# A Framework for Partitioning Parallel Computations in Heterogeneous Environments

**Jon B. Weissman**
**Andrew S. Grimshaw**

**Department of Computer Science**
**University of Virginia**

## Abstract

*In this paper we present a framework for partitioning data parallel computations across a heterogeneous metasystem at runtime. The framework is guided by program and resource information which are made available to the system. Three difficult problems are handled by the framework: processor selection, task placement, and heterogeneous data domain decomposition. Solving each of these problems contributes to reduced elapsed time. In particular, processor selection determines the best grain size at which to run the computation, task placement reduces communication cost, and data domain decomposition achieves processor load balance. We present results that indicate excellent performance is achievable using the framework. This paper extends our earlier work in partitioning data parallel computations across a single-level network of heterogeneous workstations*[1].

## 1.0 Introduction

A great deal of recent interest has been sparked within academic, industry, and government circles in the emerging technology of *metasystem*-based high-performance computing. A *metasystem* is a *shared* ensemble of workstations, vector, and parallel machines connected by local- and wide-area networks, see Figure 1. The promise of on-line gigabit networks coupled with the tremendous computing power of the metasystem makes it very attractive for parallel computations.

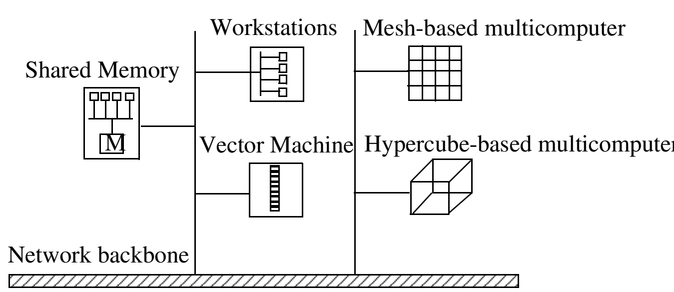



**Figure 1. A typical metasystem**

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

The potentially large array of heterogeneous resources in the metasystem offers an opportunity for delivering high performance on a range of parallel computations. Choosing the best set of available resources is a difficult problem and is the subject of this paper. Consider the set of machines in Table 1 and observe that they have different computation and communication capacities. Loosely-coupled parallel computations with infrequent communication would likely benefit by applying the fastest set of computational resources (perhaps the DEC-Alpha cluster), and may benefit from distribution across many machines. On the other hand, more tightly-coupled parallel computations are best suited to machines that have a higher communication capacity (perhaps an Intel Paragon), but may also benefit from distribution across many machines if the computation granularity is sufficient. We address the latter problem in this paper.

| Machine | Latency | Bandwidth | Processor speed |
|---|---|---|---|
| iPSC/2 | moderate | moderate | slow |
| Sparc ethernet cluster | high | low | moderate |
| DEC-Alpha cluster on Gigaswitch | high | high | very fast |
| Paragon | low | very high | fast |
| Cray T3D | low | very high | very fast |
| Sequent Symmetry | low | low | slow |

**Table 1. A spectrum of heterogeneity**

We present a framework that automates partitioning and placement of data parallel computations across metasystems such as in Figure 1. Partitioning is performed at runtime when the state of the metasystem resources are known. Three difficult problems are handled by the framework: processor selection, task placement, and heterogeneous data domain decomposition. Solving each of these problems contributes to reduced completion time. Processor selection chooses the best number and type of processors to apply to the computation. This is important because if too many processors are chosen the computation granularity will be too small, and if too few processors are selected the computation granularity will be too large. In either case, the elapsed time will be increased. Task placement is performed to limit communication costs, while data domain decomposition is done to achieve processor load balance.

To solve each of these problems, we exploit information about the computation and communication structure of the data parallel program and the computation and communication capacities of the metasystem resources. The former is provided by a set of *callback* functions that are described.

The partitioning framework is based on three runtime phases: *processor availability*, *partitioning*, and *instantiation.* Processor availability is a runtime monitoring process that determines the available set of processors based on the current usage of metasystem resources. Availability depends on the machine class.

For example, workstation availability is based on system load, while multicomputer availability is based on partition or subcube availability. Partitioning determines the number and type of processors to use from the available set, the placement of tasks across processors, and a decomposition of the data domain. Instantiation is an application-dependent phase that starts the data parallel computation across processors selected by partitioning using the determined task placement.

We begin by describing our metasystem model and the communication and processor resource information that supports the processor availability and partitioning phases. We discuss placement in the context of the metasystem communication model. Next we describe the data parallel computation model and the program information that is used to guide partitioning and placement. We then describe the partitioning method and an implementation to show how the resource and program information is used. Experimental and simulation results indicate that partitioning can be effectively automated for a large and useful class of data parallel computations. We also show that the partitioning can be done efficiently and excellent performance is achievable for these computations.

# 2.0 Framework

## 2.1 Metasystem Model

The metasystem illustrated in Figure 2 contains heterogeneous machines organized in a hierarchy. The basis of this organization is the *processor cluster* denoted by the large circles. A processor cluster contains a homogeneous family of processors that may contain workstations, vector, or parallel machines. For example, processor clusters range from tightly-coupled multiprocessors such as a Sequent in which processors communicate via shared-memory to distributed-memory multicomputers such as a Paragon or loosely-coupled workstations such as a Sun 4 cluster in which processors communicate via messages.

Communication between processors in different processor clusters is accomplished by message-passing. Taken as a whole, the metasystem is a multi-level distributed-memory MIMD machine. We will use the following notation throughout this section:

| | |
|---|---|
| $N_i$ = | the $i^{th}$ network cluster |
| $C_i$ = | the $i^{th}$ processor cluster |
| $\tau$ = | application communication topology |
| b = | message size in bytes |
| $c_1$ .. $c_4$ = | communication cost coefficients |
| f () = | processor-dependent communication function |
| $r_1$, $r_2$ = | router cost constants |
| $e_1$ = | coercion cost constant |
| k = | number of messages that cross through the router |

A *network cluster* contains one or more processor clusters and is denoted by the boxes labelled $N_1$, $N_2$ and $N_3$ in Figure 2. The key property of a network cluster is that it has private communication bandwidth with respect to other network clusters, and shared bandwidth with respect to the processor clusters it contains. For example, the total available bandwidth in the metasystem of Figure 2 is the sum of the bandwidth

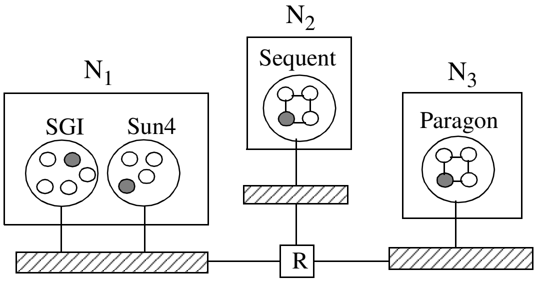



**Figure 2. Cluster-based metasystem organization**

in $N_1$, $N_2$ and $N_3$, but the available bandwidth in $N_1$ is shared between the Sun 4 and SGI clusters. Network clusters are connected by one or more routers. For wider-area metasystems, we define network clusters hierarchically as shown in Figure 3. Here $N_4$ is a network cluster that contains $N_1$, $N_2$ and $N_3$.

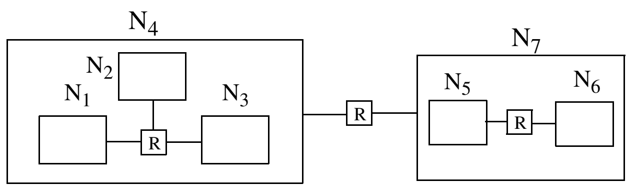



**Figure 3. Wider-area metasystem organization**

The metasystem organizations in Figure 2-3 are *locality-based* in which processors are grouped by physical connectivity. This has important implications for partitioning parallel computations in which minimizing communication overhead is key. For example, communicating processors within a processor cluster pay no routing penalty, while processors in different processor clusters pay a routing penalty. If the processors are within the same network cluster than a single hop routing penalty is incurred. We will see later in this section that task placement exploits this locality information.

In this paper, we focus on local-area metasystems such as in Figure 2 and restrict ourselves to multicomputers and workstations. Current network technology limits the benefit that tightly-coupled data parallel computations will receive by wider-area distribution due to high and often unpredictable communication costs. However, this is likely to change when the gigabit networks become more widespread. We also make the simplifying assumption of one processor cluster per network cluster. This assumption allows us to present a simpler communication model and partitioning method. We will refer only to processor clusters in the remainder of this paper.

Processor clusters maintain important resource information that will be exploited by partitioning and placement (the * is not implemented and the information in bold will be used in this paper):

- **Interconnection topology**
- **Processors** (total, avail)
- **Aggregate power** (mflops, mips)
- **Communication functions**
- B/W (peak, effective*, avail*)
- Latency (idle, effective*)
- Manager

This information is stored by a processor we call the *manager* and is denoted by the shaded circle in Figure 2. The topology refers to *bus* (ethernet), *ring* (FDDI), *mesh* (multicomputer), and *hypercube* (multicomputer) and may be different for different processor clusters. The *peak* bandwidth is the maximum communication bandwidth achievable for this cluster based on the processor type (assuming idle machines and network). Because this environment is dynamic and shared, both bandwidth and processing resources may be committed to other computations. The *effective* bandwidth may be less if processors in the cluster are loaded or unavailable, and the *available* bandwidth is the amount available to us based on the current traffic profile. The relationship between these quantities is: *peak* < *effective* < *available*. At present, we do not have a network monitor so information about the available bandwidth is not collected.

The available processors in a cluster may be less than the total number due to sharing. For example, when a workstation gets highly loaded or multicomputer partition or subcube gets allocated, we mark these processors as unavailable. The managers monitor the system state to determine availability within a cluster. This is known as *processor availability* and is the first phase of partitioning. For workstations, a simple load threshold policy is used to determine availability. We treat all processors below this threshold as available and equal in terms of computation and communication capacity. A more general strategy that is being investigated is to allow all processors to be available, but with adjustments to both communication capacity (i.e., effective bandwidth) and computation power (i.e., aggregate power) based on processor load. Operating system facilities are provided in Unix (*uptime, kmem*) and NX (*pspart*, *cubeinfo*) to determine processor availability. We do not discuss this process in any detail in this paper. The reader may assume that we can determine the available processor pool needed during partitioning. Aggregate power is the cumulative processing power based on the peak instruction rate for the processor type and the number of available processors.

## 2.2 Communication

The communication functions determine the cost of communication within the cluster and are the cornerstone of partitioning and placement. Choosing the appropriate number and type of processors depends on the communication cost that results from this selection. For example, choosing too many processors results in a small computation granularity and increased elapsed time due to high communication costs. Partitioning uses a set of functions to estimate communication costs for candidate processor selections.

In metasystems consisting of workstations and Intel multicomputers, communication is enabled by a reliable heterogeneous message-passing system (MMPS) [5]. MMPS uses UDP datagrams for communication among workstations and between processors in different clusters and NX for communication among processors in Intel multicomputer clusters. Partitioning requires that an accurate estimate of communication costs in the metasystem be known.

Consider the simple case where all communication occurs within a cluster $C_i$. The communication cost function for $C_i$ depends on the application communication topology and the interconnection topology of $C_i$. The particular cost experienced by an application depends on two *application-dependent* parameters provided to this function: (1) the size of messages exchanged, and (2) the number of communicating processors. In the model we present in the next section, these parameters will be known at runtime. We are exploring three communication topologies often found in data parallel computations: *1-D*, *ring*, and *tree.*

The *1-D* is common in scientific computing problems based on grids or matrices and is class of nearest neighbor topologies. In the *1-D* topology processors simultaneously send to their north and south neighbors and then receive from their north and south neighbors. The *ring* topology is common to systolic algorithms and pipeline computations. In the *ring* topology communication is much more synchronous. A processor receives from its left neighbor and then sends to its right neighbor. The *tree* topology is used for global operations such as reductions. In the fan-in, fan-out *tree* topology communication occurs in two phases. In fan-in a parent processor receives from all of its children before sending to its parent, while children simultaneously send to their parent. Once the root receives from its children the process is repeated in reverse during fan-out.

These communication topologies are *synchronous* in that *all* processors participate in the communication collectively. 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.

A set of accurate communication cost functions can be constructed for each cluster by benchmarking a set of *topology-specific* communication programs. These cost functions determine the average communication cost, measured as elapsed time, incurred by a processor during a single communication *cycle*. A cycle corresponds to a single iteration of the computation. For example in a single cycle of a ring communication, a processor receives one message from its left neighbor and sends one message to its right neighbor. For each cluster $C_i$ and communication topology $\tau$, 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 communicating processors within the cluster, and $b$, the number of bytes per message. For example suppose $C_1$ refers to the SGI cluster in Figure 2. 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 SGI cluster computed as elapsed time. 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):

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

The function $f$ depends on the cluster interconnect and the communication topology. For example, on ethernet we often see $f$ linear in $p$ for all communication topologies due to contention for the single ethernet channel. On the other hand, richer communication topologies such as meshes and hypercubes have greater communication bandwidth that scales more easily with the number of processors. For example, we have observed that for tree communication on a mesh, $f$ is *log* in $p$. Each communication cost 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 and interconnection topologies and is discussed later in this section.

If we decide to use processors within a particular $C_i$ only, then the cost function in (EQ 1) determines the communication cost. If processors in several clusters are used, then communication will cross cluster boundaries and two additional costs come into play:

$$T_{router}\ [C_i, C_j]\ (b) = r_1 + r_2\, b \qquad \text{(EQ 2)}$$
$$T_{coerce}\ [C_i, C_j]\ (b) = e_1 b$$

The router cost includes a latency penalty $r_1$ and a per byte penalty $r_2$ that captures any delay or buffering required in routing a message from a processor in $C_i$ to a processor in $C_j$. Since processors in different clusters may support different data formats, coercion may be needed. Coercion is paid as a per byte processor cost $e_1$ (e.g., endian conversion) by the sending or receiving processor. These costs are determined by benchmarking. Suppose that processors in $C_i$ and $C_j$ are communicating and no other clusters are used. The communication cost for processors in $C_i$ becomes the sum of the previous cost equation in (EQ 1) plus several new terms ($C_j$ may be written similarly):

$$T_{comm}\,[C_i, \tau] = T_{comm}\,[C_i, \tau] + k\,(T_{router}\,[C_i, C_j] + T_{coerce}\,[C_i, C_j])$$

where $k$ is the number of messages that cross between $C_i$ and $C_j$ per cycle. The router also increases contention for communication bandwidth and this is discussed in [17]. The total communication cost experienced by all processors, which we denote by $T_{comm}\,[\tau]$, depends on the topology $\tau$:

$$T_{comm}\,[\mathit{1\text{-}D}] = max_i\,\{T_{comm}\,[C_i, \mathit{1\text{-}D}]\} \quad \text{(EQ 3)}$$
$$T_{comm}\,[ring] = sum_i\,\{T_{comm}\,[C_i, ring]\}$$
$$T_{comm}\,[tree] = T_{comm}\,[C_{root}, tree] + max_{i \in \text{leaves}}\,\{T_{comm}\,[C_i, tree]\}$$

For the *1-D* topology, all processors communicate simultaneously and the total communication cost is limited by the slowest cluster (i.e., the cluster with the largest communication cost). For the synchronous *ring* topology, the communication cost is additive. The *tree* topology is more complicated. It has both concurrent communication (e.g., the leaves communicate simultaneously), and synchronous communication (e.g., communication is ring-like along the critical path). The cost is defined recursively.

Empirical evidence suggests that these cost functions are accurate. The benefit of this approach is that very accurate topology-specific communication costs can be estimated and these costs are key to making effective partitioning decisions. We are currently studying the impact of load on communication cost and how the communication cost functions may be adjusted at runtime to reflect reduced bandwidth. Once the cost functions are constructed they can be applied to other data parallel computations that contains these topologies.

## Placement

The communication cost functions in (EQ 1-3) are benchmarked using a set of task placement strategies. Task placement assigns tasks to processors in a communication efficient manner. Reducing communication costs is achieved by (1) maintaining communication locality (i.e., avoiding router crossings and potential coercion) and (2) effectively exploiting communication bandwidth within clusters. The former is achieved by *inter-cluster* placement and the objective is to minimize communication costs between clusters. The latter is achieved by *intra-cluster* placement and the objective is to minimize communication costs within clusters. Intra-cluster placement is also known as *mapping* or *embedding* and has been widely studied [10][11]. Algorithms for both forms of placement are topology-specific. Inter-cluster placement depends on the communication topology. In Figure 4, we present several inter-cluster placement strategies for the *1-D*, *ring*, and *tree* topologies across several clusters (the black boxes are routers). Notice that the number of router crossings or communication hops are minimized.

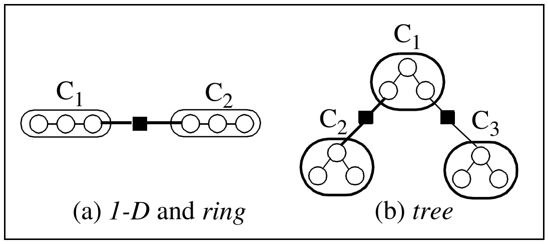



**Figure 4. Inter-cluster placement**

Intra-cluster placement is dependent on the communication topology (as is inter-cluster placement) and on the interconnection topology. Intra-cluster placement maps tasks to specific processors within a cluster. Two factors that contribute to intra-cluster communication costs are dilation (or hops) and contention. Intra-cluster placement should keep the average dilation small and limit contention. For example, a grey-scale mapping of a *1-D* topology onto a hypercube achieves minimal dilation and contention and has reduced communication overhead. On the other hand, a random placement suffices on a bus interconnect for any communication topology. High dilation and contention will tend to limit the exploitable communication bandwidth. There is a rich literature on the mapping problem and many of the algorithms are well known. Traditionally these algorithms have been applied within a static compile-time scheduling framework. Our approach will apply these algorithms in a novel way: within in a runtime partitioning framework. The algorithms for inter- and intra-cluster placement are used to benchmark the communication functions and are canned for use at runtime during partitioning.

## 2.3 Data Parallel Computations

A common approach for implementing data parallel computations in MIMD environments is SPMD [9]. In the SPMD model, the computation is performed by a set of identical tasks, placed one per processor, each assigned a different portion of the data domain. We have adopted a dynamic SPMD model in which tasks are instantiated at runtime based on the processor selection. We assume that the task implementation has been provided either by the user or as a result of a compilation process. A task executable for each architecture type is assumed to exist. The data domain is decomposed into a number of primitive data units or *PDU*s, where the *PDU* is the smallest unit of data decomposition. The *PDU* is problem and application 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 [9][12] and may arise from unstructured data domains. The partitioning method does not depend on the nature of the *PDU* but rather manipulates *PDU*s in the abstract.

Two views of the data parallel computation are provided to the partitioning framework: *task* view and *phase* view. In the task view, the computation is represented as a set of communicating tasks. The task view provides important topology information that is needed by placement. In the phase view, the computation is represented as a sequence of alternating computation and communication phases [11]. These phases are more tightly-coupled than the phases discussed in [13] which require data redistribution. A communication phase contains a synchronous communication executed by all processors as discussed in Section 2.2. A computation phase contains only computation. Communication and computation phases may be overlapped. Most data parallel computations are iterative with the computation and communication phases repeating after some number of phases. This is known as a *cycle*.

The phase view provides important information that is needed by partitioning. This information is provided by *callbacks* functions. The callbacks are a set of runtime functions that provide critical information about the communication and computation structure of the application that is used by the partitioning method. We discuss callback specification later in this section and present an implementation of callbacks in Section 2.5.1.

### Computation phase callbacks

Each computation phase must have the following callbacks defined:

- *numPDUs*
- *comp_complexity*
- *arch_cost*

The number of *PDU*s manipulated during a computation phase, *numPDUs*, depends on problem parameters (e.g., problem size). The amount of computation performed on a *PDU* in a single cycle is known as the computation complexity, c*omp_complexity*. It has two components: the number of instructions executed per *PDU*, and the number of instructions executed that are independent of the number of *PDU*s. The architecture-specific execution costs associated with *comp_complexity* are captured by *arch_cost,* provided in units of usec/instruction. The *arch_cost* contains an entry for each processor type in the target metasystem. To obtain the *arch_cost*, the sequential code must be benchmarked on each processor type.

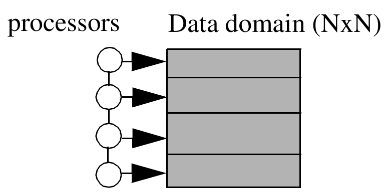
### Communication phase callbacks

Each communication phase must have the following callbacks defined:

- *topology*
- *comm_complexity*
- *overlap*

The topology refers to the communication topologies discussed in Section 2.2. The amount of communication between tasks is known as the communication complexity, *comm_complexity*. It is the number of bytes transmitted by a task in a single communication during a single cycle of the communication phase. Similar to *comp_complexity*, it has two components: the number of bytes transmitted per *PDU* and the number of bytes transmitted that are independent of the number of *PDU*s. It is used to determine the parameter $b$ in the communication cost equations. If a communication phase is overlapped with a computation phase, then the name of the computation phase is provided by the *overlap* callback. Among the computation and communication phases, two phases are distinguished. The *dominant* computation phase has the largest computation complexity, while the *dominant* communication phase has the largest communication complexity[2].

A simple example that illustrates the callbacks for a regular *N*x*N* five-point stencil computation is given in Figure 5 (the *arch_cost* is omitted). These are functions that return the values indicated. For *comp_complexity* we show only the *PDU*-dependent cost and for *comm_complexity* we show the *non-PDU*-dependent message size. This computation has been implemented using a block-row decomposition of the grid. In this application, 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.




(a) Stencil computation

*topology* ⇒ 1-D
*comm_complexity* ⇒ 4N (bytes)
*numPDUs* ⇒ N
*comp_complexity* ⇒ 5N (fp ops)

(b) Callbacks for stencil

**Figure 5. Example: 1-D stencil computation**

Notice that the callback functions may depend on problem parameters (e.g., *N*) that are unknown until runtime. The callbacks associated with the dominant phases are used by the partitioning method. In particular, the callbacks associated with the computation and communication complexity allow an estimate of the computation granularity to be computed at runtime. This estimate is used to determine the number of processors to use. The topology is used to select the appropriate communication function. The computation complexity is also used to determine a decomposition of the data domain, i.e., the number of *PDU*s to be assigned to each task. In a heterogeneous metasystem environment, processors may be assigned different numbers of *PDU*s for load balance. This information is contained in a structure known as the *partition_map* that is defined as follows:

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

$\sum A_i = numPDUs$

The *partition_map* has an entry for each processor and the association of its entries to processors is *topology-dependent*, see Figure 6. The topology-dependence reflects the data locality relationships in the prob-

2. The dominant phases may be problem or runtime dependent, and this is discussed in Section 2.5.1.

lem. This information is needed when the data domain is decomposed to processors. For example in the *1-D* stencil problem of Figure 5, a 20x20 grid might be decomposed across four processors as shown Figure 6a (processor 1 gets the first 2 *PDU*s or rows, processor 2 gets the next 5 *PDU*s, and so on). The *partition_map* is a logical decomposition of the data domain and is computed at runtime by the partitioning method. The application is responsible for using it in a manner appropriate to the problem. In Section 2.5, we sketch an implementation in which the grid is physically decomposed using the *partition_map* and the pieces are passed to the appropriate tasks.

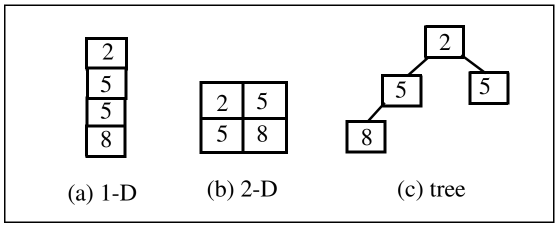



**Figure 6. Topology-dependent partition_map (numPDUs = 20)**

## Callback Specification

The callback mechanism is very powerful and can be applied to data parallel computations less regular than the five-point stencil. Since the callbacks may be arbitrary and complex functions, they can handle data-dependent computations by pre-processing the data domain. For example, the computation complexity for a sparse matrix problem depends on the non-zero structure of the matrix, and this is problem-dependent. But a simple callback can be written to capture this dependence. We are currently doing this with a finite-element code that we have written previously [15].

For irregular or control-dependent data parallel computations, it is likely that off-line benchmarking of the sequential code is needed to determine average values for some callbacks. A simple example is Gaussian elimination in which the amount of computation and communication changes from cycle to cycle. For irregular, control- or data-dependent computations it is likely that the domain programmer will have to write the callback functions by-hand. For regular problems such as the stencil computation, we believe that current compiler technology may be able to generate the callbacks if sufficient language support is provided. In the absence of compiler technology, we believe that domain programmers should be able to provide the callbacks. Another strategy is to provide libraries of callbacks for well-known computational structures and this technique is discussed in Section 2.5. Current compiler technology also looks promising for generating the SPMD task implementation and automatically decomposing the data domain for regular problems [2][9].

## 2.4 Partitioning

Partitioning is the second stage in the three stage process introduced in Section 1.0. Once *processor availability* has determined the available set of processors (the first phase), the partitioning phase computes the best subset of processors to use and a distribution of the data domain across these processors. Choosing the subset of processors in known as *processor selection* and the objective is to determine the best grain size at which to run the computation. Dividing the data domain is known as *data domain decomposition*, and the objective is to achieve processor load balance. Data domain decomposition was introduced in Section 2.3. A load balanced decomposition and an appropriate computation granularity lead to reduced completion time and this is the objective of partitioning. Partitioning is performed once at runtime. Dynamic repartitioning in the event of load imbalance could be accommodated with the framework, but it is outside the scope of this paper. We will use the following notation throughout this section:

| | |
|---|---|
| $p_i$ = | a particular processor |
| $A_i$ = | number of PDUs assigned to processor $p_i$ |
| $V_i$ = | number of available processors within cluster $C_i$ |
| $P_i$ = | number of processors selected for $C_i$ |
| $w_i$ = | relative processor weight for $i^{th}$ processor based on *arch_cost* |
| m = | number of clusters |
| g() = | the amount of computation as a function of $A_i$ |
| $d^{(i)}_1$ = | non-PDU cost constant for $i^{th}$ processor |
| $d^{(i)}_2$ = | PDU cost constant for $i^{th}$ processor |
| $T_c$ = | per cycle elapsed time |
| $T_{startup}$ = | startup overhead |
| $T_{comm}$ = | per cycle communication cost |
| $T_{comp}$ = | per cycle computation cost |

More formally, we define a *processor configuration* as a set of processors $P_i$ ($0 \le P_i \le V_i$, $i$=1 to $m$), where $V_i$ is the number of processors available within $C_i$. Partitioning computes a processor configuration that yields an appropriate computation granularity and a decomposition of the data domain $A_i$ associated with each $p_i$ (i.e., the *partition_map*) that load balances the processors. We discuss each of these problems in turn, beginning with data domain decomposition.

### 2.4.1 Data Domain Decomposition

We compute a load balanced decomposition based on the dominant computation phase. The amount of time spent in a single cycle of the dominant computation phase, denoted by $T_{comp}$, is defined as follows (shown for a processor $p_i$):

$$T_{comp}[p_i] = comp\_complexity * arch\_cost * g(A_i)$$

The computation time depends on the problem and processor characteristics and on number of *PDU*s, $A_i$, given to each processor. In the general case, the dependence on $A_i$ may be arbitrary function $g$ of $A_i$. In practice however, it is common that the dependence is linear for SPMD computations. Invoking callbacks for *comp_complexity* and *arch_cost* at runtime determines two cost constants that are problem-dependent: $d_1$ is a *non_PDU* cost and $d_2$ is a *PDU* cost.

The form for $T_{comp}$ becomes (shown for processor *i*):

$$T_{comp}\,[p_i] = d_1^{(i)} + d_2^{(i)} g(A_i) \qquad \text{(EQ 4)}$$

Load balance requires that $T_{comp}$ be the same for all processors (*P* total processors):

$$d_1^{(1)} + d_2^{(1)} g(A_1) = d_1^{(2)} + d_2^{(2)} g(A_2) = \dots\, d_1^{(P)} + d_2^{(P)} g(A_P)$$
$$\text{subject to } \sum A_i = numPDUs$$

If $g$ is non-linear then this is a difficult system to solve. Fortunately, $g$ is most often linear for SPMD computations in which the same computation is performed on each data element (i.e., *PDU*) independently. If $g$ is linear, we can combine these equations easily ($P_j$ comes in with the second constraint):

$$A_i = \left(\sum_j \frac{w_i}{w_j \cdot P_j}\right) \cdot \left[NumPDUs - \sum_{j \neq i} p_j \frac{d_1^{(j)} - d_1^{(i)}}{d_2^{(j)}}\right],\ w_i = \frac{d_2^{(i)}}{max\left\{d_2^{(j)}\right\}} \qquad \text{(EQ 5)}$$

If we assume that the *non_PDU* cost is 0 (i.e., $d_1^{(i)} = 0$), we get a simple equation for the *partition_map*:

$$A_i = \sum_i \frac{w_i}{w_j \cdot P_j} \cdot NumPDUs \qquad \text{(EQ 6)}$$

This equation has the property that faster processors will receive a greater share of the data domain and processors in the same cluster will receive an equal share. Since $A_i$ must be integral, the individual entries in the *partition_map* are rounded to the nearest integer. Computing the *partition_map* via (EQ 6) requires knowing the number of processors to be used (i.e., $P_j$). This is the subject of processor selection, discussed next.

### 2.4.2 Processor Selection

The job of processor selection is to choose a subset of available processors that are best applied to the computation. Nearly all parallel computations reach a point of diminishing returns with respect to the number of processors due to communication overhead. At this point, we have achieved the best computation grain for the problem and the elapsed time will be minimized. Because processor power and cluster communication capacities differ in the metasystem, locating this point is difficult. Our method is a heuristic that is guided by runtime cost estimation that uses information provided by the callbacks of Section 2.3.

The elapsed time $T_{elapsed}$, may be defined as follows:

$$T_{elapsed} = T_{startup} + \sum_{i=1}^{cycles} T_c[i] \text{ or} \tag{EQ 7}$$

$$T_{elapsed} = T_{startup} + cycles \cdot T_c \text{, where } T_c = \frac{\sum T_c[i]}{cycles} \tag{EQ 8}$$

The startup overhead, $T_{startup}$, may include initial data distribution costs or problem setup costs. The amount of time spent in the $i^{th}$ iteration or cycle is denoted by $T_c$[i] and the average over all $T_c$[i] is denoted by $T_c$. If $T_{startup}$ is small relative to the elapsed time, then minimizing $T_{elapsed}$ can be achieved by minimizing $T_c$. $T_c$ contains both computation and communication cost terms and is defined as follows:

$$T_c = T_{comp} + T_{comm} \text{ or} \tag{EQ 9}$$
$$T_c = max\ \{T_{comp}, T_{comm}\} \text{ if computation and communication are overlapped.}$$

Since $T_c$ is assumed to be an average, $T_{comp}$ and $T_{comm}$ must be computed as averages in the event that they differ from cycle to cycle. An example of this is Gaussian elimination discussed in Section 3.1. Our heuristic computes $T_{comp}$ and $T_{comm}$ based on the dominant computation and communication phases[3]. The form for $T_{comp}$ was given in (EQ 4) and the form for $T_{comm}$ was given in (EQ 3). The appropriate $T_{comm}$ will be selected by the topology of the dominant communication phase. The function $T_c$ is non-linear in the number of processors and possibly non-convex (if *max* appears). The optimal solution therefore requires minimization of a non-linear, possibly non-convex objective function, subject to constraints, and heuristics must therefore be used.

We describe two heuristics for processor selection, $H_1$ and $H_2$, that have yielded promising results. Both heuristics explore a series of processor configurations in an attempt to achieve a minimum $T_c$, hence minimized elapsed time. For each configuration explored, we compute $T_c$ via (EQ 9). To do this we first compute the *partition_map* $A_i$ via (EQ 6). Once $A_i$ is determined, we can compute $T_{comp}$ and $T_{comm}$ easily by invoking the callbacks and selecting the appropriate communication function. All of these computations are simple and can be performed efficiently at runtime. For a given configuration, the placement heuristics are used to determine task placement and the expected communication costs that result using this placement are computed by $T_{comm}$.

It is not possible to explore all processor configurations since the space is exponential in both the number of processors and clusters. The first step of the processor selection heuristics are to order the clusters in order to reduce the search space. The idea is that the best clusters should be explored first.

## Heuristic $H_1$

3. A more accurate and computationally expensive technique is to compute $T_c$ for all phases.

Heuristic $H_1$ has been designed for workstation network environments in which communication capacities are the same within the metasystem (e.g., ethernet-based clusters), and routing costs are high. The algorithm begins by ordering the clusters based on aggregate computation power. In this environment, the clusters with the highest aggregate computation power will often have the smallest communication costs. Using clusters with a higher aggregate computation power will lead to reduced completion time since $T_{comp}$ and $T_{comm}$ will both be smaller. The next stage of the algorithm explores the processor configuration space in a greedy fashion. All processors of a cluster are selected before processors in the next cluster are considered thus avoiding router crossings if possible. This algorithm tries to maintain communication locality by avoiding the router penalty and potential coercion overhead. The algorithm terminates when adding processors in the current cluster causes $T_c$ to increase, and is sketched in Figure 7.

```
Order clusters C1 .. Cm by aggregate computation power
Initialize curr_config, min_cost
For each cluster Ci {
        // Determine config that yields min Tc given previous Pj (j<i)
        best_curr_config = get_curr_config (curr_config, Ci);

        // If Tc has increased we are done
        if (best_curr_config.cost > min_cost)
             break;
        else {
             curr_config = best_curr_config;
             min_cost = best_curr_config.cost;
        }
}
return best_curr_config;
```

**Figure 7. Heuristic $H_1$**

The algorithm computes two things in get_curr_config: the best processor configuration (and task placement) based on the previous configuration and the current cluster, and the *partition_map*. It has the property that once $P_j$ is computed for cluster $C_j$, it is not modified in subsequent iterations. For each cluster it locates the best number of processors by a form of binary search, the details of which are provided in [17]. The worst-case order of this algorithm is $O(m\log_2 P)$ for $m$ clusters and $P$ total processors. The algorithm is scalable and the overhead in practice is small. In Section 3.1, we present some experimental results that indicate $H_1$ is efficient and produces excellent results.

A more general heuristic is needed when communication capacities differ in the metasystem or if router costs are not prohibitively high. The latter means that additional communication bandwidth may by effectively exploited across multiple clusters even with a router penalty. A greedy strategy will not work well in this case.

## Heuristic $H_2$

Heuristic $H_2$ has been designed for general metasystems and relaxes the assumptions made in $H_1$. Because communication capacities may be different, a simple cluster ordering strategy based solely on computation power will not always work well. For example, consider that a slow network of very fast machines such as a DEC-Alpha cluster might be chosen over a Paragon partition because the DEC-Alpha is faster than the i860. Clearly this may be a poor choice for some tightly-coupled parallel computations. Instead, we adopt a ordering heuristic based on $T_c$ since $T_c$ gives us a real measure of cost that includes both computation and communication. We apply $H_1$ to each cluster in isolation to compute the minimum $T_c$. We then order the clusters by this value.

A more complex two-phase strategy is adopted for exploring the processor configurations, see Figure 8. In phase 1, we add processors for the current cluster using the same algorithm as described in $H_1$. It is guaranteed that adding processors will decrease the $T_{comp}$ component of $T_c$. The addition of processors will never decrease $T_{comm}$, though it may remain unchanged. In phase 2, we try to reduce the $T_{comm}$ component of $T_c$. Recall from Section 2.2 that the total communication cost is a function of the communication cost contributed by each cluster based on intra-cluster placement, see (EQ 3). The cluster that contributes the maximum communication cost is targeted for reducing the overall communication cost. In phase 2, we add processors to the current cluster while removing processors from the cluster than contributed the largest communication cost. This is guaranteed to reduce $T_{comm}$, but the impact on $T_c$ is unpredictable since $T_{comp}$ may increase since we are trading potentially faster processors for slower ones. The cluster that contributes the largest cost may change during the course of phase 2 as processors are traded. The configuration that yields the minimum $T_c$ after both phase 1 and phase 2 is stored. This is the starting configuration that used as the next cluster is considered, much like in $H_1$. Unlike $H_1$, $H_2$ is conservative and does not terminate until all clusters are explored. The worst-case order of this algorithm is $O(mP)$: $O(mlog_2P)$ for cluster ordering, $O(mlog_2P)$ for phase 1, and $O(mP)$ for phase 2. In practice, the linear dependence on $P$ is tolerable since the amount of computation performed for each configuration is very small. We are exploring heuristics to improve the time bound for phase 2. In Section 3.2, we present some simulation results that indicate that $H_2$ is viable.

## 2.5 Implementation

The final phase of the partitioning framework is *instantiation*. Instantiation initiates the data parallel computation across the processor configuration using the determined task placement. We present a partial implementation of the stencil computation implemented in Mentat to illustrate the instantiation process. Mentat is parallel object-oriented parallel processing system based on C++ [7]. We also present a C++ callback interface that is currently being implemented.

```
Order clusters C_1 .. C_m by T_c
Initialize curr_config, min_cost
For each cluster C_i {
        // Phase 1 -- Try to reduce T_comp
        // Determine config that yields min T_c given previous P_j (j<i)
        best_curr_config = get_curr_config (curr_config, C_i)
        // min_cost is stored

        // Phase 2 -- Try to reduce T_comm
        curr_config.P_i = 0
        // Repeatedly trade processors in C_i with processors in C_k (k<i)
        //    where C_k is the cluster with the largest communication cost
        // C_k may change during phase 2 -- if it is the current cluster, exit
        while ((curr_config.P_i <= N_i) && (k!=i)) {
             curr_config.P_i++;
             curr_config.P_k--;
             T_c = get_Tc (curr_config);
             if (T_c < min_cost) {
                best_curr_config = curr_config;
                min_cost = T_c;
             }
        }
        curr_config = best_curr_config;
}
return best_curr_config;
```

**Figure 8. Heuristic $H_2$**

### 2.5.1 Callback Specification

The callbacks are encapsulated by a class called a domain, see Figure 9. The domain class can be tailored to the specific computation by means of derivation. For example, we have defined a stencil_domain derived from domain for stencil computations and we show the implementation of the comp_complexity callback for the single computation phase. This class is instantiated at runtime with a programmer-defined structure known as a parameter vector (PV) that contains any problem-dependent parameters that are used in the implementation of the callbacks (e.g., $N$ for the stencil problem). Notice that the callback functions may depend on the number of processors, np, that are determined at runtime. Also observe that the dominant computation and communication phases are determined by callbacks since the dominant phases may depend on problem parameters. The domain instance is used by the partitioning framework at runtime.

### 2.5.2 Program Interface

A Mentat code fragment for the stencil computation that uses the stencil_domain is given in Figure 10. The main program marshals the relevant program parameters into PV, instantiates the stencil_domain and calls the function partition that implements the partitioning heuristics of Section 2.4. This function

```
class domain {
char** PV;
public:
        virtual domain (char** PV);
        virtual phase dominant_comp_phase (int np);
        virtual phase dominant_comm_phase (int np);
        virtual phase_rec num_phases ();

        virtual comp_rec comp_complexity (int np, phase comp_phase);
        virtual int numPDUs (phase comp_phase);
        virtual cost_rec arch_cost (proc_type proc, phase comp_phase);

        virtual comm_rec comm_complexity (int np, phase comm_phase);
        virtual phase overlap (phase comm_phase);
        virtual top topology (phase comm_phase);
}

class stencil_domain: domain {
public:
        ...
        comp_rec comp_complexity (int np, phase comp_phase) {
                int N = atoi (PV[0]);  // extract problem size
                comp_rec *CR = new comp_rec;
                CR->PDU_inst = 5*N; // 5 fp operations per PDU in this problem
                CR->non_PDU_inst = 0;
                return *CR; }
        ...
}
```

**Figure 9. Domain class callback interface**

returns a structure that contains the processor configuration and *partition_map*. In the Mentat implementation, this structure is passed to a Mentat routine DP_create that instantiates a task (Mentat object) on each selected processor and passes the names of the neighboring Mentat objects to each Mentat object to enable communication. It is then up to application to start the data parallel computation and use the *partition_map* in an appropriate manner. In this code fragment, the grid is physically decomposed by the function 1D_carve, and the member function compute is called on each Mentat object. Each Mentat object is passed its portion of the problem via compute. The *1-D* communication to exchange north and south borders occurs within the implementation of compute.

# 3.0 Results

## 3.1 Experimental

We have obtained very promising results with $H_1$ on several real parallel codes: the stencil computation and Gaussian elimination with partial pivoting. The stencil code has been implemented on an hetero-

```
main() {
        stencil_class *workers, mo;
        stencil_domain *dom;
        ...
        // Problem-specific code: (N and Grid are read from file)
        PV[0]= itoa (N); // marshal PV for problem instance
        dom = new stencil_domain (PV); // instantiate domain

        PM = partition (dom);
        mclass* workers = (mclass*) DP_create (PM, mo);

        // Application-specific code
        1D_grid = 1D_carve (Grid, PM.partition_map);
        for (int i=0; i<PM.total; i++)
            workers[i].compute (1D_grid[i], N);
        ...
}
```

**Figure 10. Stencil main program**

geneous workstation network and Gaussian elimination on the Intel Gamma. In both environments, the communication capacities are the same (i.e., the workstation network is ethernet only and the Intel Gamma is a single machine with a hypercube interconnect), so $H_1$ is appropriate.

## Stencil

We have implemented two versions of the stencil code, STEN-1 and STEN-2 in which communication is handled by the MMPS communication library discussed in Section 2.2. STEN-2 overlaps computation and communication while STEN-1 does not. The callbacks for the stencil code were given in Figure 5b. The codes were run on two ethernet-connected clusters of Sun 4s joined by a router. Cluster $C_1$ contains 6 Sparc2s and $C_2$ contains 6 Sun 4 IPCs. The *arch_cost* for this problem was determined to be .3 usec and .6 usec for $C_1$ and $C_2$ respectively. Thus, the Sparc2s are about twice as fast as the Sun 4 IPCs and the *partition_map* will give each Sparc2 processor twice as many *PDU*s as each Sun 4 IPC via (EQ 6). The codes were implemented using a *1-D* topology and the communication cost functions were derived by benchmarking. Heuristic $H_1$ computes $T_{comp}$ and $T_{comm}$ by invoking callbacks with $T_c$ computed to be:

$T_c$ [STEN-1] = $T_{comp} + T_{comm}$

$T_c$ [STEN-2] = $max\ \{T_{comp}, T_{comm}\}$

We ran $H_1$ off-line on a range of problem sizes for both STEN-1 and STEN-2: $N$=60, 300, 600, and 1200. When STEN-1 and STEN-2 were run on the network using the processor configuration and *partition_map* computed by $H_1$, minimum elapsed times were obtained [17]. The codes were run multiple times when the network and processors were lightly loaded and averages reported. We also timed $H_1$ and found that the largest overhead was approximately 250 usec on a Sparc2, while the elapsed times for STEN-1 and STEN-

2 were in the hundreds of milliseconds. This overhead is quite tolerable and it was worth paying: an improper data domain decomposition increased elapsed time by as much as 35% due to load imbalance and an a poor processor configuration may increase elapsed time by a far greater percentage.

### Gaussian Elimination

We have implemented Gaussian elimination with partial pivoting (GE) on the 64-node Intel Gamma at Caltech with communication handled by the NX library. GE is implemented using a fan-in *tree* for pivot determination and a row-cyclic decomposition of the *N*x*N* matrix (i.e., the *PDU* is a row). Because this environment is homogeneous, decomposing the data domain is straightforward: each processor gets an equal number of *PDU*s. However, processor selection is still needed. Since the amount of communication and computation change from cycle to cycle in GE, we provide average values for *comp_complexity* and *comm_complexity*. The callbacks for the dominant computation phase (forward reduction) and the dominant communication phase (pivot exchange) are given in Figure 11.

*topology* ⇒ tree
*comm_complexity* ⇒ 4(N/2) (bytes)
*numPDUs* ⇒ N
*comp_complexity* ⇒ $(2/3N^3+N(N-1))/\ N(N-1)$
⇒ $(2N^2)/(3N-3)$ (fp ops)

**Figure 11. Callbacks for Gaussian elimination**

On average a pivot row of length *N*/2 is communicated and the average instruction count is obtained by taking the total instruction count [4] and dividing by the number of *PDU*s (*N*) and *cycles* (*N*-1). A single cluster $C_1$ of 32 i860 nodes was used. The *arch_cost* for GE was determined to be 0.18 usec and $T_{comm}$ was determined by benchmarking the *tree* topology on the Intel Gamma. Heuristic $H_1$ computes $T_{comp}$ and $T_{comm}$ by invoking callbacks with $T_c$ computed to be:

$$T_c\ [\mathrm{GE}] = T_{comp} + T_{comm}$$

We ran $H_1$ off-line on a range of problem sizes for GE: *N*=128, 256, 512, and 1024 and obtained the following processor configurations: $P_1$ = 2, 4, 16, and 28 respectively. Notice that not all processors are used. When GE was run on the Gamma using the processor configuration and *partition_map* computed by $H_1$, minimum elapsed times were obtained for *N*=128 and *N*=1024, and elapsed times within 10% of the smallest elapsed times were obtained for *N*=256 and *N*=512. Measured overhead for $H_1$ was a few hundred usec.

## 3.2 Simulation

A simulation study of heuristic $H_2$ was performed to test the efficacy of the algorithm. The simulation environment supports both synthetic problem and metasystem generation. Real costs can easily be inserted into the simulator to replace any synthetic cost if desired. For each problem instance/metasystem pair, the simulator computes the best $T_c$ by $H_2$ and then the optimal $T_c$ by exhaustive search of the processor configuration space and the comparative results are tallied.

A metasystem is determined by generating processor clusters together with the information described in Section 2.1. All generated information is uniformly distributed over a fixed range, see Figure 12. The ranges are limited to values that have been empirically observed and are reasonable. For example, a latency constant is restricted to be in the millisecond range on an ethernet-based cluster, while a bandwidth constant is restricted to be in the microsecond range. We simulated both ethernet-based clusters and mesh-based multicomputer clusters. For each communication topology (*ring*, *1-D*, *tree*), the communication cost functions are determined by generating the cost constants in (EQ 1) and (EQ 2). For bus interconnects, $f$ in (EQ 1) is linear in the number of processors for all topologies. For the mesh-based multicomputer, $f$ is $log\ p$ for the *tree* topology, linear for the *ring*, and nearly-independent[4] of $p$ for the *1-D* topology due to a dilation one intra-cluster placement of the *1-D* topology onto the mesh. The total communication cost $T_{comm}$ *is* computed by the functions in (EQ 3).

| metasystem parameters | problem parameters |
|---|---|
| num_clusters = [1 .. 5] | top = [tree, ring, 1-D] |
| num_processors_per_cluster = [1 .. 10] | NumPDUs = [1, 100, 500, 1000, 5000, 10000] |
| processor_rate = [1 .. 100] mflops | comm_complexity = [1 .. NumPDUs] bytes |
| interconnect = [mesh, bus] | arch_cost = [.1 .. 1] usec/instruction |
| latency_constant = [0 .. 1000] usec | comp_complexity = [1 .. 10000] instructions |
| bandwidth_constant = [.1 .. 10] usec/byte | |

**Figure 12. Simulation parameters**

A problem instance contains a communication phase and a regular computation phase that is linear and we use (EQ 6) to compute the *partition_map*. The values for *comp_complexity*, *arch_cost*, and *topology* are generated with uniform distributions to simulate a range of problem granularities and *numPDUs* takes on the values: 1, 100, 500, 1000, 5000, 10000. To keep things simple, all *non_PDU* terms are 0, and the phases are non-overlapping. The *arch_cost* is inversely proportional to the peak processor rate. For each problem instance, a number of values for *comm_complexity* on the range [1 .. *numPDUs*] are simulated since the size of messages depends on the how the problem was decomposed.

We present two sets of simulation results. In Table 2, only workstation clusters are simulated. In Table 3, both workstations and multicomputers are simulated. In both cases, we run in two modes: (1) zero router/coercion cost and (2) non-zero router/coercion cost. We separate results based on the application communication topology since $H_2$ has a slightly different formulation for each topology. For comparison we present the results for the same experiments but with cluster ordering turned off to show the benefit of

4. This is achieved by setting $c_3$ and $c_4$ very small.

this scheme. In all cases, cluster ordering is highly beneficial. Each table cell is the tabulation of 50 meta-systems and 900 problem instances for a total of 45,000 experiments. We show two values per cell, the % of runs within 5% of optimal and 10% of optimal respectively.

| **topology** | **w/ cluster ordering** (5%, 10% optimal) | **w/o cluster ordering** (5%, 10% optimal) |
|---|---|---|
| *ring* | 98.6, 99.5 | 63.9, 70.7 |
| *1-D* | 89.3, 94.4 | 72.7, 81.2 |
| *tree* | 91.6, 95.3 | 61.5, 68.8 |

**(a) No router/coercion**

| **topology** | **w/ cluster ordering** (5%, 10% optimal) | **w/o cluster ordering** (5%, 10% optimal) |
|---|---|---|
| *ring* | 98.7, 99.6 | 61.4, 67.7 |
| *1-D* | 88.9, 94.6 | 63.9, 70.7 |
| *tree* | 92.6, 95.9 | 52.5, 59.7 |

**b) router/coercion**

**Table 2. Simulation results: workstation clusters only**

The results in Tables 2-3 indicate that the algorithm performs equally well for multicomputer and workstations clusters. We see that cluster ordering is pivotal to the success of the algorithm - a 30% improvement in most cases. The algorithm finds a configuration that yields an elapsed time for $T_c$ that is within 10% of optimal (acceptably close in our view) over 90% of the time. We also see that the inclusion of router and coercion overhead does not perturb the performance of the algorithm. This validates our local ordering strategy based on $T_c$ in which we consider only communication costs within the cluster and not between clusters during cluster ordering. The performance results differ slightly between the different topologies with performance higher for the *ring* than for either the *1-D* or *tree* topologies. The reason is that $T_c$ is a more complex function for these topologies due to the presence of *max* in the formulation for $T_{comm}$, see (EQ 3), and $H_2$ is more prone to fall into a local minima.

| **topology** | **w/ cluster ordering** (5%, 10% optimal) | **w/o cluster ordering** (5%, 10% optimal) |
|---|---|---|
| *ring* | 97.7, 99.3 | 67.5, 76.4 |
| *1-D* | 91.4, 95.0 | 69.9, 76.5 |
| *tree* | 89.2, 91.7 | 63.3, 70.2 |

**(a) No router/coercion**

| **topology** | **w/ cluster ordering** (5%, 10% optimal) | **w/o cluster ordering** (5%, 10% optimal) |
|---|---|---|
| *ring* | 98.8, 99.7 | 64.3, 71.1 |
| *1-D* | 92.3, 96.4 | 65.9, 73.0 |
| *tree* | 88.1, 91.6 | 63.5, 71.1 |

**b) router/coercion**

**Table 3. Simulation results: workstation and multicomputer clusters**

The results from one sample run in Table 2 for the *1-D* topology is given in Figure 13 to illustrate the experimental parameters and a sample solution. Notice that the communication and computation capacities differ: $C_2$ has the most powerful processors (75 mflops), but $C_0$ has the greatest communication capacity (i.e., the smallest cost constants), this is analogous to the difference between a network of DEC Alpha workstations and an Intel Paragon. Also observe that the communication function $f$ (EQ 1) is linear in $p$ on the workstation network. The number of *PDU*s shown is the number given to each processor within each cluster (i.e., in $H_2$ we give 11 *PDU*s to each processor in $C_3$). For the optimal solution, we round-off the *partition_map* to integer values and make sure that the total equals *NumPDU*s.

$C_0$ = 6 processors, peak_rate = 45 mflops, $T_{comm}$[1-D] (p, b) = 88p + b (.56 + 1.04p)
$C_1$ = 4 processors, peak_rate = 16 mflops, $T_{comm}$[1-D] (p, b) = 577p + b (3.7 + 6.88p)
$C_2$ = 2 processors, peak_rate = 75 mflops, $T_{comm}$[1-D] (p, b) = 411p + b (2.7 + 4.85p)
$C_3$ = 10 processors, peak_rate = 55 mflops, $T_{comm}$[1-D] (p, b) = 91p + b (.58 + 1.07p)

NumPDUs = 100
Msg_Size = 100

$H_2$ processor configuration: $C_3$ = 5, $C_0$ = 5, $C_2$ = 0, $C_1$ = 0
$H_2$ partition_map: $A_3$ = 11, $A_0$ = 9 PDUs

optimal processor configuration: $C_3$ = 6, $C_2$ = 0, $C_1$ = 0, $C_0$ = 6
optimal partition_map: $A_3$ = 9.2, $A_0$ = 7.5 PDUs

$T_c$ [$H_2$] = 3347.8 usec, $T_c$ [optimal] = 3228.4 usec => 3.7% difference

**Figure 13. Sample simulator run**

# 4.0 Related Work

PVM [14] is the most widely used system for supporting parallel processing in heterogeneous environments. PVM provides low-level routines that manage communication and process interaction, but it has no support for partitioning. The programmer is responsible for problem decomposition and selecting the number of processors. Dataparallel C [9][12] provides a language and run-time system that supports both static and dynamic partitioning of regular data parallel computations across heterogeneous workstation networks. Dataparallel C is limited to workstations or multicomputers (but not both) and does not provide granularity control. An approach similar to our partitioning framework is adopted in [1]. Partitioning parallel computations on a heterogeneous network are guided by information provided by benchmarking. Classes of well-known parallel operations are benchmarked on different processor configurations and the best configuration selected by interpolation. The partitioning framework is more general and not limited to specific parallel operations.

The Augmented Optimal Selection Theory Model (AOST) and its variants [3] are targeted to more general metasystems such as in Figure 1. A model for statically mapping large-grain parallel computations onto a pool of dedicated heterogeneous parallel and vector machines is proposed. The applications appropriate for AOST contain loosely-coupled modules that are mapped to the machines that will execute them most efficiently. Their approach relies heavily on off-line benchmarking. Unlike AOST, the partitioning framework is based on a shared metasystem environment.

The Oregami project [11] adopts a philosophy similar to ours in that program phase information and resource information are used to guide partitioning and placement of parallel programs. A canned library of placement heuristics have been developed. We intend to adopt the same strategy for intra-cluster placement. Oregami differs from our framework in two significant ways: it is limited to homogeneous parallel machines and it is a compile-time approach.

# 5.0 Conclusion

We described a framework for partitioning data parallel computations across heterogeneous metasystems at runtime. Three difficult problems, processor selection, task placement, and heterogeneous data domain decomposition, are handled automatically by the framework. The framework is applicable to a large class of data parallel computations and the early results indicate that excellent performance is achievable. Our approach exploits program and resource information in a novel way. A model of data parallel computations was presented in which program information is made available by a powerful callback mechanism. A C++ implementation of callbacks based on domain classes was given. A hierarchical metasystem organization was also presented in which resource information is collected and maintained by the system. A key part of this model is a set of topology-specific communication functions that are central to the processor selection process. We are currently implementing the framework in Legion [8], a heterogeneous parallel processing system based on Mentat.

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

[3] R.F. Freund and H.J. Siegel, "Heterogeneous Processing," I*EEE Computer*, June 1993.

[4] G.H. Golub and J.M. Ortega, *Scientific Computing and Differential Equations*, Academic Press, Inc., 1992.

[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*, Vol. 21, No. 3, June 1994.

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

[8] A.S. Grimshaw et al, "Legion: The Next Logical Step Toward a Nationwide Virtual Computer," Computer Science Technical Report, CS-94-21, University of Virginia, June, 1994.

[9] 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.

[10] F.T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan-Kaufmann Publishers, 1992.

[11] 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.

[12] 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.

[13] D.M. Nicol and P.F. Reynolds, Jr., "Optimal Dynamic Remapping of Data Parallel Computations," *IEEE*

*Transactions on Computers*, Vol. 39, No. 2, February 1990.

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

[15] J.B. Weissman, Andrew S. Grimshaw, and Robert R. Ferraro, “Parallel Object-Oriented Computation Applied to a Finite Element Problem,” *Journal of Scientific Programming*, Vol. 2 No. 4, 1993.

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

[17] J.B. Weissman and A.S. Grimshaw, “Network Partitioning of Data Parallel Computations,” *Proceedings of the Third International Symposium on High-Performance Distributed Computing*, August 1994.