Network Partitioning of Data Parallel Computations

Jon B. Weissman and Andrew S. Grimshaw

Department of Computer Science
University of Virginia

Abstract

Partitioning data parallel computations across a network of heterogeneous workstations is a difficult problem for the user. We have developed a runtime partitioning method for choosing the number and type of processors to apply to a data parallel computation, and a decomposition of the data domain in order to achieve reduced completion time. The partitioning method utilizes information about the problem in the form of callback functions and uses a set of topology-specific communication functions to estimate communication costs. We show that the method makes effective partitioning decisions and has runtime overhead that is easily tolerated. In particular, we show that for two implementations of a canonical stencil application, minimum elapsed times are obtained for a range of problem sizes on a network of heterogeneous workstations[1].

1.0 Introduction

The increasing availability of network computing resources, including high-speed networks and high performance machines, presents an opportunity for delivering good performance on a range of parallel applications. Developing applications to run efficiently in such an environment however can be extremely difficult. We have seen the advent of software tools that help enable parallel programming on a heterogeneous network [2][10]. Unfortunately, most of these tools are limited in one way or another: the programmer is still often responsible for problem partitioning, data domain decomposition, processor selection, and task placement.

We have developed a runtime partitioning method for choosing the number and type of processors to apply to a data parallel computation, and a decomposition of the data domain in order to achieve reduced completion time. The problem of automatically choosing the number of processors is quite difficult and has received little attention in the literature.

The partitioning method utilizes information about the computation and communication structure of the implementation provided in the form of *callback* functions. The method also relies upon a set of *topology-specific* communication functions that have been constructed off-line. These communication functions and callbacks allow an estimate of the computation granularity to be computed at runtime based on the available processing resources. This estimate is used to determine the number and type of processors that are best applied to the computation. The partitioning method also computes a static decomposition of the data domain that gives processor load balance. We believe that the partitioning method is applicable to a large class of data parallel computations.

The partitioning method is based on a heterogeneous network model which is hierarchical in structure. This organization is more realistic than a flat organization. The method is also based on a model for data parallel computations. Both the network model and data parallel computation model are discussed. We then present the partitioning method. The results show that the method makes effective partitioning decisions and has runtime overhead that is easily tolerated. In particular, we show that for two implementations of a stencil application, minimum elapsed times are obtained for a range of problem sizes on a network of heterogeneous workstations.

2.0 Related work

A number of research efforts in the area of partitioning data parallel computations on a heterogeneous network have emerged [1][9]. The dataparallel C runtime system [9] is targeted to partitioning regular data parallel computations using a dynamic load balancing strategy to handle processor heterogeneity. One advantage of this approach is that load imbalance due to processor sharing can also be handled. This approach is limited to regular data parallel computations and only addresses the data decomposition

1. This work has been partially funded by grants NSF ASC-9201822, JPL-959303, and NASA NGT-50970.

problem. It is also assumed that the problem is of sufficient size to utilize all processors and amortize the cost of dynamic load balancing. Reeves et al [1] propose a strategy for partitioning data parallel computation based on benchmarking. Their approach is limited to specific data parallel operations such as reductions and a set of possible processor configurations. Our approach is not limited to a set of data parallel operations or processor configurations, but only to a common set of communication topologies.

3.0 Heterogeneous network

The heterogeneous network contains a number of physical network segments connected by one or more routers. The essential property of a network segment is that it has *private* bandwidth. We partition the processors on the network into groups called *clusters*. A *cluster* contains a homogeneous group of processors. We make a number of assumptions about this organization:

- Segments have equal communication bandwidth
- Each segment contains a single cluster
- Every pair of segments is connected by a single router

The first assumption requires that the network segments have the same communication capabilities, e.g., all segments are ethernet-connected or FDDI-connected. The second assumption restricts each network segment to be *homogenous* since a single cluster will contains processors of the same type. The first assumption simplifies the partitioning algorithm (Section 5.0) and the second and third assumptions simplify the communication cost functions presented later in this section. The third assumption means that messages will travel one hop at most. These assumptions are valid for most cluster computing environments such as Sandia HEAT. An example of a network containing three clusters (Sun4's, HP's, and RS-6000's) on three ethernet-connected network segments joined by a router is shown in Fig. 1.

Each cluster contains a processor that is responsible for managing the resources of the cluster for scheduling. This processor is known as the *cluster manager* (shaded nodes in Fig. 1). Each cluster manager stores the following information:

- *bandwidth* (bits/sec)
- *processor nodes* (total, available)
- *instruction speed* (integer, floating point)

We assume that the processors on the network are *shared*. The cluster manager monitors the load status of its processors and uses a simple threshold policy to determine if a processor is *available*. We treat all processors below this threshold as available and equal in terms of computation power. The threshold can be made sufficiently small to support this assumption. This assumption simplifies the partitioning algorithm, but is not a necessary requirement. In the general case, we will allow all processors to be available but with the associated instruction speed adjusted to reflect current load. The instruction speed refers to the speed of the processor type contained in the cluster.

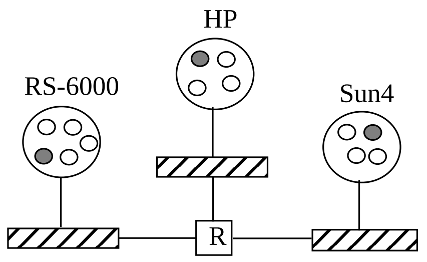

Figure 1. Network Organization

Communication between all machines on the network is enabled by a reliable heterogeneous message-passing system (MMPS) based on UDP datagrams [5]. Partitioning the data parallel computation requires that an accurate estimate of the communication costs using MMPS be known. Consider the simple case where all communication occurs within a cluster C_i. The communication cost function for C_i depends on three *application-dependent* parameters: (1) the size of messages exchanged, (2) the number of communicating processors, and (3) the application communication topology. In the model we present in the next section, these parameters will be known at runtime and the number of communication topologies will be limited to a restricted set.

The latter assumption allows a set of very accurate message cost functions to be constructed for each cluster type by benchmarking a set of *topology-specific* communication programs. Although we have assumed that clusters have equal communication capacity, the cost functions for different clusters may be different due to processor speed differences. For example, we would expect that communication is faster on a cluster of Sun4's than on a cluster of Sun3's. The topologies we consider include a common set of regular patterns such as *1-D*, *tree*, *broadcast* and *ring*. These communication topologies are *synchronous* in that all processors participate in the communication at the same logical time. The synchronous nature of the communication means that the communication cost experienced by all processors is roughly the same and is determined by the processor experiencing the greatest cost. This observation has been verified by empirical data.

These cost functions determine the average communication cost, measured as elapsed time, incurred by a processor during a single communication *cycle*. During a communication cycle, each processor does an asynchronous send to each neighboring processor followed by a blocking receive from each neighboring processor. For example in a *1-D* topology, each processor sends to its north and south neighbors and then receives from its north and south neighbors. For each cluster C_i and communication topology τ, we have a communication cost function of the form:

$$T_{comm}\,[C_i,\,\tau]\,(b,\,p)$$

The cost function is parameterized by p, the number of processors within the cluster, and b, the number of bytes per message. For example suppose C_1 refers to the RS-6000 cluster in Fig. 1. The cost function $T_{comm}\,[C_1,\,1\text{-}D]\,(b,\,p)$ refers to the average cost of sending and receiving a b byte message in a *1-D* communication topology of p processors within the RS-6000 cluster computed as elapsed time. The parameter p is one way to capture contention effects within the cluster since the offered load is linear in p on ethernet. The cost functions have a latency term that depends on p and a bandwidth term that depends on both p and b (c_1 and c_2 are latency constants and c_3 and c_4 are bandwidth constants):

$$T_{comm}\,[C_i,\,\tau]\,(b,\,p) = c_1 + c_2\,p + b(c_3 + c_4\,p) \quad \text{(EQ 1)}$$

Each communication function is benchmarked using different p and b values to derive the appropriate constants. A set of communicating tasks are mapped over the processors to perform the benchmarking. The placement of tasks depends on the communication topology and several strategies are presented in [11].

For communication that crosses cluster boundaries, there are two complications that make cost estimation more difficult. First, if communicating processors in different clusters support different data formats, then a per message coercion cost must be included. Second, communication between clusters must pass through a router and a per message router cost must be included. We extend the benchmarking strategy to measure these costs:

$$T_{router}\,[C_i,\,C_j]\,(b)$$
$$T_{coerce}\,[C_i,\,C_j]\,(b)$$

Suppose now that processors spanning clusters C_i and C_j are communicating in a *1-D* topology. A processor in C_i that communicates with a processor in C_j will incur a greater communication cost and vice-versa. The empirical evidence indicates that the router may be treated as an additional station that contends for the ethernet channel plus internal router delay. Both coercion cost and router delay is a per byte penalty. Thus, the communication cost for a processor within C_i or C_j that communicates across the router will be (shown for a processor in C_i):

$$T_{comm}\,[C_i,\,\tau]\,(b,\,p{+}1) + T_{router}\,[C_i,C_j]\,(b) + T_{coerce}\,[C_i,\,C_j]\,(b)$$

Since T_{router} and T_{coerce} are linear in the message size b, this equation has the same form as Eq. 1 with the constant c_3 reflecting coercion and router costs. For communication topologies that are not bandwidth-limited (i.e., broadcast), the actual communication cost, which we denote by $T_{comm}\,[\tau]$, experienced by all processors is the maximum cost over all clusters:

$$T_{comm}\,[\tau] = max\,\{T_{comm}\,[C_i,\,\tau]\} \quad \text{(EQ 2)}$$

For bandwidth-limited topologies such as broadcast, the available bandwidth is linear in the *total* number of processors. The cost functions should be viewed as average case due to the large amount of non-determinism inherent in UDP-based communications. For example, large messages and many communicating processors will increase the likelihood of packet retransmissions, but in the average case the cost functions presented are fairly accurate.

4.0 Data parallel computations

We have adopted an SPMD model of computation in which a set of identical tasks are instantiated across some number of processors with a single task placed on each processor [3][4]. Each task computes on a separate region of the data domain. The data domain is decomposed into a number of *primitive data units* (*PDU*s), where the *PDU* is the smallest unit of data decomposition. The *PDU* is problem and implementation specific. For example, the *PDU* might be a row, column, or block of a matrix in a matrix-based problem, or a collection of particles in a particle simulation. The *PDU* is similar to the *virtual processor*, but is more general since the *PDU* may arise from unstructured data domains unlike the *virtual processor*. The partitioning algorithm does not depend on the nature of the *PDU* but rather manipulates *PDU*s in the abstract.

We assume that the task implementation has been provided (either by the user or as a result of a compilation process). We discuss some properties of this task implementation later in this section. The partitioning algorithm requires that information about the computation and communication structure of the implementation be provided. In particular, the data parallel computation may be viewed as a sequence of alternating computation and communication *phases* [8]. A communication phase may include either sending or receiving messages or both. A computation phase contains only computation. If communication and computation are overlapped, it is common to have a communication phase consisting of asynchronous sends, followed by a computation phase, followed by a communication phase consisting of blocking receives. Most data parallel computations are iterative in nature with the computation and communication phases repeating after some number of phases or *cycles*.

In our model, *annotations* associated with the computation and communication phases must be provided. The annotations distill critical information about the implementation that is needed by the partitioning algorithm and may be provided by the user or a compiler. In this paper, we are concerned with how the annotations are used and not the mechanisms for specifying them. We present a method for specifying the annotations based on *callback functions* [6] that is discussed shortly.

Computation phase annotations

Each computation phase is annotated as follows:

- *number of PDUs* (num_PDUs)
- *computational complexity*

The number of *PDU*s depends on problem parameters (e.g., problem size) and the nature of the *PDU*. The amount of computation done on a *PDU* in a single cycle is known as the c*omputational complexity*. The computational complexity is the number of instructions executed per *PDU*. These instructions may also include memory operations.

Communication phase annotations

Each communication phase is annotated as follows:

- *topology*
- *communication complexity*
- *overlap* (computation phase)

The topologies include regular synchronous patterns such as *1-D*, *2-D*, *tree*, *ring*, and *broadcast*. The amount of communication between tasks is known as the *communication complexity*. The communication complexity is the number of bytes transmitted by a task in a single communication during a single cycle of the communication phase. It is assumed that each task initiates a single communication to each neighboring task during a single cycle of the communication phase. If this communication phase is overlapped with a computation phase, the name of the computation phase is provided by the *overlap* annotation. Among the computation and communication phases, two phases are distinguished. The *dominant* computation phase has the largest computational complexity, while the *dominant* communication phase has the largest communication complexity.

Partitioning determines the number of *PDU*s to be assigned to each task (i.e., processor). This information is contained in a structure known as the *partition vector* (*A*) that is defined as follows:

A_i = number of *PDU*s assigned to processor p_i

$\sum A_i$ = *num_PDU*s

The implementation is responsible for using the *partition vector* in a manner appropriate to the implementation. For example, suppose the *partition vector* contained a *1-D* partition of a 20x20 matrix across four processors ($p_1..p_4$), as in Fig. 2. In this example, the implementation takes this abstract partition and decomposes the data domain into the appropriate number of *PDU*s (i.e., rows) to be distributed to each of the four processors.

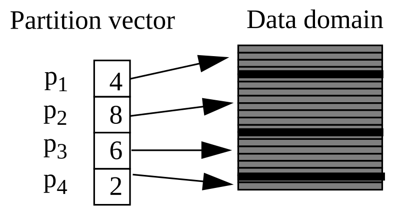

Figure 2. Partitioning the data domain

An example illustrating the annotations for a particular implementation of a dense *N*x*N* five-point stencil computation is given below. The *N*x*N* five-point stencil computation has been implemented using a block-row decomposition of the grid as shown in Fig. 2. In this implementation, the *PDU* is a single row and the processors are arranged in a *1-D* communication topology. The stencil computation is iterative and consists of two *dominant* phases: a *1-D* communication to exchange north and south borders, and a simple computation phase that computes each grid point to be the average of its neighbors. The communication and computation phases are annotated as follows (assume 4 byte grid points and no computation/communication overlap):

topology = *1-D*
communication complexity = 4*N* (bytes)
num_PDUs = *N*
computational complexity = 5*N* (fp ops)

The annotations are implemented by a set of *callback* functions that will be invoked at runtime. Notice that these functions may depend on problem parameters such as the problem size (e.g., *N*). The callbacks associated with the *dominant* phases are used by the partitioning algorithm. In particular, the callbacks associated with the computational and communication complexity allow an estimate of the computation granularity to be computed at runtime. This estimate is needed in the computation of the *partition vector* and to determine the number of processors to use. The topology is used to select the appropriate communication function (Section 3.0) that is needed to estimate communication costs in the partitioning algorithm.

5.0 Partitioning

Before partitioning can be done, the available processors N_i within each cluster C_i have to be known. A cooperative algorithm is run by each cluster manager that determines the available processors. The details of this algorithm may be found in [11]. For small problems or very large networks, it is likely that all available processors will not be needed. It is assumed that once the available processors have been determined, load fluctuation due to other users is small. Under this assumption, no dynamic load balancing will be needed.

The objective of partitioning is to (1) choose the number of processors P_i ($0 \leq P_i \leq N_i$) within each C_i to use and (2) decompose the data domain to determine the A_i associated with each P_i (i.e., the *partition vector*) in order to achieve reduced completion time. Completion time is determined by two factors: load balance and computation granularity. Computation granularity restricts the amount of parallelism that can be efficiently exploited. This is captured by (1). Load balance ensures that all processors will finish at the same time, a necessary condition for reduced completion time. For synchronous communication topologies, the communication costs for all processors is the same, hence decomposing the data domain to achieve load balance is fairly straightforward [6] and is presented later.

We are developing a class of algorithms for partitioning that solves (1). Once (1) is known, (2) follows easily. The general partitioning problem can be formulated as an optimization problem in which completion time is minimized as an objective function *f* is minimized. We present this objective function later in this section and show how it is easily constructed using program information.

A heuristic for solving the general partitioning problem is based on three observations: (1) *communication locality* is important since router (and coercion) costs can increase communication overhead (2) additional communication bandwidth can be exploited by utilizing clusters on different network segments and (3) processor power is important since the use of faster processors will reduce completion time. Notice that observations (1) and (2) are in conflict. While router costs are high, these costs may be tolerable if the gain in communication bandwidth is sufficiently great. For example, a *1-D* topology has a potential for exploiting additional communication bandwidth due to greater communication locality. On the other hand, a *broadcast* topology is inherently bandwidth limited.

The general partitioning problem requires that a system of nonlinear equations be solved for P_i. We are currently exploring heuristics for the general case. What makes the general problem difficult is that the relationship between (1) and (2) can be subtle. In this paper, we present a heuristic for solving a simpler partitioning problem that is biased toward (1). The basis for our partitioning algorithm is that it is possible to order processors and clusters under a set of assumptions. Since we have assumed equal communication capacity and processor homogeneity within each cluster (Section 3.0), we order clusters based on the instruction rate of the processor type. Clusters are considered in this order with more powerful clusters chosen first. In order to maintain communication locality, all available processors within a cluster are considered before choosing processors in another cluster. This heuristic considers the use of faster processors and communication locality as more important than additional communication bandwidth.

The heuristic algorithm we propose explores a series of *processor configurations*. A processor configuration is a set of values P_i for each C_i, i.e., a fixed set of processors. If S_i is the instruction rate for cluster C_i, then for a fixed number of processors P_i, we can compute the partition vector A_i by requiring that the processors are load balanced (*m* is the number of clusters):

$$A_i = \left(\sum_{j=1}^{m} \frac{S_i}{S_j \cdot P_j} \right) \cdot NumPDUs \qquad \text{(EQ 3)}$$

This form for A_i assumes that the computational complexity is linear in the number of *PDU*s. While this is often the case, a more general form for A_i that does not have this restriction is discussed in [6]. Solving for the best P_i requires that an objective function *f* be constructed. We define the objective function *f* as an estimate of the processor elapsed time per cycle which we denote by T_c. If the processors are load balanced and the computation is *synchronous* then the total elapsed time $T_{elapsed}$ is defined as follows (*I* is the number of cycles):

$$T_{elapsed} = I * T_c + T_{startup}$$

The startup overhead includes any startup costs such as initial data distribution. In this paper, we assume that the computation is of sufficient granularity to amortize the startup costs. If $T_{startup}$ is sufficiently small relative to T_c, then $T_{elapsed}$ is minimized as T_c is minimized. T_c is based on three quantities that can be computed efficiently at runtime using callbacks: T_{comp}, T_{comm}, and $T_{overlap}$. T_{comp} is the time that a processor spends in computation during a cycle of the dominant computation phase. T_{comm} is the time that a processor spends in communication during a cycle of the dominant communication phase. $T_{overlap}$ is the portion of T_{comm} that can be overlapped with T_{comp}. These estimates are computed for a particular configuration. For this configuration, A_i is first computed by Eq. 3 and these estimates are computed as follows (shown for processor p_i in cluster C_i):

$$T_{comp}[p_i] = S_i * computational\ complexity * A_i \qquad \text{(EQ 4)}$$

$$T_{comm}[C_i, \tau] = \beta b + \gamma \qquad \text{(EQ 5)}$$

$$T_c[p_i] = T_{comp}[p_i] + T_{comm}[p_i] - T_{overlap}[p_i] \qquad \text{(EQ 6)}$$

The *computational complexity* and S_i are determined by invoking the appropriate callbacks. For T_{comm}, the topology of the dominant communication phase (τ) is used to select the appropriate communication cost function at runtime (Eq. 1). For a fixed number of processors (i.e., p=P_i) the communication cost function reduces to the form of Eq. 1 where β and γ are constants and b is the size of messages exchanged (which may depend on A_i in some cases). All of these equations are computed at runtime using program information provided by the callback functions. We have shown T_c for a particular p_i, but it does not matter since T_c will be the same for all processors since the communication cost is the same and the computation is load balanced.

The heuristic works as follows. For each candidate processor configuration considered, the algorithm computes the associated A_i via Eq. 3. Once A_i is known, the elapsed time estimate T_c can be computed via Eq. 6 for this configuration. The canonical relationship between T_c and number of processors is depicted in Fig. 3.

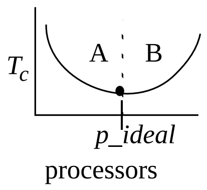

Figure 3. Elapsed time vs. processors

In region A, computation granularity is too large and insufficient parallelism is being exploited. In region B, computation granularity is too small and too many processors have been used. The ideal number of processors, *p_ideal*, occurs at the minimum of the curve. An iterative algorithm to locate *p_ideal* based on binary search has been developed. The algorithm assumes a single global minima. In some cases, several minima may be possible due to architecture or message-system protocol characteristics. An algorithm to deal with this more general case is being developed.

The algorithm begins by ordering the candidate processors (clusters) as discussed earlier. The algorithm then chooses clusters in this order. For example, suppose that a cluster containing N_1 processors is selected first. The algorithm then computes the number of processors within this cluster (1 .. N_1) to apply to the problem by trying different processor configurations and estimating the cost. Then the next cluster is tried, assuming the previous allocation of processors, and so on. The details of this algorithm may be found in [11].

This algorithm requires that Equations 3 and 6 are recomputed $Klog_2P$ times worst case, where K is the number of clusters and P is the total number of processors. We believe that this is a scalable algorithm. For smaller problems, both K and P will tend to be small. For example, suppose K = 5 and P = 20. In the worst case where all clusters and processors are selected, this will require that the equations are recomputed $5log_2 20$ or 20 times. Each time the equations are recomputed, the number of floating point operations executed is proportional to K (the loop bound on Eq. 3). This overhead is quite small.

6.0 Results

To demonstrate the feasibility of the partitioning method, we have implemented a dense NxN iterative five-point stencil application on a network of heterogenous workstations containing two clusters (6 Sun4 Sparc2's and 6 Sun4 IPC's on two ethernet-connected network segments joined by a router). The implementation uses a row decomposition of the underlying grid (i.e., the *PDU* is a row of the NxN grid) as depicted in Fig. 2. The callbacks for the dominant phases of the stencil computation were given in Section 4.0. Communication is handled by calls to the MMPS library [5].

The implementation arranges the tasks (i.e., processors) in a *1-D* topology. Since the Sparc2 is more powerful than the IPC, the Sparc2 cluster is considered first by the partitioning algorithm. Processors in the IPC cluster will only be used if the entire Sparc2 cluster is used. Task placement is important in the event that both clusters are used since router costs may be large. For the *1-D* topology placement is simple: tasks are assigned to the processors in the Sparc2 cluster followed by processors in the IPC cluster north to south. Only a single processor in each cluster needs to communicate across the router.

We have implemented two versions of the stencil computation to test the applicability of our approach, STEN-1 and STEN-2. In STEN-1, communication is not overlapped with computation. In STEN-2, transmission of the north and south borders is overlapped with the grid computation. We present results for STEN-1 and STEN-2 for several problem sizes: N=60, 300, 600, and 1200. All data presented is the result of multiple runs with averages shown. All benchmarking has been done when the network and processors were lightly loaded.

For STEN-1 and STEN-2, the partitioning algorithm computes the computation time estimate using callbacks (shown for processor p_i):

$$T_{comp}\,[p_i] = S_i * [5N] * A_\mathrm{i}$$

The algebraic term in square braces is the computational complexity or the number of operations or instructions per *PDU*, S_i is the average floating point instruction rate for p_i, and A_i is the number of *PDU*s (rows) for processor p_i. The communication time estimate T_{comm} was determined by benchmarking the *1-D* communication topology for different numbers of processors and message sizes. Since all processors are Sun4's, no message coercion was necessary. The parameter b is determined by the *communication complexity* callback and is equal to 4N for a problem of size N.

	STEN-1					STEN-2			
N	P_1	P_2	A_1	A_2		P_1	P_2	A_1	A_2
60	1	0	60	0		2	0	30	0
300	6	0	50	0		6	2	43	21
600	6	4	75	38		6	6	67	33
1200	6	6	171	86		6	6	171	86

Table 1. Results of partitioning algorithm for stencil computation

The message cost functions were determined to be (P_1 and P_2 are the number of Sparc2's and IPC's respectively, all in units of msec):

$$T_{comm}\,[C_1, 1\text{-}D] \approx (-.0055 + .00283P_1)b + 1.1P_1$$
$$T_{comm}\,[C_2, 1\text{-}D] \approx (-.0123 + .00457P_2)b + 1.9P_2$$
$$T_{router}\,[C_1, C_2] \approx .0006b$$
$$T_{comm}\,[1\text{-}D] \approx \max\,(T_{comm}\,[C_1, 1\text{-}D],\; T_{comm}\,[C_2, 1\text{-}D]) + T_{router}\,[C_1, C_2]$$

For P_2 = 2, T_{comm} doesn't fit the data well and may take on negative values. It turns out that the absolute value of this quantity is a very good approximation to the actual cost, so this solves the problem. We have also determined that S_i for the Sparc2 $\approx$ 0.3 usec and S_i for the IPC $\approx$ 0.6 usec. This is an average over obtained by benchmarking several floating point operations. A_i is computed to be (by Eq. 3):

$$A_i\,[p_i \text{ Sparc2}] = 2N / (2P_1 + P_2)$$
$$A_i\,[p_i \text{ IPC}] = N / (2P_1 + P_2)$$

The factor 2 is the approximate ratio of S_i for the Sparc2 and the IPC and reflects the fact that the Sparc2's are about 2 times faster than the IPC's (in floating point performance). Consequently, the tasks on the Sparc2's will receive 2 times more *PDU*s than the tasks on the IPC's. The computation time estimates are the following (in msec):

$$T_{comp}\,[p_i \text{ Sparc2}] = 0.0003*[5N]*[2N/(2P_1+P_2)]$$
$$T_{comp}\,[p_i \text{ IPC}] = 0.0006*[5N]*[N/(2P_1+P_2)]$$

The partitioning algorithm uses T_{comp} and T_{comm} to estimate T_c at runtime using the callbacks. For STEN-1, the implementation does not overlap communication and computation (i.e., $T_{overlap}$ = 0) and T_c is computed to be (in msec):

$$T_c\,[p_i] = T_{comp}\,[p_i] + T_{comm}\,[1\text{-}D]$$

For STEN-2, the implementation overlaps communication and computation. The amount of overlap is the minimum of the execution time estimates for T_{comp} and T_{comm}. $T_{overlap}$ and T_c are computed to be (in units of msec):

$$T_{overlap}\,[p_i] = \min\,(T_{comp}\,[p_i], T_{comm}\,[1\text{-}D])$$
$$T_c\,[p_i] = T_{comp}\,[p_i] + T_{comm}\,[1\text{-}D] - T_{overlap}[p_i]$$

The results of the partitioning algorithm are shown in Table 1 (P_1 and P_2 are the number of Sparc2's and IPC's respectively, and A_1 and A_2 are the number of *PDU*s (rows) given to each Sparc2 and each IPC respectively). In Table 2, we present the measured elapsed times for STEN-1 and STEN-2 with the minimum for each problem size denoted by *. The number of iterations is 10. The time for STEN-1 is the top entry in the table cell. These timings do not include the initial grid distribution cost. The sequential time shown is for the Sparc2. As expected, STEN-2 outperforms STEN-1 for all problem sizes due to communication overlap. More importantly, observe that the partitioning algorithm accurately predicted the best configuration for both STEN-1 and STEN-2 for all problem sizes. Notice also that the IPC's were not utilized until the problem was sufficiently large.

For N=1200, we also show the elapsed time obtained with an equal decomposition of the data domain for P=12 (each processor gets 100 rows). This clearly leads to a load imbalance and indicates the benefit of a heterogeneous data decomposition. In fact, a poor data decomposition has the effect of significantly reducing the effective parallelism. For example, the benefit gained by using 6 additional processors for STEN-1 is lost - using 6 Sparc2's results in a smaller elapsed time (3984 vs. 4157).

These results demonstrate that it is possible to automatically partition a data parallel computation at runtime such as the stencil computation. The method is also extremely efficient. Recall that the overhead is O($Klog_2P$). For K=2 clusters and P=12 processors, the equations are recomputed 6 times. This overhead is easily amortized since the elapsed times for the stencil computation are in the hundreds to thousands of msecs. There is additional overhead required to determine the available processors within each cluster but it is also small relative to elapsed time [11]. The stencil computation is very regular and has *uniform* computational and communication complexity. We expect the method to work well in these cases. We have also had success applying the method to Gaussian elimina-

N	elapsed time 1 Sparc2 msec	elapsed time 2 Sparc2s msec	elapsed time 4 Sparc2s msec	elapsed time 6 Sparc2s msec	elapsed time 6 Sparc2s + 2 IPCs msec	elapsed time 6 Sparc2s + 4 IPCs msec	elapsed time 6 Sparc2s + 6 IPCs msec
60	55	52*	75	78	86	96	98
	55*	56	70	71	82	88	90
300	1346	753	439	337*	338	346	361
	1346	709	394	313	266*	268	278
600	5535	2862	1511	1117	1059	985*	1099
	5535	2797	1453	1019	943	894	822*
1200	21985	11038	5699	3984	3758	3604	3088* (4157)
	21985	10972	5554	3770	3398	3230	2822* (3443)

Table 2. Measured elapsed times for STEN-1 and STEN-2. The * indicates the predicted minimum point.

tion with partial pivoting, an application that has *non-uniform* computational and communication complexity.

7.0 Summary and Future Work

We have presented a method for automatically partitioning data parallel computations across a heterogeneous network under a set of assumptions. The method relies upon information about the computation and communication structure of the implementation in the form of callbacks, and a set of topology-specific communication functions. At present, the callbacks are provided by the programmer, but we are exploring the possibility of compiler-generated callbacks. The preliminary results indicate that the method is efficient and that minimum elapsed times for two different implementations of the stencil computation were achieved. This is an encouraging result.

Future work includes applying the partitioning method to larger-scale data parallel computations and relaxing the assumptions about the network model. A more general partitioning algorithm will be needed for the latter. A strategy to handle load imbalance due to processor sharing is also the subject of future work. One possibility is to dynamically recompute the *partition vector* in the event of load imbalance. We also plan to demonstrate that our approach is applicable to a *metasystem* environment that may contain machines of different classes such as multicomputers and workstations together. We are currently implementing the partitioning algorithm within the Mentat [7], a parallel processing system developed at the University of Virginia.

8.0 References

[1] A.L. Cheung, and A.P. Reeves, "High Performance Computing on a Cluster of Workstations," *Proceedings of the First Symposium on High-Performance Distributed Computing*, Sept 1992.

[2] A. Beguelin et al., "HeNCE: Graphical Development Tools for Network-Based Concurrent Computing," *Proceedings SHPCC-92*, pp. 129-136, Williamsburg, VA, May, 1992.

[3] G. Fox et al, "Fortran D Language Specification," TR90-141, Department of Computer Science, Rice University, December 1990.

[4] P.J. Hatcher, M.J. Quinn, and A.J. Lapadula, "Data-parallel Programming on MIMD Computers," *IEEE Transactions on Parallel and Distributed Systems*, Vol 2, July 1991.

[5] A.S. Grimshaw, D. Mack, and T. Strayer, "MMPS: Portable Message Passing Support for Parallel Computing," *Proceedings of the Fifth Distributed Memory Computing Conference,* April 1990.

[6] A.S. Grimshaw, J.B. Weissman, E.A. West, and E. Loyot, "Metasystems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems," *Journal of Parallel and Distributed Computing*, in press.

[7] A.S. Grimshaw, "Easy to Use Object-Oriented Parallel Programming with Mentat," *IEEE Computer*, May 1993.

[8] V.M. Lo et al, "OREGAMI: Tools for Mapping Parallel Computations to Parallel Architectures," CIS-TR-89-18a, Department of Computer Science, University of Oregon, April 1992.

[9] N. Nedeljkovic, and M.J. Quinn, "Data-Parallel Programming on a Network of Heterogeneous Workstations," *Proceedings of the First Symposium on High-Performance Distributed Computing*, Sept. 1992.

[10] V.S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, December, 1990.

[11] J.B. Weissman, "Multigranular Scheduling of Data Parallel Programs," TR CS-93-38, Department of Computer Science, University of Virginia, July 1993.