

---

# Compile-time Scheduling Algorithms for Heterogeneous Network of Workstations

MICHAŁ CIERNIAK, MOHAMMED JAVEED ZAKI AND WEI LI

*Computer Science Department, University of Rochester, Rochester, NY 14627, USA*  
*Email: {cierniak,zaki,wei}@cs.rochester.edu*

---

In this paper, we study the problem of scheduling parallel loops at compile-time for a heterogeneous network of workstations. We consider *heterogeneity* in various aspects of parallel programming: *program, processor, memory* and *network*. A heterogeneous program has parallel loops with different amount of work in each iteration; heterogeneous processors have different speeds; heterogeneous memory refers to the different amount of user-available memory on the machines; and a heterogeneous network has different cost of communication between processors. We propose a simple yet comprehensive model for use in compiling for a network of processors, and develop compiler algorithms for generating *optimal* and *near-optimal* schedules of loops for load balancing, communication optimizations, network contention and memory heterogeneity. Experiments show that a significant improvement of performance is achieved using our techniques.

*Received October 31, 1996; revised July 15, 1997*

---

## 1. INTRODUCTION

Network based distributed computing has attracted a lot of attention lately, due to the recent advances in high speed networks (e.g. ATM - Asynchronous Transfer Mode) having low latency and high bandwidth, and due to the ubiquitous presence of workstations linked over LANs. With most of the machines in a network underutilized, there has been research on harnessing this power in a useful way, for example, to use the workstations to solve the so called “grand challenge” problems. Furthermore, such a network of machines – the “virtual parallel machine”, may consist of possibly different types of processors, along with vector and multi-processor machines. The inherently dynamic nature of the configuration of the virtual machine, depending on which machines are available at the time of running the program, makes architecture-dependent programming almost impossible. The role of generating an efficient parallel code must be filled by compilers.

The sources of heterogeneity in a network of workstations (NOW) include the processors, with processors of different speeds; the memory, with different amount of available memory on different machines; the network, with varying cost of communication among pairs of processors; and at the program level, where the program may have parallel loops that have varying amount of work in each iteration. There are a number of research issues that crop up in this environment, such as:

- **Program Domain:** Parallel applications fall into a number of categories. These programs may have regular or irregular computation and communication, or they may be composed of several sub-tasks with different processor/machine affinities. Other characteristics, such as the communication-to-computation ratio, could dictate the decision to parallelize and the parallelization used.
- **Different parallelizations:** In heterogeneous environments, problem decomposition and task placement can have dramatic effects on performance. Depending on the underlying machine architecture and other machine-specific characteristics, different parallelizations may be required for good performance on different machines.
- **Machine/Processor Heterogeneity:** A heterogeneous system may consist of various shared and distributed memory MIMD machines, SIMD and vector machines, and sequential workstations interconnected by a network. This has a significant impact on scheduling and load-balancing.
- **Processor Selection:** Typically a large number of machines may be available for use, but we have to select the optimal subset of these machines which will give us the minimum overall execution time. We have to tradeoff increased computation power versus the increased overhead as we increase the number of machines.
- **Mapping:** Both the programs and machines may

have certain characteristics which require the sub-tasks to be mapped to specific machines, to obtain the best performance [3, 9].

- **Memory:** The amount of physical memory may be different for different machines. When we want to decide on the largest problem that can be efficiently performed, we must consider the available memory on each machine, and also the memory requirements of the application. We will discuss this point in more detail in section 7.
- **Network Latency:** Network latency is one of the primary concerns for a heterogeneous NOW. High latency can make communication extremely expensive, and restrict the scalability of the system.
- **Network Bandwidth:** Bandwidth is also a bottleneck, especially for Ethernet LAN. Although it is easier to increase the physical bandwidth (e.g. ATMs have a much higher bandwidth), the amount of application level bandwidth remains a small fraction of the available physical bandwidth [19]. With different interconnection networks, the network heterogeneity can become a significant factor in the parallel performance of applications.
- **Contention Effects:** Communication on an Ethernet LAN is more expensive due to high latency and low bandwidth. The network traffic tends to be highly bursty on LANs. Moreover, contention for the bus becomes a critical performance factor. Any performance prediction model for a heterogeneous NOW must take into account the contention that may be caused in the network. Modeling this is a very complex task.
- **Load Balancing:** Homogeneous static load balancing algorithms must be adapted to work for heterogeneous NOWs. For multi-user set-up, runtime dynamic load-balancing may be required [27]. We have to trade-off the task-switching cost versus the load-imbalance cost. There are many dynamic load balancing schemes [21, 16], but these cannot be used for problems with subtasks of various capabilities. Since NOWs are usually *loosely-coupled*, i.e., are connected via non-dedicated network, there is also the issue of external load on the network.
- **Data Coercion:** Machine heterogeneity entails different methods of data representation on different machines. Although this introduces the overhead of data conversion while sending messages among the machines, it is generally not that significant [11].
- **Software Issues:** Differences in the host operating systems, file systems, database systems, inter-process communication, compilers and languages available should be masked while dealing with heterogeneous systems. Efficient software systems are needed which automate most of the decisions that need to be made in such environments such as automating the data decomposition, distribution, synchronization and communication for the appli-

cations across a wide range of platforms.

In this paper, we assume an SPMD/Master-Slave model of computation, i.e., all processes essentially execute the same program, but on different data-sets. We further assume that all the parallelism comes from doall loops. Since all the tasks are similar, the problem consists of efficient data partitioning among a set of machines, taking into consideration the processor speeds and communication costs, so as to minimize the execution time of the program.

The objective of this paper is to propose compile-time techniques for scheduling parallel loops for a heterogeneous NOW. In particular, we make the following technical contributions:

- We propose a simple model for a heterogeneous network of machines. It serves as a conceptual starting point in compiling for load balancing and communication in such an environment.
- We show by experiments that the conventional ways of measuring processor speed and memory capacity are insufficient for a heterogeneous NOW. We show that *normalized processor speed*, which may be application dependent, gives a better estimate of processor performance, and that the *resident memory size*, which may also be application dependent, gives a better estimate of the memory requirement. Furthermore, we show how these two parameters taken together influence scheduling, and lead to better performance.
- We develop a set of *architecture-conscious* compile-time scheduling approaches for generating optimal or near-optimal scheduling of loops for load balancing and communication, for a network of heterogeneous machines.
- We present experimental results to verify that these techniques produce very good results in practice. We show that the architecture-conscious scheduling algorithms result in much better performance than the naive *architecture-oblivious* scheduling approach. Examples are drawn from a mix of synthetic and real applications, from scientific computing and economics modeling [17].

The rest of this paper is organized as follows. We will briefly present the related work in the next section, before introducing our compile-time model. In section 3, we introduce our program model, which is followed by our machine model in section 4. In section 5, we consider scheduling for heterogeneous programs (on homogeneous machines). In section 6, we look at the case of heterogeneous processors, with the same communication links, which is followed by scheduling for heterogeneous memory in section 7. Section 8 deals with the case where the network communication links are heterogeneous, i.e., there are different communication costs between different points in the network. In section 9, we extend our model to handle the case of scheduling

for load balancing while avoiding network contention. We then present experimental results on our proposed techniques in section 10. Finally, we conclude in Section 11.

## 2. RELATED WORK

In this section we look at some of the load balancing schemes which have been proposed in the literature.

### 2.1. Static Scheduling

Compile-time *static* loop scheduling is efficient and introduces no additional runtime overhead. For UMA (Uniform Memory Access) parallel machines, usually loop iterations can be scheduled in *block* or *cyclic* fashion. For NUMA (Non-Uniform Memory Access) parallel machines, loop scheduling has to take data distribution into account [14]. The simplest approach is the *static block* scheduling scheme, which assigns equal block of iterations to each of the available processors. *Static interleaved* scheme assigns iterations in a cyclic fashion [20].

There has been relatively little work in static scheduling for heterogeneous network of workstations. It has also mainly focused on homogeneous applications. Earlier work dealing with processor heterogeneity appears in [4, 8, 11], and the requirements for distributed computing over LANs have been analyzed in [19]. This paper presents compile-time static scheduling algorithms for heterogeneous programs, processors, memory, and network. Preliminary results of this paper can be found in [26, 6, 7].

### 2.2. Dynamic Scheduling

When the execution time of loop iterations is not predictable at compile-time, runtime *dynamic* scheduling can be used at the additional runtime cost of managing task allocation. The dynamic scheduling strategies fall under different models, which include schemes based on predicting the future from past loads, the *task queue model*, and the *diffusion model*.

**Predicting the Future:** A common approach taken for load balancing on a workstation network is to predict future performance based on past information. For example, in [18], a global distributed scheme is presented, and load balancing involves periodic information exchanges. Dome [1] implements a global central and a local distributed scheme, and the load balancing involves periodic exchanges. Siegell [23] also presented a global centralized scheme, with periodic information exchanges. The main contribution of this paper was the methodology for automatic generation of parallel programs with dynamic load balancing. In Phish [2], a local distributed receiver-initiated scheme is described, where the processor requesting more tasks, chooses a processor at random from which to steal more work.

CHARM [22] implements a local distributed receiver-initiated scheme. In [27] the authors presented different strategies for dynamic load balancing in the presence of transient external load. They examined both global vs. local, and centralized vs. distributed schemes, and presented a hybrid compile and run-time system that automatically selects the best load balancing scheme for a given loop/task from among the repertoire of different strategies.

**Task Queue Model:** A host of approaches have been proposed in the literature targeting shared memory machines. These fall under the *task queue model*, where there is a logically central task queue of loop iterations. Once the processors have finished their assigned portion, more work is obtained from this queue. The simplest approach in this model is *self-scheduling* [24], where each processor is allocated only one iteration at a time. In *fixed-size chunking* [13], each processor is allocated  $K$  iterations, while in *guided self-scheduling* [21] each processor is assigned  $1/P$ -th of the remaining iterations, where  $P$  is the number of processors. *Affinity scheduling* [16] also takes processor affinity into account. A number of more elaborate schemes based on the self-scheduling idea are also extant.

**Diffusion Model:** Other approaches include *diffusion models* with all the work initially distributed, and with work movement between adjacent processors if an imbalance is detected between their load and their neighbor's. An example is the *gradient model* [15] approach.

## 3. PROGRAM MODEL

In this paper we will look at parallel loops, that is, loops whose iterations do not depend on one another. We have to address two issues — the amount of computation and the amount of communication in each iteration of the parallel loop. We would like to generate schedules for the loop which are optimal, both in terms of communication and computation, to achieve the best speed-up on a heterogeneous NOW.

### 3.1. Parallel loops

To create a static schedule with a good load balance, we have to know the amount of computation in every iteration. We consider two cases of parallel loops: *homogeneous* and *heterogeneous*. By homogeneous loops, we mean parallel loops that have the same amount of computation in each iteration. Heterogeneous loops have varying amount of work in each iteration. We introduce a program parameter into our model:  $x_i$  — number of operations in iteration  $i$ .

For the homogeneous case,  $x_i$  is constant; for the heterogeneous case, we assume that  $x_i = ai + b$ , i.e., we restrict our attention to loops where the computation is an affine function of the normalized loop index, which captures a large set of scientific programs. Examples include Cholesky, TRFD from the Perfect benchmark suite [12], and SPEC95 benchmark applications.

### 3.2. Communication

With the right placement of data, a parallel loop does not require any communication during execution, i.e., there is no data flow between iterations of a parallel loop. However, communication may be necessary between different loops. Therefore, there may be communication caused by each iteration because of subsequent computation. We assume that each iteration of the parallel loop contributes a precisely defined amount of data to the messages sent after the loop completes. Further, we assume that the messages are being sent after all iterations assigned to a given processor have completed, i.e., there is only one message per processor, consisting of data from all iterations, rather than one message per iteration.

The example below illustrates this type of communication.

```

1: DOALL  $k = 1, n$ 
       $Y(k) = Y(k) + W(k)$ 
end DOALL
2: DOALL  $k = 1, n$ 
      do  $j = 1, n$ 
         $A(k) = A(k) + V(k, j) * Y(j)$ 
      end do
end DOALL

```

Values  $Y(k)$  produced in some of the iterations of loop 1, will be used in more than one iteration of loop 2. Every iteration of loop 1 contributes one word to the message sent between loops 1 and 2.

We introduce the following program parameter which denotes the amount of communication:  $y_i$  — number of bytes that have to be sent as the result of iteration  $i$ . Communication contributed by each iteration can be constant (the *homogeneous* case) — as in the example above — or it can vary (the *heterogeneous* case). In the heterogeneous case, we assume that  $y_i = ci + d$ , that is, the amount of communication is an affine function of the normalized loop index.

## 4. MACHINE MODEL

The machine model must account for processor, memory, and network heterogeneity. Below we present each case separately.

### 4.1. Processor Model

In a fully detailed processor model, we would need to consider the speed of a processor in terms of the number of floating point operations per second, and the number of integer operations per second. We also need to consider memory access time, and the interaction of these with different cache and memory sizes. Multiple instruction issue and instruction pipelining would further complicate the performance model. The multitude of machine parameters makes their use in performance prediction very difficult. Therefore, for our discussion,

we will only consider a single parameter,  $\gamma$ , to describe the speed of a machine:  $\gamma_i$  — time for one operation on processor  $i$ . The machines may be homogeneous or heterogeneous.

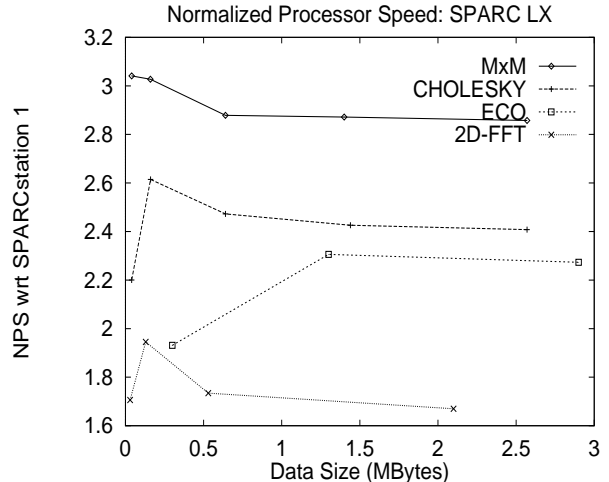


FIGURE 1. NPS: Sun SPARC LX vs. SPARC 1

There are a number of ways to calculate,  $\gamma$ , the speed of the processor. For example, we could use the MIPS (*million instructions per second*), MFLOPS (*million floating-point operations per second*), Whetstone, or the Dhrystone ratings. In modern processors different operations have different cost, and furthermore, instruction pipelining and multiple instruction issue render it quite hard to come up with a single figure that characterizes the performance. Therefore, while these figures may give an indication of the processor capabilities, reliable and consistent performance measure can only be found by using the execution time of different real applications on the machines in consideration.

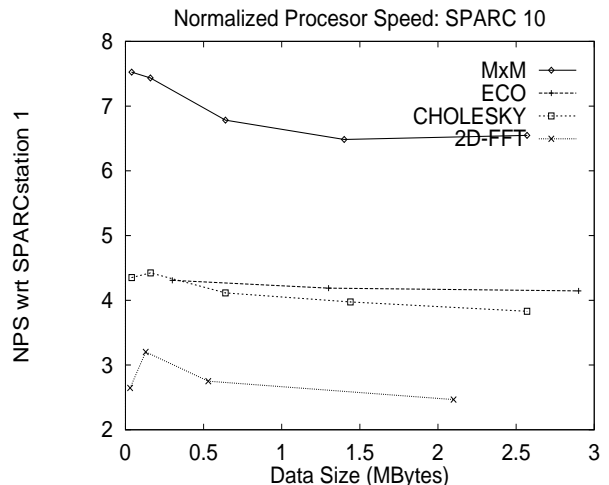


FIGURE 2. NPS: Sun SPARC 10 vs. SPARC 1

In our approach, we summarize the processor speeds via the notion of *normalized processor speed* (NPS), defined as the ratio of the time taken to execute on the

processor in consideration, with respect to the time taken on a base processor. Consider figures 1 and 2, which show the processor performance of SPARCstation 10 and LX on different applications (see section 10 for a description of these applications). The time is normalized against the performance of SPARCstation 1. Our experiments indicate that machine performance varies for different applications. Since the processor speeds vary from one application to another, we approximate the speed based on small trial application runs. On the other hand, we may obtain these by compile-time performance prediction. In [25], the author describes a detailed, architecture-specific, compile-time performance prediction framework. Porting to different architectures and compilers is quite involved, though possible.

**TABLE 1.** Performance Ratios: SPARC 5 wrt SPARC LX

MIPS ratio	Normalized Processor Speed		
	MxM	CHO	2D-FFT
3.5	1.5	1.7	1.8

Table 1 shows the MIPS ratio and the normalized processor speeds for different applications, for the SPARC 5 vs. LX. In table 2 we show the execution times obtained on a configuration of 2 machines – a SPARC 5 and a LX. The second and third columns show the execution time using the MIPS ratio and NPS values to balance the work load. It is clearly seen that the normalized processor speed should be used in scheduling, since it results in a balanced computation load and hence better performance for the application.

**TABLE 2.** Running Time (SPARC 5 + SPARC LX)

Application	MIPS	NPS
MxM(600)	1486.3s	1152.7s
CHO(600)	99.1s	87.1s
2D-FFT(512)	36.8s	32.3s

#### 4.2. Memory Model

Heterogeneous NOWs have a large amount of computational power as well as a large amount of combined memory. We would like to exploit the available resources by solving as large a problem as possible. For example, solving large instances of numerical scientific applications, and other real world applications like weather modeling, computational dynamics, and other “grand challenge” applications. The largest data size is limited by the amount of combined memory present in the system. The amount of memory available may be different for different processors. To summarize the

memory heterogeneity, we introduce the parameter,  $m_i$  – the amount of memory available on processor  $i$ .

**TABLE 3.** Memory Capacities

machine type	total memory	available memory
Sun SPARC 1	16Mb	12.5Mb
Sun SPARC LX	32Mb	24.5Mb
Sun SPARC 5	32Mb	24.5Mb
Sun SPARC 10	128Mb	102Mb

Table 3 shows the the amount of actual physical memory and the amount that is available to user applications in our configuration. We can use the above values to decide on the largest problem size we can run, by calculating when the total memory requirement of an application would exceed the user-available memory capacity on a given machine. Ways of estimating the memory requirement for an application will be discussed in section 7.

#### 4.3. Network Model

For a network of workstations, we also have to consider the cost of communication between any two machines, i.e., we must consider the interplay of latency and bandwidth between point to point in the network. Furthermore, when we talk of communication between two machines we must consider the cost of packing (marshaling) the data, receiving the data, and the cost of the “real” communication, that is, the time actually spent in the physical medium. We have two parameters for each of the above three cases — the startup time (independent of the message size), and the actual time spent in performing the action (proportional to the message size).

Rather than dealing with six or more parameters, we simplify our model and consider the startup time and the cost for the action to be the sum of the costs for all three stages, and thus have the following two parameters:  $\alpha_i$  — startup time for a message on processor  $i$ , and  $\beta_i$  — time to send one byte of data on processor  $i$ . The network of machines can be either homogeneous or heterogeneous. In the former case,  $\alpha_i = \alpha$  and  $\beta_i = \beta$ , for all the machines. In the latter case, these vary with the machine. The values for the latency and band-width are obtained via off-line network characterization experiments.

The discussion so far assumed that messages from different machines can be sent at the same time. For many machines this is not a realistic assumption. Contention in the network adds complexity to the model. The discussion of this, more complex, case will be deferred until Section 9.

## 5. SCHEDULING FOR HETEROGENEOUS PROGRAMS

In this section we consider heterogeneous programs (parallel loops) on parallel machines with homogeneous processors and a homogeneous network. As discussed in Section 4, the following machine parameters describe this type of machines:  $p$  denotes the number of processors in the system,  $\gamma$  the time to execute one operation,  $\alpha$  the communication initialization time,  $\beta$  the time to send one byte of a message, and  $n$  the number of iterations of the loop.

As a simple introduction to loop scheduling, we first consider homogeneous parallel loops without communication. Every processor has the same speed, every iteration requires the same amount of computation, and there is no communication. With these assumptions, every processor should execute approximately the same number of iterations. If  $n$  is a multiple of  $p$ , every processor will have exactly the same number of iterations:  $n/p$ . Otherwise, some processors will execute  $\lfloor n/p \rfloor$  while others will have  $\lfloor n/p \rfloor + 1$  iterations. In this case, it is not important which processors have one more iteration to execute. In the case of homogeneous loops with communication, every iteration causes the same number of bytes to be sent. Therefore, the static scheduling above will evenly distribute both computation and communication.

### 5.1. Heterogeneous Loops; No Communication

For heterogeneous loops, again, we deal with the communication-free case first. As discussed earlier in this section, this type of a loop is characterized by the parameter,  $x_i = ai + b$ , for  $i = 1, \dots, n$ . Rather than solving this problem directly, we will show how to transform this loop into a homogeneous parallel loop and use the scheduling strategy for homogeneous loops presented above. This transformation is graphically shown in Figure 3.

#### 5.1.1. Case I: $n = 2pt$ , for some integer $t$

We first consider a special case when the number of iterations  $n$  is a multiple of  $2p$ . We can transform this loop into a homogeneous parallel loop with  $n/2$  iterations. Note that the sum of the work in iterations  $i$  and  $(n - i + 1)$  is a constant:

$$x_i + x_{n-i+1} = ai + b + a(n - i + 1) + b = a(n + 1) + 2b$$

We can therefore combine iterations  $i$  and  $(n - i + 1)$  into one iteration of a new parallel loop. This new loop is homogeneous with  $a(n + 1) + 2b$  operations in every iteration. As all processors execute exactly  $n/p$  iterations of the transformed loop, there is no imbalance.

#### 5.1.2. Case II: $n \neq 2pt$ , for any integer $t$

In the general case, there may be imbalance. Let  $r = n \bmod (2p)$ . If  $r \neq 0$ , the imbalance is caused

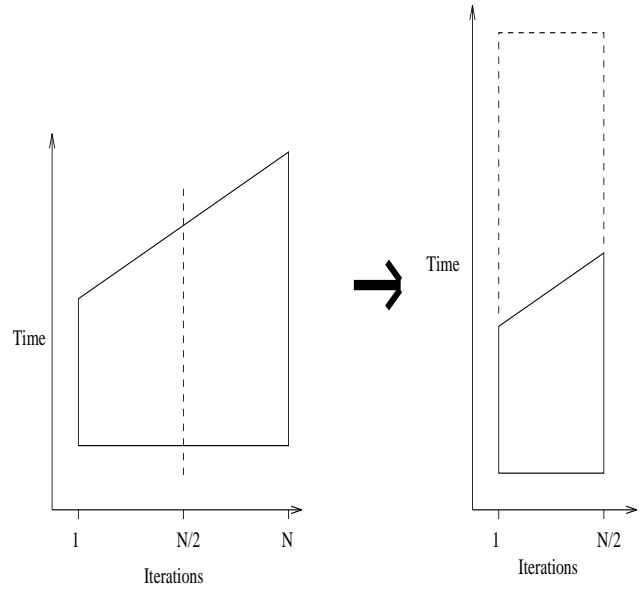


FIGURE 3. Transforming a heterogeneous loop into a homogeneous loop

by the remaining  $r$  iterations. We can make the imbalance very small by choosing those  $r$  iterations to be very short (the loop is not homogeneous). To achieve this, we take the first  $r$  iterations if  $a > 0$ , since we have an increasing amount of computation in this case, and the last  $r$  iterations otherwise. Now, if  $r \leq p$ , then  $r$  processors get one iteration each, otherwise the first  $r \bmod p$  processors get two iterations (we can transform the  $2(r \bmod p)$  consecutive iterations into a homogeneous loop), the remaining processors take the longest  $2p - r$  iterations. The schedule obtained in this way is close to optimal. We call this approach *bitonic scheduling* [7], since the iterations are assigned to processors in an increasing and decreasing fashion.

We shall illustrate this optimization with the following example. Let the number of iterations,  $n = 10$ , and the number of processors,  $p = 3$ . Let  $x_i = i$  i.e.,  $a = 1$  and  $b = 0$ . To get the optimal schedule, we first compute  $r = 10 \bmod 6 = 4$ . Because  $a > 0$ , we take away the first four iterations. The last 6 iterations can be perfectly balanced with each processor getting 2 iterations. In our case processors 1, 2, and 3 get iterations 10,5; 9,6; and 8,7, respectively. Since  $r > p$ , we compute  $r \bmod p = 4 \bmod 3 = 1$ , and thus, the first processor gets two iterations from the beginning, i.e., it gets iterations 1 and 2. The other two processors can pick up the two remaining iterations. So processor 2 and 3 get iterations 3 and 4, respectively. Figure 4b) pictorially represents our discussion above. Clearly, our schedule is optimal.

We will contrast our technique with another popular technique for load balancing. Often, iterations of heterogeneous loops are assigned in an interleaved fashion — using round-robin scheduling. For our example above, with interleaved scheduling, processor 1 gets its

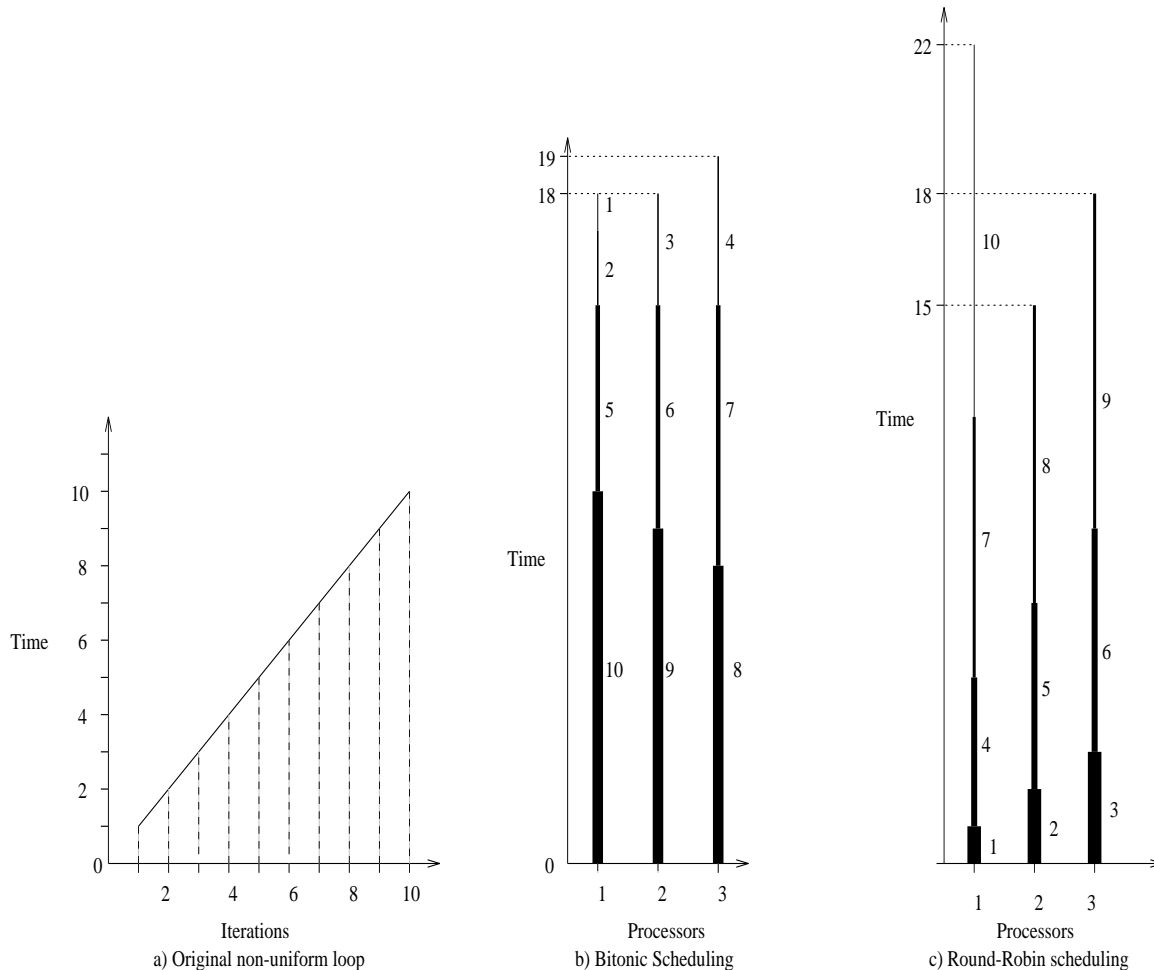


FIGURE 4. Bitonic scheduling vs. Round-robin scheduling

erations 1,4,7,10; processor 2 gets iterations 2,5,8; and processor 3 gets iterations 3,6,9. The completion time using our schedule, 19s, is shorter than the completion time using interleaving, 22s. Figure 4c) clearly shows that this strategy is non-optimal.

## 5.2. Heterogeneous Loops; With Communication

The case with communication can be handled with a slight modification of the above transformation. Every iteration of the new parallel loop will cause  $c(n+1) + 2d$  bytes to be sent. When  $2p$  divides  $n$ , the homogeneous loop obtained this way can be scheduled as described in Section 5.1.1. When  $2p$  does not divide  $n$ , we have to use an approach similar to the approach used in 5.1.2. We find  $r = n \bmod (2p)$ . We can perfectly schedule  $n - r$  iterations. And we choose the  $r$  iterations, such that they are the “cheapest” in terms of the imbalance they produce, and assign them as before. The difference is that, this time we don’t use the sign of  $a$  to determine which iterations are the “cheapest”. We have to use the sign of  $(\gamma a + \beta c)$  (recall that  $y = ci + d$  is the communication for iteration  $i$ ), because this constant

determines whether the time spent on computation and communication increases or decreases with the iteration number.

## 6. SCHEDULING FOR HETEROGENEOUS PROCESSORS

In this section we consider parallel machines with heterogeneous processors and a homogeneous network. The following machine parameters describe these types of machines:  $p$  (number of processors in the system),  $\gamma_i$  (time for processor  $i$  to execute one operation),  $\alpha$  (communication initialization time), and  $\beta$  (time to send one byte of a message). As usual,  $n$  denotes number of iterations of the loop.

We call the straightforward way of assigning the same amount of work to each processor the *architecture-oblivious* approach, and the algorithms developed in the following sections the *architecture-conscious* approach.

### 6.1. Homogeneous parallel loops, no communication

We create the schedules by trying to balance the computation on all the processors. We note that, to evenly

distribute computation, every processor should have a fraction of all the work given by the following formula:

$$w_i = \frac{\frac{1}{\gamma_i}}{\sum_{k=1}^p \frac{1}{\gamma_k}}$$

To get the optimal load balance, we should assign  $z_i = w_i n$  iterations to processor  $i$ . Since  $z_i$  is not necessarily an integer number, we have to decide whether  $\lfloor z_i \rfloor$  or  $\lceil z_i \rceil$  should be used. If the iteration space is large this decision is not very critical. We break the tie in the following way. Processor  $i$  works on iterations  $\lfloor \sum_{k=1}^{i-1} z_k \rfloor + 1$  through  $\lfloor \sum_{k=1}^i z_k \rfloor$ . The schedule obtained in this way is optimal. A similar approach, by distributing the load proportionally to the relative speeds of the processors, has been used with success in [11].

### 6.2. Homogeneous parallel loops, with communication

When there is communication, the algorithm in Section 6.1 will not necessarily generate an optimal schedule. Here we present an optimal solution. For the uniform case,  $x_i = x$  and  $y_i = y$  for  $i = 1, \dots, n$ . The communication time caused by  $z_i$  iterations is  $\alpha + \beta y z_i$  (recall that all objects to be sent are packed into one message and sent after the computation has completed). Hence, the total time spent by processor  $i$  on computation and communication is

$$t_i = \gamma_i x z_i + \alpha + \beta y z_i = g_i z_i + \alpha$$

where  $g_i = \gamma_i x + \beta y$ . Note that for this to work, we have to ensure that  $\gamma_i x$  and  $\beta y$  are in the same units, say, microseconds.

As in the other cases, our goal is to find a set of  $z_i$ 's that minimizes:  $\max_{i=1}^p t_i$ . If we assign a non zero amount of work to every processor, such a minimum will yield  $t_i = t_j$  for  $i, j = 1, \dots, p$ . A set of  $z_i$ 's that minimizes  $\max_{i=1}^p g_i z_i$  will also minimize  $\max_{i=1}^p t_i$ , because  $\alpha$  is constant. We can redefine  $w_i$  to be:

$$w_i = \frac{\frac{1}{g_i}}{\sum_{k=1}^p \frac{1}{g_k}}$$

and proceed as in Section 6.1.

### 6.3. Heterogeneous parallel loops

In this case, we can again first transform a heterogeneous loop into a homogeneous loop and then apply the methods described above. Note that, although this approach results in a schedule which is optimal for the transformed, homogeneous loop, it is not necessary optimal for the original, heterogeneous loop. The possible load imbalance is, however, very small. The work assigned to each processor is different from the optimum by at most one iteration.

## 7. SCHEDULING FOR HETEROGENEOUS MEMORY

In this section we examine ways of estimating the memory requirement for an application. We then extend our scheduling algorithms to account for heterogeneous memory.

### 7.1. Resident Memory Size (RMS)

Our experiments show that using the total memory requirement is generally not a good criterion for judging the largest problem size we can run efficiently.

TABLE 4. Effect of NPS & total memory

Data Size	Total Mem.	NPS		NPS+Tot Mem	
		Mem.	Time	Mem.	Time
1424	48.7M	28.0M	2091.5s	24.5M	2418.4s

Table 4 shows the results obtained for the Matrix Multiplication program on a configuration having a SPARC 5 and a SPARC LX machine. We first distributed the work among the two machines proportional to their NPS values, using the *architecture-conscious* technique from the last section. This distribution causes the total memory requirement for the SPARC 5 (28.0Mb, column 3) to exceed the user available memory for it (24.5Mb). We then redistributed the data among the processors so that we respect the memory constraint on the SPARC 5. But this caused an increase in the execution time (see columns 4 and 6). The reason is that the total memory requirement is a very conservative measure, and generally over-estimates the memory requirement of an application. We therefore introduce a new notion – the *Resident Memory Size (RMS)* for a given program segment, defined as the minimum number of pages of physical memory required to ensure that all page fault misses are cold misses (i.e., due to the first reference) for that segment, using a particular page replacement algorithm. We believe that this gives a better indication of the memory requirement for an application. Note that RMS is a program level analogue of the operating systems notion of *Working Set Size (WSS)* with an appropriate window size (WSS is defined as the set of pages in the most recent  $\Delta$  (window size) page references).

For a particular application, as we increase the data size we will reach a critical point beyond which the performance of the program degrades rapidly. This critical data size cannot simply be obtained from the total memory requirement for the application. Usually the RMS should be a good approximation of this critical point. For example consider the matrix multiplication program, which computes  $C = A * B$ , where  $A$ ,  $B$ , and  $C$  are  $N \times N$  matrices. The total memory requirement

for this program is  $3N^2$ . However, notice that all three matrices need not occupy the memory at the same time. If we compute the  $C$  matrix, a row at a time, we need to keep only one page of  $C$  and one row of  $A$  in memory, but we must have the whole of matrix  $B$  in memory. Therefore, if we calculate the resident memory size for MxM, we get the following, approximate, formula:

$$\text{RMS} = (N^2 + N) * \text{ElementSize}/\text{PageSize} + 1$$

The above RMS is calculated using an ideal page replacement scheme. Using the LRU (Least Recently Used) page replacement instead, would give

$$\text{RMS} = (N^2 + 2N)\text{ElementSize}/\text{PageSize} + 2$$

We observe that if the resident memory size is less than the user available memory then our program will not suffer from the effects of memory limitations. If, on the other hand, the program's RMS is larger than the available memory then some of the pages required will not be in memory, and we will have to take a page fault. As the input data size increases, the RMS increases, ultimately exceeding the available memory. If we attempt to run very large programs then we will cause the machines to *thrash*, severely degrading the performance.

We use a compile-time algorithm to approximate the RMS. We compute the number of pages contributed to RMS by every array reference in a loop nest. We first find the *stride vector* [5] for a given reference and then determine the outermost loop carrying reuse. For all loops enclosed by this loop we use strides and loop bounds to calculate the number of reused pages.

Let us illustrate the algorithm with an example. Consider the following loop nest from the matrix multiply program.

```

for i = 1 to n do
  for j = 1 to n do
    for k = 1 to n do
      c[i, j] += a[i, k] * b[k, j]
    
```

Assume row major mapping for all arrays. The stride vectors for references to arrays a, b, and c are:

$$v_a = \begin{pmatrix} n \\ 0 \\ 1 \end{pmatrix}, \quad v_b = \begin{pmatrix} 0 \\ 1 \\ n \end{pmatrix}, \quad v_c = \begin{pmatrix} n \\ 1 \\ 0 \end{pmatrix}$$

For a given reference a stride vector has one element for every loop enclosing this reference. An element of a stride vector is equal to the memory stride for consecutive iterations of the corresponding loop. In our example, the bottom element of  $v_a$  is 1, which means that the stride for accesses to array a in loop-k is unitary. The two other elements of  $v_a$  inform that the stride in loop-j is 0, and the stride in loop-i is  $n$ .

Stride vectors are used to describe the locality of memory accesses. Assume that a page holds  $p$  array elements and that  $1 < p < n$ . Consider the reference to

array a. We can see from the stride vector that there is temporal reuse carried by loop-j and spatial reuse carried by loop-k. The outermost loop carries no reuse.

For the reference to the array a, loop-j is the outermost loop with reuse. According to our algorithm, we consider all loops enclosed by the loop-j, that is the loop-k. This reference contributes  $\text{RMS}_a = \frac{1n}{p} = \frac{n}{p}$  pages, where 1 is the stride in loop-k and  $n$  is the number of iterations of that loop.

Similarly for the reference to array b, loop-i is the outermost loop carrying reuse, and we have to consider all loops enclosed by it, i.e., loop-j and loop-k. Each of those loops has  $n$  iterations and the strides are 1 and  $n$  respectively. The number of pages (ignoring boundary conditions) is  $\text{RMS}_b = n(\frac{n}{p})$ , that is the number of iterations of loop-j multiplied by the number of pages referenced in loop-k.

Calculation of the RMS for the reference to the array c is similar to  $\text{RMS}_a$ . This time the stride in the innermost loop is 0. Hence,  $\text{RMS}_c = \frac{0n}{p} = 0$ . Because we need at least one page to keep the current element of c in memory, we take  $\text{RMS}_c = 1$ .

The resident memory size for all three arrays in this example is  $\text{RMS} = \text{RMS}_a + \text{RMS}_b + \text{RMS}_c$ . Hence,

$$\text{RMS} = \frac{n}{p} + \frac{n^2}{p} + 1 = \frac{n^2 + n}{p} + 1$$

The result is the same as the formula shown earlier in this section for an ideal page replacement algorithm.

The limitation of the above algorithm is that it is very conservative. While the RMS value obtained for regular problems should work well in practice, it may not be a good approximation for irregular problems.

## 7.2. Combined effect of processor and memory heterogeneity

In this section we point out how to efficiently run large problem instances on a particular configuration of the NOW. We look at the interaction of the *normalized processor speed* and the *resident memory size*, both of which are application dependent, and show their combined effect on scheduling.

Deciding on the largest problem instance to be solved is a subtle issue. It depends on a number of criteria, such as how long are we willing to wait? or what measure of efficiency do we desire?, etc. In this section, we will not deal with the problem of finding out the largest problem instance to solve. Instead, we will look at how we might achieve good performance, i.e., minimal execution time, for program instances where the RMS value exceeds the memory available to a user application on at least one processor in the NOW.

Table 5 shows the results obtained for MxM(2788 × 2788) on a configuration of SPARC 10 and SPARC 5 workstation. The SPARC 5 has approximately four times less memory than SPARC 10 (table 3). We first ran the program by distributing the work based on the

TABLE 5. Effect of NPS &amp; RMS

Total Mem.	Total RMS	Mem SPARC5	NPS		MEM		NPS+RMS	
			RMS	time	RMS	time	RMS	time
186.6Mb	62.3Mb	24.5Mb	28.8Mb	"∞"	12.5Mb	19902s	24.5Mb	14477s

NPS values, but the RMS (28.8Mb) exceeded the memory on the SPARC 5 (24.5Mb), and caused the machine to thrash. We had to stop the execution. We then distributed the data so that the RMS on SPARC 5 was equal to the available memory (see under NPS+RMS). We also used the memory ratio of the machines to schedule the work (see under MEM), however this results in a load imbalance as more work is assigned to the SPARC 10, and thus it takes a longer time to complete. We can clearly see that the execution time obtained by using both the NPS and RMS values is the best, while using just the NPS values we could not even run on the chosen data size.

### 7.2.1. Scheduling algorithm

We first try to distribute the data among the processors proportional to their NPS values for the particular application in consideration, using the algorithms from the previous sections. We also calculate the RMS value for the program. Using this RMS value and the user available memory we determine whether we exceed the memory on any processor, and redistribute the excess amount among the other processors by recursively applying the same technique. The schedule obtained in this way tries to respect the processor speed ratios, and even when memory becomes a factor, it tries to be as close to the processor speed ratios as possible, while satisfying the memory constraints. This approach should give near-optimal performance for a given data size.

## 8. SCHEDULING FOR HETEROGENEOUS NETWORKS

In this section we consider parallel machines with heterogeneous processors and a heterogeneous network. The following machine parameters describe these types of machines:  $p$  (number of processors in the system),  $\gamma_i$  (time for processor  $i$  to execute one operation),  $\alpha_i$  (communication initialization time on processor  $i$ ), and  $\beta_i$  (time to send one byte of a message from processor  $i$ ). As usual,  $n$  denotes the number of iterations of the loop. Loops without communication are equivalent to the loops for systems with a homogeneous network. Therefore, for these, the solutions from Sections 6.1 and 6.3 can be applied to the case of a heterogeneous network.

### 8.1. Homogeneous parallel loops, with communication

The time spent by processor  $i$  on computation and communication can be calculated in the same way as in the homogeneous case. That is, we will define the time to execute one iteration of the loop on processor  $i$ ,  $g_i = \gamma_i x + \beta_i y$ , and note that time spent by processor  $i$  on computation and communication is:

$$t_i = g_i z_i + \alpha_i$$

To eliminate load imbalance caused by different communication startup times,  $\alpha_i$ , we find a processor with the largest value of  $\alpha_i$ , and we add extra iterations to processors with shorter times.

Let  $\alpha_j = \max_{i=1}^p \alpha_i$ . The number of extra iterations for a given processor is:

$$e_i = \left\lfloor \frac{\alpha_j - \alpha_i}{g_i} \right\rfloor$$

We can now use the algorithm from Section 6.2 on the remaining  $(n - \sum_{k=1}^p e_k)$  iterations to obtain  $z'_i$  and assign  $z_i = z'_i + e_i$  iterations to every processor.

The solution presented in this section is not necessarily optimal, but the schedule found by this algorithm is very close to the optimum. The work allocated to any processor is different by at most two iterations from the work corresponding to the perfect load balance.

### 8.2. Heterogeneous parallel loops, with communication

To schedule a heterogeneous loop with communication, we again transform it into a homogeneous parallel loop. Transforming the loop first and then applying the algorithm from Section 8.1, would cause the use of iterations of the transformed loop as extra iterations to balance the communication initialization cost. However, those iterations have a higher cost, both in terms of computation and communication, than any single iteration of the original loop. It is, therefore, desirable to eliminate the initialization imbalance first, and to then transform the remaining iterations into a homogeneous parallel loop.

Because differences between the parameters  $\alpha_i$  may be small in some cases, we want to use the iterations with the smallest cost possible. Note that by cost, in this context, we mean only the time contributed by an

iteration, without the communication startup cost. The cost of an iteration  $i$  on a processor  $j$  is given by:

$$g_{i,j} = \gamma_j x_i + \beta_j y_i = (\gamma_j a + \beta_j c)i + \gamma_j b + \beta_j d,$$

since  $x_i = ai + b$ , and  $y_i = ci + d$

To ensure the use of the shortest possible iterations, we should use iterations from the beginning of the iteration space if  $(\gamma_j a + \beta_j c) > 0$ , since in this case  $g_{i,j}$  is an increasing function of  $i$ , and from the end of the iteration space otherwise. The sign of  $(\gamma_j a + \beta_j c)$  is machine dependent, so for some processors, we should allocate iterations from the beginning of the iteration space, but for other processors, from the end. This general case can be handled by an extended version of our algorithm. In practice, however, constants  $a$  and  $c$  have the same sign, which implies the same sign for  $(\gamma_j a + \beta_j c)$  no matter what the machine parameters are ( $\gamma_j$  and  $\beta_j$  are always positive). Therefore, we will describe here a solution for this simpler case only.

Let  $\alpha_j = \max_{i=1}^p \alpha_i$ . We will start assigning extra iterations from iteration 1 upwards if  $(\gamma_j a + \beta_j c) > 0$  and from iteration  $n$  downwards otherwise. Since the two cases are very similar, we will describe the first one only.

To simplify the notation, let us introduce:

$$E_i = \sum_{k=1}^i e_k$$

We will compute  $e_i$  in order:  $e_1, e_2, \dots, e_p$ . For a given processor  $i$ , we choose the maximum  $e_i$ , such that

$$\sum_{k=E_{i-1}+1}^{E_i} (g_{k,i}) = \sum_{k=E_{i-1}+1}^{E_i} (\gamma_i x_k + \beta_i y_k)$$

does not exceed  $(\alpha_j - \alpha_i)$ . That is, the time taken to execute the extra iterations,  $k$ , on processor  $i$ , is less than or equal to the initialization imbalance for that processor.

The next step transforms the parallel loop from  $E_p$  to  $n$  into a homogeneous loop. In effect every processor has two sets of iterations to execute:

- iterations  $E_{i-1} + 1, \dots, E_i$  of the original loop, and
- iterations  $[\sum_{k=1}^{i-1} z_k] + 1, \dots, [\sum_{k=1}^i z_k]$  from the transformed loop.

As in the previous section, the schedule found here is sub-optimal, although very close to the perfect balance. The difference for any processor between this schedule and the perfect balance again does not exceed two iterations.

## 9. SCHEDULING FOR CONTENTION AVOIDANCE

Sections 5, 6 and 8 considered a machine model which allowed messages sent from different machines to travel

in the network at the same time — in parallel. On many existing parallel machines, for instance on a network of workstations using Ethernet as the interconnect, the performance will suffer if many messages are being sent at the same time. On such parallel multicomputers, it is desirable to schedule a parallel program in such a way that only one processor (workstation) sends a message at a given time.

We assume that the machines effectively sequentialize all messages. That is, at any given time only one message can be in transit in the physical medium, which is not accessible to the other processors until the send operation is completed. This model should be a good approximation of many bus-based multicomputers.

Programs running on machines that sequentialize communication, need a different set of optimizations. In this section we will describe a method to minimize execution time of a homogeneous parallel loop on a homogeneous multicomputer.

### 9.1. The Model

We will extend the machine model discussed in the previous sections. The following parameters describe every processor in the parallel machines considered in this section:

- $\gamma$  — time to execute one operation,
- $\alpha', \beta'$  — communication parameters for the part of the send operation performed locally (this is the part of communication that is not sequentialized),
- $\alpha'', \beta''$  — communication parameters for the part of the send operation that requires access to a shared physical medium and which is sequentialized.

As before a homogeneous parallel loop is described by the following two parameters:  $x$  (number of operations to be performed in one iteration), and  $y$  (length of the message caused by a single iteration, but as in the earlier sections messages from all iterations assigned to a given processor are combined and sent as one larger message).

There are  $p$  processors. Processor  $i$  works on  $z_i$  iterations. If we assume that the message can be sent immediately with no contention (without waiting for other processors to free the shared communication medium), the total time to execute  $z_i$  iterations and broadcast a message resulting from this iteration is:

$$T_i' = z_i x \gamma + \alpha' + z_i y \beta' + \alpha'' + z_i y \beta'' = t_i' + t_i''$$

where  $t_i' = z_i x \gamma + \alpha' + z_i y \beta'$ , is the work that can be performed locally without interference with other processors, and  $t_i'' = \alpha'' + z_i y \beta''$  is the part of communication that has to be sequentialized.

In reality, every processor first performs local operations for  $t_i'$  time, and then waits until the shared medium becomes free and sends its data in  $t_i''$  time. During this  $t_i''$  period, other processors cannot send anything.

Without loss of generality, assume that processor  $i$  broadcasts its message before processor  $i + 1$ . This is justified, because all processors are identical and their ordering is arbitrary. With this assumption, we can give the real time that the processor  $i$  spends on computation and communication:

$$T_i = \max(t'_i, T_{i-1}) + t''_i$$

where  $T_0 = 0$ . This formula expresses a simple fact that a processor cannot begin accessing the shared medium before its computation has completed ( $t'_i$ ), or its predecessor has released the communication channel ( $T_{i-1}$ ).

## 9.2. Optimal Schedule

A simple-minded strategy would assign the same number of iterations to every processor — all processors have the same speed and all iterations have the same cost. This strategy would cause each processor, except the first one, to wait for the communication channel. Moreover, every processor would wait longer than its predecessor. If we define the total execution time to be the time when the last send completes, the execution time achieved by this strategy is not optimal. This fact is illustrated in Figure 5b).

We show a static scheduling strategy that is optimal in that it results in the shortest possible execution time on a given number of processors (that is, all available processors are used).

**THEOREM 9.1.** The shortest execution time is achieved when  $t'_i = T_{i-1}$ , for  $i = 2, \dots, p$ .

**Proof (sketch):** The total time spent on the sequentialized part of communication is the same for every schedule and is equal to:  $\sum_{k=1}^p t''_k = p\alpha'' + ny\beta''$ .

Consider a schedule such that  $t'_i = T_{i-1}$ , for  $i = 2, \dots, p$ . Let us call it a contention-free schedule. We will show that any change in this schedule will increase the execution time.

Consider a new schedule, in which processor  $i$  broadcasts its message before processor  $i + 1$  (this can be assumed without loss of generality, because all processors are identical). Let  $i$  be the first processor whose local time  $t'_i$  differs from the local time under the contention-free schedule.

Note that the local time  $t'_i$  and the communication time  $t''_i$  are related and any change in the number of iterations assigned to  $i$  will change both times for processor  $i$ . There are two cases:

(1) The local time under the new schedule is longer than the local time under the contention-free schedule.

The sum of the remaining communication times (including processor  $i$ ),  $\sum_{k=i}^p t''_k$ , is the same as in the contention-free schedule. In the contention-free schedule this was also the time left to the completion of the execution. In the new schedule, because  $t'_i$  has increased, we have to start communication for processor  $i$  later than in the contention-free schedule. The total

time to complete is at least the same as that for the contention-free schedule so the execution time will be longer.

(2) The new local time is shorter.

We are left with some extra work that processor  $i$  did in the contention-free schedule. If all processors  $j$ , such that  $j > i$ , have the same amount of work as they used to have under the contention-free schedule, this extra work will be left over. Hence, one of the remaining processors has to perform this additional work increasing the execution time.

We will show below, an algorithm that will find a contention-free schedule if it exists. We can use Theorem 9.1 to simplify the formula for the completion time of the  $i$ th processor:  $T_i = t'_i + t''_i$ . We can use this formula to find the optimal schedule. A schedule is defined by the set of  $z_i$ , for  $i = 1, \dots, p$ . By Theorem 9.1 we have  $t'_i = T_{i-1}$ , but we know that  $T_{i-1} = t'_{i-1} + t''_{i-1}$ , so this equality can be rewritten as:  $t'_i = t'_{i-1} + t''_{i-1}$ . If we expand this formula, we get:

$$z_i x\gamma + \alpha' + z_i y\beta' = z_{i-1} x\gamma + \alpha' + z_{i-1} y\beta' + \alpha'' + z_{i-1} y\beta'' \quad \text{or}$$

$$wz_i = vz_{i-1} + \alpha''$$

where,  $w = x\gamma + y\beta'$ ,  $v = x\gamma + y\beta' + y\beta''$ . We have  $p - 1$  of these equations for  $i = 2, \dots, p$ . These  $p - 1$  equations together with the “exhaustiveness” equation,

$$\sum_{i=1}^p z_i = n$$

constitute a system of  $p$  linear equations with  $p$  unknowns.

$$\begin{aligned} vz_1 - wz_2 &= -\alpha'' \\ vz_2 - wz_3 &= -\alpha'' \\ &\vdots \\ vz_{p-1} - wz_p &= -\alpha'' \\ z_1 + z_2 + \dots + z_p &= n \end{aligned} \quad (1)$$

The solution to this system of equations defines the optimal schedule.

## 9.3. Validity of the Solution

We will now show that the above system of equations (1) has a unique solution. The coefficient matrix of this system of equations is:

$$M_p = \begin{bmatrix} v & -w & 0 & \dots & 0 \\ 0 & v & -w & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & -w \\ 1 & 1 & 1 & \dots & 1 \end{bmatrix}$$

$M_p$  is a  $p \times p$  matrix. Its determinant is equal to:

$$\det M_p = v \det M_{p-1} + w^{p-1}$$

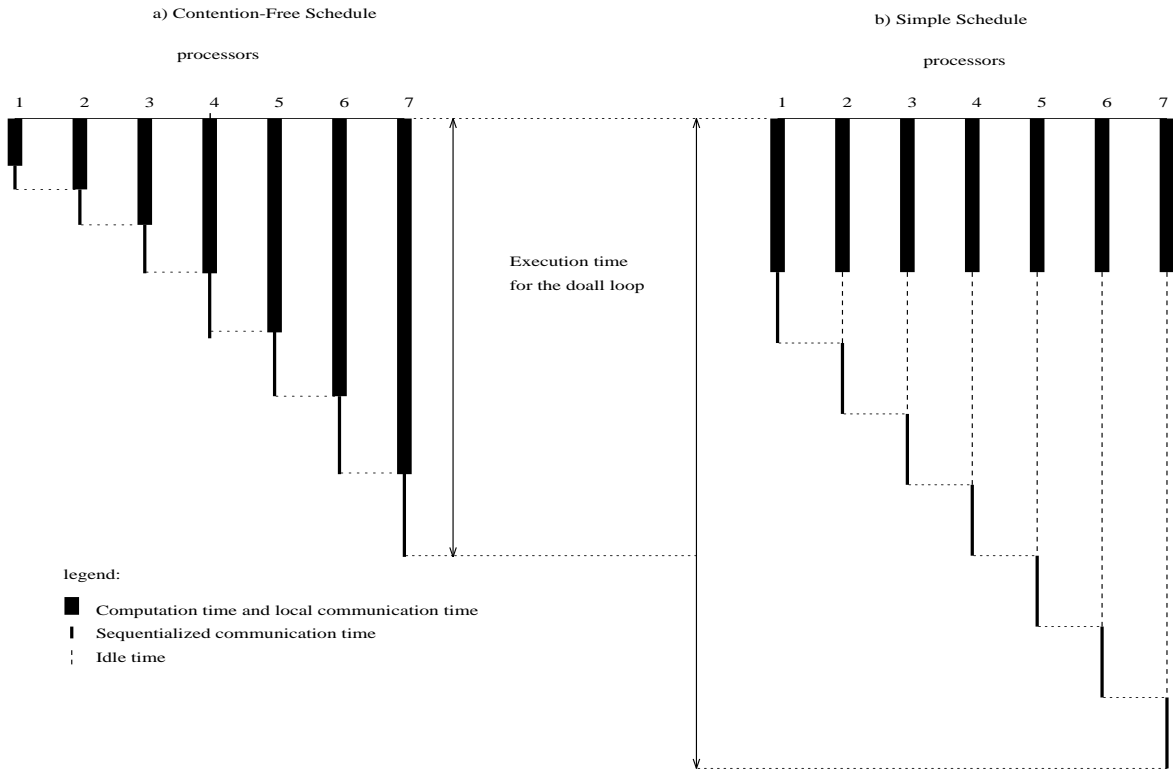


FIGURE 5. Contention-free Schedule vs. Simple Schedule

where  $M_{p-1}$  is a  $(p-1) \times (p-1)$  matrix of the same structure as  $M_p$ . The matrix  $M_{p-1}$  can be obtained from  $M_p$  by deleting the first row and the first column. Because  $v, w, \det M_1 > 0$ , by the recursive nature of this equation,  $\det M_p > 0$  for  $p = 1, 2, \dots$ . Therefore, this system of equations has a unique solution.

Note that we have not shown that the solution is always valid — a solution may contain negative  $z_i$ 's. This corresponds to a set of parameters with a very high relative startup communication cost. However, if a contention-free schedule exists, the solution to the above system of equations will describe this optimal schedule.

In practice the solution is always positive. Let us consider an example with two processors:

$$\begin{cases} vz_1 - wz_2 = -\alpha'' \\ z_1 + z_2 = n \end{cases}$$

We have

$$(v+w)z_1 = wn - \alpha''$$

hence  $z_1$  is negative if and only if  $\alpha'' > wn = (x\gamma + y\beta')n$ . This condition would be true if the startup time for a send,  $\alpha''$ , was longer than all the computation in the loop,  $(x\gamma n)$ . Clearly, we do not want to parallelize a loop like that in the first place.

It is also worth noting that, for a given solution, if  $z_1 > 0$ , then  $z_j > 0$  for  $j > i$ . This is true, because  $wz_i = vz_{i-1} + \alpha''$ , that is, if  $z_{i-1}$  is positive, then  $z_i$  is positive too.

## 10. EXPERIMENTAL EVALUATION

To verify the proposed scheduling techniques, we conducted experiments and measured the execution time and the speedup of several applications. Where appropriate, we also compare our approach with straightforward scheduling. The results of our experiments are encouraging. Our techniques show significant performance improvements over traditional approaches.

The rest of this section is organized similarly to the whole paper. First we compare our approach to scheduling heterogeneous loops on homogeneous processors with the popular round-robin load-balancing technique. We did not run experiments for homogeneous loops on homogeneous processors, as scheduling those is easy and well understood. Then we give results for scheduling both homogeneous and heterogeneous loops on heterogeneous processors. The last part of this section gives results for our approach to contention avoidance. The experiments for calculating the NPS, and for scheduling in the presence of memory heterogeneity were presented in sections 4 and 7.2, respectively.

All our experiments were performed on a network of Sun workstations (SPARC 1, LX, 5 and 10), interconnected via an Ethernet LAN. PVM (Parallel Virtual Machine) [10], a message passing software system mainly intended for network based distributed computing, was used to parallelize the applications. The latency obtained with PVM is approx. 2414.5  $\mu s$ , and bandwidth is approx. 0.96 Mbytes/s. We assume that

there is no external load on the processors or network, i.e., the NOW is used in a dedicated user mode.

### 10.1. Applications

The applications used for our experiments are:

- *Matrix Multiply (MXM)*: Multiplication of two square matrices.
- *2D-FFT*: Two dimensional Fast Fourier Transformation.
- *Cholesky Factorization (CHO)*: Find a lower triangular matrix  $L$  with positive diagonal elements such that  $A = LL^T$ , where  $A$  is a dense symmetric positive definite matrix.
- *Spatial Price Equilibrium Modeling (ECO)*, a commodity trade model [17]: For a set of supply and demand markets with given tariffs, transportation costs, supply and demand price functions, this program finds the amount of goods shipped between different markets.
- *TRIANG*: This is a synthetic program [7], which has a loop nest with varying computation in each iteration of the outermost loop.
- *Livermore Fortran Kernel (LFK)*: modified loop 10 from Livermore Fortran Kernels [7].

### 10.2. Scheduling for Heterogeneous programs

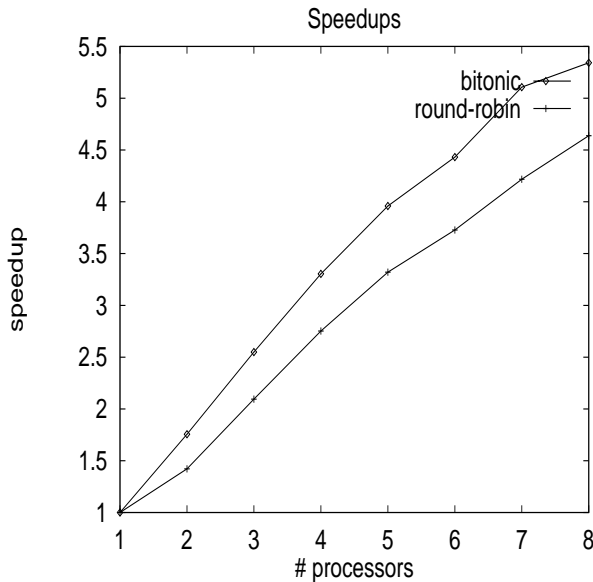


FIGURE 6. Speedups — TRIANG

TRIANG [6] is a program with a heterogeneous loop used in the experiments presented in this section. Figure 6 shows the speedups for two different parallelization of TRIANG. Label bitonic marks the results for the parallelization from Section 5. We compare our approach with the round-robin scheduling. The round-robin technique schedules a doall loop on  $p$  processors by assigning iterations  $0, 0 + p, 0 + 2p, \dots$  to processor

$0$ , iterations  $1, 1 + p, 1 + 2p, \dots$  to processor 1 and so on. This approach is very popular in practice. It is very simple and yields acceptable performance.

In this experiment, we assume that the arrays are not distributed before and after the loop nest. Therefore, our timings include the time required to send out necessary data to all processors and to gather results from all participating processors. Because communication in a network of workstations is very expensive, the speedups are not close to the optimum. Figure 6 demonstrates that the bitonic schedule consistently outperforms the round-robin technique.

### 10.3. Scheduling for Heterogeneous Processors

We have chosen three applications to measure performance of scheduling in a heterogeneous environment. Matrix multiply and economics are examples of a homogeneous loop. TRIANG is an example of a heterogeneous loop.

#### 10.3.1. Homogeneous loops, no communication

To find a static schedule on a network of heterogeneous computers, we use the normalized processor speeds, which are shown in Table 6, for the MXM program.

TABLE 6. Normalized speeds for matrix multiply.

machine type	speed
SPARCstation 1( $T_1$ )	1.00
SPARCstation LX( $T_2$ )	1.85
SPARCstation 10( $T_3$ )	3.00

We can use normalized speeds to compute a “speedup” for a heterogeneous machine configuration. We can define this generalized speedup to be a ratio of the uniprocessor execution time on the base processor to the execution time of the parallel program. We can also define the “ideal” speedup for a particular configuration to be the sum of normalized speeds of all processors in a given configuration.

The results for matrix multiply are given in Figure 7 and Table 7. The program multiplies two square matrices of the size  $600 \times 600$ . The configuration column describes how many machines of a given type were used in the experiment. Type 1 ( $T_1$ ) is SPARCstation 1, type 2 ( $T_2$ ) is SPARCstation LX, and type 3 ( $T_3$ ) is SPARCstation 10. Architecture-oblivious schedule assigns the same number of iterations to every processor. Architecture-conscious schedule assigns a number proportional to the processor speed.

As expected, the results show that the architecture-conscious schedule is always better than the architecture-oblivious one. For some configurations the difference is not significant, for others it is very large. Intuitively, the slowest machine’s execution time will

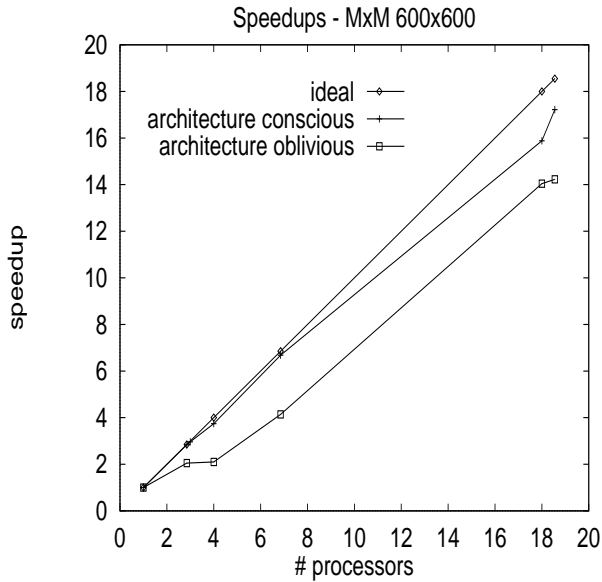


FIGURE 7. Speedups — matrix multiply

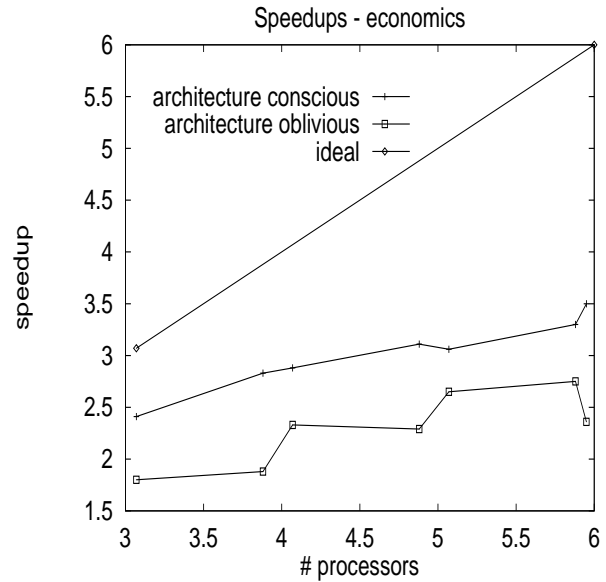


FIGURE 8. Speedups — economics

TABLE 7. Speedups — matrix multiply

Configuration	ideal	arch- -conscious	arch- -oblivious
$1T_1, 1T_2$	2.85	2.84	2.00
$1T_1, 1T_3$	4.00	3.74	2.05
$2T_1, 1T_2, 1T_3$	6.85	6.68	4.03
$1T_1, 3T_2, 2T_3$	12.55	10.26	6.12
$1T_1, 8T_2, 1T_3$	18.8	18.75	10.06
$10T_1, 3T_2, 1T_3$	18.55	17.22	13.92
$2T_1, 12T_2, 1T_3$	26.2	23.09	15.40
$15T_1, 1T_3$	18.00	15.88	13.73

dominate the time for the whole program. So, the configuration with many fast machines and few slow ones will suffer most from architecture-oblivious scheduling. If, on the other hand, a configuration contains mostly slow machines, architecture-conscious scheduling will not improve the execution time significantly.

There is one more interpretation for the sum of normalized speeds. It says how many base processors would be equivalent in speed to a particular configuration. Note that this number can be fractional, so for example the first configuration in Table 7 is equivalent to 2.85 base processors. We can use this observation to plot the speedup as a function of the number of processors. This approach gives a concise visualization of the parallel performance, but we shouldn't overestimate its accuracy. In particular, there may be many different configurations with the same base processor equivalent, but their speedups may be different.

### 10.3.2. Homogeneous loops, with communication

The second example of the homogeneous loop case is a program for spatial price equilibrium modeling in economics, the ECO [17] application. This program has a set of parallel loops. However, parallelization of the program on a network of workstations is a non-trivial task, since communication is required across the loops and data has to be broadcast between the loops. Figure 8 shows the speedups obtained on a variety of heterogeneous configurations of machines. The architecture-conscious schedules consistently outperform the architecture-oblivious schedules. In spite of the large amount of communication in the program and the high cost of network communication, a satisfactory parallel performance was achieved.

### 10.3.3. Heterogeneous parallel loops

For the heterogeneous loop case, we parallelized the TRIANG program. We can verify the performance of the parallelized code by plotting the normalized speedups and speedups for homogeneous configurations that consist of base processors. We can see that for bitonic scheduling (Figures 9) the performance of the heterogeneous parallelization is close to the homogeneous case.

The comparison with the homogeneous case is an interesting metric. Since in most cases, the architecture-conscious heterogeneous scheduling would be much better than the naive architecture-oblivious scheduling, the architecture-conscious homogeneous case provides an upper bound of how well a heterogeneous solution can perform.

## 10.4. Scheduling for Contention Avoidance

We show experimental results for contention avoidance scheduling on the LFK program. The outermost loop

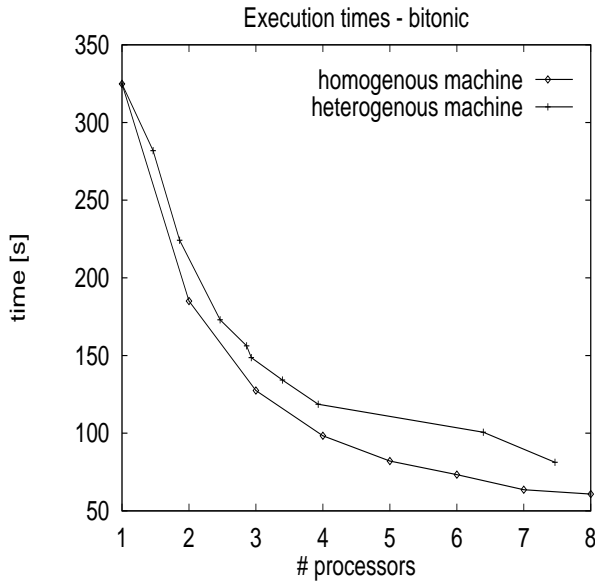


FIGURE 9. Execution times — TRIANG, bitonic scheduling

is a doall loop and it is being parallelized. We assume, however, that for the next stage of computation array PX must be broadcast to all processors. This causes high level of contention in our Ethernet network. We can use the algorithm developed in Section 9 to maximize the speedup by minimizing contention.

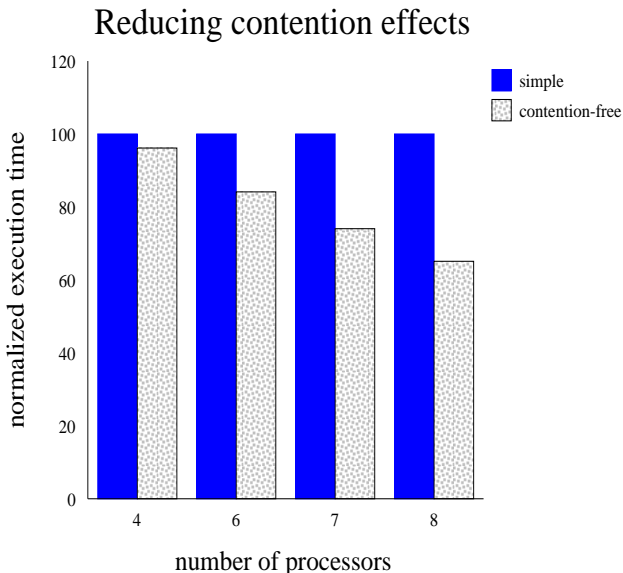


FIGURE 10. Execution times — LFK 10

Figure 10 shows the performance of two parallelizations of the modified LFK 10 nest. We can see that for a small number of processors contention is not a very big problem. But as the number of processors increases, performance of the simple parallelization deteriorates very quickly. The architecture-conscious schedule results in a significantly faster program.

For this example the speedups achieved by the architecture-conscious schedule are not very good, which is generally true about programs with excessive communication. We may expect that if a program exhibits contention, the technique presented in Section 9 will improve its performance, but the speedup will always be significantly worse than the optimum. The reasons for using this technique, even though it is inherently suboptimal, are:

- We get a better performance than the sequential program. If we need high performance at any cost, we may choose to parallelize such a code even if we know that the machine will be underutilized.
- Most real applications have many phases in the program. If most of phases can be parallelized, then it is better for data to remain distributed. Therefore, parallelization of code fragments that do not display great parallelism/speedups is still necessary to maintain data locality and reduce communication in the later phases, since data would have to be on a single node if the code fragment were sequentialized.

## 11. CONCLUSIONS

In this paper, we looked at the general issues in heterogeneous computing, and we studied the problem of scheduling parallel loops at compile-time for a heterogeneous network of machines.

We proposed a simple yet comprehensive model for a network of processors. To model heterogeneous processors, we introduced the parameter, *normalized processor speed*, which is highly application dependent. To solve large problem instances, we need an estimate of the memory requirement of an application. We proposed a new estimate of this requirement, the *resident memory size*. Finally, we developed compiler algorithms for generating *optimal* and *near-optimal* schedules of loops for load balancing, communication optimizations and network contention, in the presence of program, processor, memory and network heterogeneity. Our experiments showed that the new techniques can significantly improve the performance of parallel loops over existing techniques.

## ACKNOWLEDGEMENTS

This work was supported in part by an NSF Research Initiation Award and ARPA contract F19628-94-C-0057.

## REFERENCES

- [1] J.N.C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: parallel programming in a heterogeneous multi-user environment. CMU-CS-95-137 30786, Carnegie Mellon Univ - Sch. of Computer Science, April 1995.

- [2] R. D. Blumofe and D. S. Park. Scheduling large-scale parallel computations on network of workstations. *3rd IEEE Intl. Symposium on High-Performance Distributed Computing*, april 1994.
- [3] S. H. Bokhari. *Assignment problems in parallel and distributed computing*. Kluwer Academic Publishers, 1987.
- [4] A. L. Cheung and A. P. Reeves. High performance computing on a cluster of workstations. *1st IEEE Int. Symposium on High Performance Distributed Computing*, pages 152–160, September 1992.
- [5] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of PLDI '95*, June 1995.
- [6] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. Technical Report 540, Computer Science Dept., Univ. of Rochester, October 1994.
- [7] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *4th IEEE Int. Symposium on High Performance Distributed Computing*, August 1995.
- [8] P. E. Crandall and M. J. Quinn. A decomposition advisory system for heterogeneous data-parallel processing. *3rd IEEE Int. Symposium on High Performance Distributed Computing*, August 1994.
- [9] R. F. Freund. Optimal selection theory for superconcurrency. In *Supercomputing '89*, November 1989.
- [10] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge, Tennessee, May 1993.
- [11] A. S. Grimshaw, J. B. Weissman, E. A. West, and E. C. Loyot. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *J. Parallel and Distributed Computing*, 21(3):257–270, 1994.
- [12] L. Kipp. Perfect Benchmarks Documentation, Suite 1 Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, October, 1993.
- [13] C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Trans. on Software Engineering*, 11:1001–16, October 1985.
- [14] W. Li and K. Pingali. Access normalization: Loop restructuring for NUMA compilers. *ACM Trans. on Computer Systems*, 11(4):353–375, November 1993.
- [15] F.C.H. Lin and R.M. Keller. The gradient model load balancing method. *IEEE Trans. on Software Engineering*, SE-13:32–38, jan 1987.
- [16] E.P. Markatos and T.J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 5(4), April 1994.
- [17] A. Nagurney, C. F. Nicholson, and P. M. Bishop. Spatial price equilibrium models with discriminatory ad valorem tariffs: formulation and comparative computation using variational inequalities. In J.C.J.M. van den Bergh, P. Nijkamp, and P. Rietveld, editors, *Recent Advances in Spatial Equilibrium Modeling: Methodology and Applications*. Springer-Verlag, Heidelberg, 1995.
- [18] N. Nedeljkovic and M. J. Quinn. Data-parallel programming on a network of heterogeneous workstations. *1st IEEE Int. Symposium on High Performance Distributed Computing*, pages 152–160, sep 1992.
- [19] M. Parashar, S. Hariri, A. G. Mohamed, and G. C. Fox. A requirement analysis for high performance distributed computing over LAN's. *1st IEEE Int. Symposium on High Performance Distributed Computing*, pages 142–151, September 1992.
- [20] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.
- [21] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Trans. on Computers*, C-36(12):1425–1439, December 1987.
- [22] V. A. Saletore, J. Jacob, and M. Padala. Parallel computations on the charm heterogeneous workstation cluster. *3rd IEEE Int. Symposium on High-Performance Distributed Computing*, april 1994.
- [23] B.S. Siegell. Automatic generation of parallel programs with dynamic load balancing for a network of workstations. CMU-CS-95-168 30880, Carnegie Mellon Univ. - Sch. of Computer Science, May 1995.
- [24] P. Tang and P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Intl. Conference On Parallel Processing*, August 1986.
- [25] Ko-Yang Wang. Precise compile-time performance prediction for superscalar-based computers. In *Proceedings of PLDI*, June 1994.
- [26] M. J. Zaki, W. Li, and M. Cierniak. Performance impact of processor and memory heterogeneity in a network of machines. In *4th Heterogeneous Computing Workshop*, April 1995.
- [27] M. J. Zaki, W. Li, and S. Parthasarathy. Customized dynamic load balancing in a heterogeneous network of workstations. In *5th IEEE Int. Symposium on High Performance Distributed Computing (also CSTR-602, CS Dept., Univ. of Rochester)*, August 1996.
- [28] M. J. Zaki, W. Li, and S. Parthasarathy. Customized dynamic load balancing in a heterogeneous network of workstations. To appear in *Journal of Parallel and Distributed Computing*, late 1997.