero. If the input, output, and computation costs are all linear functions, then these operators are indeed scalars and eqn. (3) is a standard matrix vector inequality. It is formulated as a linear programming problem to which we seek a non-negative non-zero feasible solution, $\vec{s}$. Such a solution is easily found, if one exists, by well-known techniques [8]. This solution can then be scaled to satisfy the normalization requirement.

One is usually interested not just in finding any feasible solution for a particular n, but rather one that minimizes the total time, T, taken by the separable phase to execute. In the spirit of eqn. (1), we can write the following set of constraints on T:

$$T \geq \sum_{i=1}^{k} a(s_i) + y(s_i) + \sum_{i=k}^{n} b(s_i) \qquad \text{(5a}$$

for all $k = 1, \ldots, n$, where n is the number of pieces the task is split into. In addition, of course, T must be greater than the lower bound:

$$T \geq \sum_{i=1}^{n} a(s_i) + \sum_{i=1}^{n} b(s_i) \qquad \text{(5b}$$

These constraints can then be condensed into a single matrix inequality, with the first n rows of the inequality corresponding to the n constraints from eqn. (5a), the $n+1$ row corresponding to (5b), and the last $(n+2)$ row arising from the normality constraint on our measure of size:

$$\begin{bmatrix} 1 & -(a+b+y) & -b & & -b \\ 1 & -a & -(a+b+y) & & -b \\ 1 & -a & -a & & . \\ . & . & . & \cdots & . \\ . & . & . & & -b \\ 1 & -a & -a & & -(a+b+y) \\ 1 & -(a+b) & -(a+b) & & -(a+b) \\ 0 & 1 & 1 & & 1 \end{bmatrix} \begin{bmatrix} T \\ s_1 \\ s_2 \\ . \\ . \\ s_n \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \\ . \\ . \\ 0 \\ 1 \end{bmatrix} \qquad \text{(6}$$

where a, b, y should once again be considered operators. If a(.), b(.), y(.) are indeed linear functions, then eqn. (6) becomes a standard linear programming problem which we can solve with the objective function of minimizing T.

If a, b, y are affine rather than linear functions, so that $a(s_i) = a_0 + as_i$ and so forth, linear programming can still be used. Eqn. (3) now becomes:

$$\begin{bmatrix} 1 & -(a_1+b_1+y_1) & -b_1 & & -b_1 \\ 1 & -a_1 & -(a_1+b_1+y_1) & & -b_1 \\ 1 & -a_1 & -a_1 & & . \\ . & . & . & \cdots & . \\ . & . & . & & -b_1 \\ 1 & -a_1 & -a_1 & & -(a_1+b_1+y_1) \\ 1 & -(a_1+b_1) & -(a_1+b_1) & & -(a_1+b_1) \\ 0 & 1 & 1 & & 1 \end{bmatrix} \begin{bmatrix} T \\ s_1 \\ s_2 \\ . \\ . \\ s_n \end{bmatrix} \geq \begin{bmatrix} a_0+y_0+nb_0 \\ 2a_0+y_0+(n-1)b_0 \\ . \\ . \\ . \\ na_0+y_0+b_0 \\ n(a_0+b_0) \\ 1 \end{bmatrix} \qquad \text{(7}$$

where $\vec{y_0}$ is a constant vector with each term equal to y_0, and similarly $\vec{a_0}$ and $\vec{b_0}$ are also constant vectors. Thus the right hand side now has a known constant vector rather than zero, but the standard techniques of linear programming still apply.

The only quantity that we may not have prespecified is n, the number of pieces. Our formulation in eqns. (3) and (4) does not prescribe a value of n directly, but rather uses it in determining the dimensions of the matrices and vectors to be used. Fortunately, there is a simple way around this problem, and that is to try successively large values until a solution is found for the linear programming problem thus set up.

2.4 A Special Case

A case of special interest occurs when the input and output times are identical for any task of the same size, (though they may be different for different sizes of tasks). That is, $a(.) = b(.)$. Moreover, let $a(.)$ and $b(.)$ be affine. Then row k of inequality (3) becomes:

$$\sum_{i=1}^{n} a(s_i) - (a(s_k) + y(s_k)) \geq 0$$

For affine $a(.)$ the summation on the left-hand side is a constant for a given n independent of the piece sizes. Call it A. Then we can write:

$$y(s_k) + a(s_k) \leq A$$

Since this inequality has to be satisfied for every k, for monotonically non-decreasing functions a(.) and y(.), we wish to minimize $\max_{k=1}^{n} s_k$. The solution to this "minimax" is to have all pieces equal, resulting in each $s_k = \frac{1}{n}$. In other words, equal piece sizes are indeed the best we can do in this special case.

However, it should be noted that usually the input time is considerably greater than the output time (not just because the quantity of data input is greater, but also because the "input" time also includes the time to transfer program and to start up a a remote process). Therefore, one should normally expect $a(.)$ to be larger than $b(.)$.

3. EXPERIMENTAL VERIFICATION

In this section, we report on our experience with parallelization using the partitioning model presented in Section 2.

3.1 Test Bed

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 megabit local area network. Each processor runs a modified UNIX kernel developed for the NEST project [1,4,9]. 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.

The location independence of processes is realized by requiring that a remote process carry with it the view of the file system that exists on the originating machine. For example, if a file "/a/b" exists on two machines robin and cuckoo, and a remote process initiated at robin and executing at cuckoo references "/a/b", by default, the file made available to the process is the "/a/b" that exists on robin, and not the one that exists on cuckoo. This capability is important as we assume in our model that the input data required by the slaves resides on the master, and the final results are also deposited at the master. It is also possible for a remote process to selectively access files that exist on the machine on which the process is executing. This capability was used for creating, for example, temporary files. Furthermore, remote execution on U* also preserves pipes, parent-child relationships and signals across machine-boundaries. This feature allows the multiprocessing applications to be easily developed within the UNIX system's *fork-exec* framework [17] by simply changing *execs* to *rexecs*.

3.2 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 0.5 megabyte of data from the master in chunks of 128 kilobyte blocks. Time required for eight sequential remote reads was 155.4 seconds. Table 3.1 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.

Table 3.1. 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

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. This experiment justifies our "one at a time" assumption from the performance stand point.

3.3 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 seconds on a 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 seconds, which is almost eight times speed-up. Although the speed-ups 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 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

Table 3.3. Optimal Partition

Number of Processors	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
1	1.000000							
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

matrix and all of the second matrix, 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.

3.4 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 results of this naive parallel matrix multiplication have been summarized in Figure 3.1 and Table 3.4.

3.5 Semi-Naive Approach

Drawing upon the results of the preliminary experiment reported in Section 3.2, 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 is still divided equally amongst the slaves, but through appropriate signaling, only one slave reads or writes data at a time. The results have been summarized in Figure 3.1 and Table 3.4. The time required to complete the multiplication are significantly less than in 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.

3.6 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 2 is used to determine data distribution amongst slave processors.

3.6.1 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 3.2 summarizes the values of a, b, and y as functions of s, which is taken to be the number of columns of the matrix. Table 3.3 shows the optimal values of the partitions for different number of processors, obtained by solving the corresponding linear program.

Table 3.2. Estimates of a, b, and y as functions of s (All units in seconds)

parameter	coefficient	intercept
a	1.05	1.21
b	1.59	0.10
y	44.52	0

3.6.2 Execution Time

Figure 3.1 and Table 3.4 show the theoretically optimal time required for matrix multiplication for different number of processors using the optimal partition given in Table 3.3. The time required decreases till five processors are used. For more than five processors, the computation becomes communication bound and the time required starts increasing. Although not shown in Figure 3.1, this trend continues as more processors are used.

Figure 3.1 and Table 3.4 also show the experimentally determined execution times. The execution times obtained using optimal piece 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 explained by the initial start-up 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.

Table 3.4. Execution Times

Proce-	Execution Time (seconds)			
ssors	Naive	Semi-Naive	Optimal	
			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

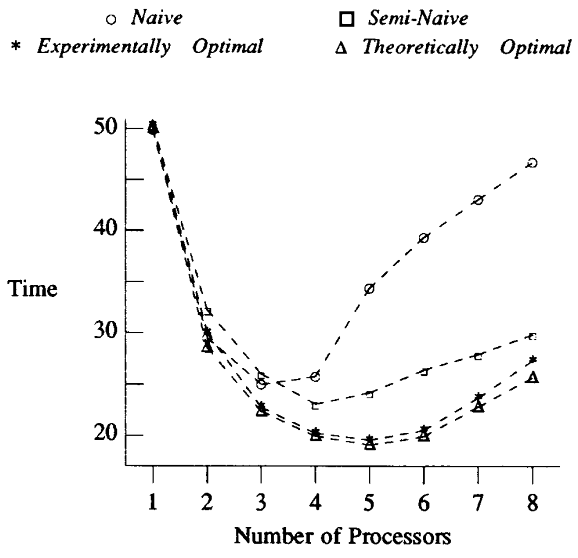

Figure 3.1. Execution Times

4. SUMMARY AND FUTURE WORK

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 focussed on data partitioning rather than algorithm partitioning (the same instruction stream executes different part of data). Our model can be used to partition computations that can be organized as 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 is an initial step toward attacking the difficult problem of parallel processing on loosely-coupled processors. This model can be extended in many ways. First of all, our model assumes that the cost functions are deterministic. This assumption is quite reasonable for numeric computations. However, for non-numeric computations (such as sorting), the computation times are often data dependent. The model should be extended to allow for variances in the cost functions. Our 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 partioning 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 are 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, our model can be extended to work for computations that are organized as multi-level trees.

5. ACKNOWLEDGEMENTS

Alson 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, *Proc. IEEE 1st Int'l Conf. Computer Workstations*, San Jose, California, Nov. 1985, 198-208.

[2] P. Agrawal and R. Agrawal, Software Implementation of a Recursive Fault Tolerance Algorithm on a Network of Computers, *Proc. 13th Int'l Symp. Computer Architecture*, Tokyo, Japan, June 1986, 65-72.

[3] R. Agrawal and H. V. Jagadish, Parallel Computation on Loosely-Coupled Workstations, Technical Memorandum, AT&T Bell Laboratories, Murray Hill, New Jersey, 1986.

[4] R. Agrawal and A. K. Ezzat, Location Independent Remote Execution in NEST, *IEEE Trans. Software Eng. 13*, 8 (Aug. 1987), 905-912.

[5] E. H. Baalbergen, Parallel and Distributed Compilations in Loosely-Coupled Systems: A Case Study, *IEEE Workshop Large Grained Parallelism*, Providence, Rhode Island, Oct. 1986.

[6] M. Beck, D. Bitton and W. K. Wilkinson, Sorting Large Files on a Backend Multiprocessor, *Proc. 1986 Int'l Conf. Parallel Processing*, St. Charles, Illinois, Aug. 1986.

[7] H. J. Boehm and W. Zwaenepoel, Parallel Attribute Evaluation on a Local Network, *IEEE Workshop Large Grained Parallelism*, Providence, Rhode Island, Oct. 1986.

[8] G. B. Dantzig, *Linear Programming and Extensions*, Princeton Univ. Press, Princeton, New Jersey, 1963.

[9] A. K. Ezzat and R. Agrawal, Making Oneself Known in a Distributed World, *Proc. 1985 Int'l Conf. Parallel Processing*, St. Charles, Illinois, Aug. 1985, 139-142.

[10] E. J. Gilbert, Algorithm Partitioning Tools for a High-performance Multiprocessor, STAN-CS-83-946, Computer Science Dept., Stanford Univ., Stanford, California, Feb. 1983. Ph.D. Dissertation.

[11] H. V. Jagadish, Techniques for the Design of Parallel and Pipelined VLSI Systems for Numerical Computations, Information Systems Lab., Stanford U., 1985. Ph.D. Dissertation.

[12] H. V. Jagadish, S. K. Rao and T. Kailath, Multiprocessor Architectures for Iterative Algorithms, *IEEE Proceedings*, Aug. 1987.

[13] L. Kleinrock, Distributed Systems, *IEEE Computer 18*, 11 (Nov. 1985), 90-103.

[14] D. Kuck, R. Kuhn, B. Leasure and M. Wolfe, The Structure of an Advanced Retargettable Vectorizer, *Proc. COMPSAC '80*, 1980.

[15] T. A. Marsland, M. Olafsson and J. Schaeffer, Multiprocessor Tree-Search Experiments, in *Advances in Computer Chess 4*, D. Beal (ed.), Pergamon Press, Oxford, 1985, 37-51.

[16] S. K. Rao, Regular Iterative Algorithms and their Implementation on Processor Arrays, Information Systems Lab., Stanford U., 1985. Ph.D. Dissertation.

[17] D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, *Commun. ACM 17*, 7 (July 1974), 365-375.

[18] R. T. Rockafeller, *Convex Analysis*, Princeton Univ. Press, Princeton, New Jersey, 1970.

[19] S. Sahni, Scheduling Multipipeline and Multiprocessor Computers, *IEEE Trans. Computers C-33*, 7 (July 1984), 637-645.

[20] M. Stumm and D. Cheriton, Distributed Parallel Computations under V, *IEEE Workshop Large Grained Parallelism*, Providence, Rhode Island, Oct. 1986.