Performance Impact of Processor and Memory Heterogeneity in a Network of Machines *

Mohammed Javeed Zaki, Wei Li, Michał Cierniak
Department of Computer Science
University of Rochester

Abstract

In this paper we present a summary of the issues that need to be dealt with to effectively use a heterogeneous network of machines. We specifically point out the impact of processor and memory heterogeneity on the performance of parallel programs. Our results indicate that processor speeds are application dependent and we show how to use this information along with memory constraints for efficient scheduling.

1 Introduction

Network based distributed computing has attracted a lot of attention lately, due to the recent advances in high speed networks 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.

A network of machines (NOM) can have heterogeneity at various levels [3]. More specifically, the NOM may be heterogeneous in the processors, with processors of different speeds; the network, with varying cost of communication among pairs of processors; and the memory, with different amount of available memory on different machines. There is a fourth aspect, which is at the parallel program level, i.e., the program has parallel loops that have varying amount of work in each iteration.

A lot of current research on heterogeneous computing is directed at the use of different types of parallel processors, processing components, or connectivity paradigms to maximize performance, cost-effectiveness, and/or development effort. For example, such research efforts include the matching of individual code segments to best-suited machines. This involves identifying the optimal processor(s) for each task in a heterogeneous application, i.e., finding a set of machines to which the different tasks can be assigned, so as to minimize the overall execution time. Generalized optimal selection Theory (GOST) [11], Synthesis-of-Systems (SOS) theory [4], and strategies based on heuristics [8] are among the proposed solutions to this problem.

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. When dealing with this model in a heterogeneous NOM environment, the problem is not that of efficient mapping of subtasks to appropriate machines suitable for that particular subtask. Since all the tasks are similar in the SPMD model, the issue concerns 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 analyze the above problem, and to point out issues that must be taken into account for efficient computation on a heterogeneous NOM.

In particular, we study the performance impact of heterogeneous processors and memory. We show by experiments that the conventional ways of measuring processor speed and memory capacity are insufficient for a heterogeneous network of machines. 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.

The rest of the paper is organized as follows: We discuss the general problem of computing in heteroge-

*This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

neous environments in Section 2. Section 3 describes our research methodology. We look at the effect of heterogeneous processors in Section 4, and of heterogeneous memory in Section 5. Section 6 deals with the combined effect of both processor and memory heterogeneity on scheduling. This is followed by the conclusion in Section 7.

2 Issues in Heterogeneous Computing

In this section we present the issues that arise when computing on a heterogeneous network of machines.

- **Program Domain**: The parallel applications fall into a number of categories. In the SPMD model all the processors apply the same algorithm to different data points. These programs may have regular or irregular computation and communication. Many problems are composed of several subtasks with different properties. These subtasks may be parallelized individually, and may need to be done on specific machines for efficient computation. Other characteristics of the parallel program must be considered to decide whether any performance improvement can be obtained on NOMs. For example, if the communication to computation ratio is large, then it might not be worthwhile to parallelize the application. These factors become critical while considering NOMs due to the other reasons cited below.

- **Machine Heterogeneity**: A heterogeneous system may consists of various shared and distributed memory MIMD machines, SIMD and vector machines, and sequential workstations interconnected by a network. For example, we may have a multiprocessor SGI and Sun machine, and many HP, DEC, SGI and Sun workstations connected by a network. The network can be heterogeneous too, with different machines on a department-wide, campus-wide, or nation-wide network. There may also be several networks within a department.

- **Network Latency**: Network latency is one of the primary concerns for a heterogeneous NOM. High latency can make communication extremely expensive. As the number of machines in the network increases, high latency would restrict the scalability of the system, rendering all the parallelism and computing power ineffectual. It is crucial that the latency be reduced by developing high speed networks. As processors become faster, latency will continue to be a bottleneck.

- **Network Bandwidth**: Bandwidth is also a bottleneck, especially for Ethernet LAN. Although it is easier to increase the physical bandwidth (e.g., ATM promises a much higher bandwidth), rather than the latency, and despite the case that physical bandwidth is reaching a higher range, the amount of application level bandwidth is a small fraction of the available physical bandwidth [12]. With different interconnection networks, the network heterogeneity can become a significant factor in the parallel performance of applications.

- **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 communication overhead as we increase the number of machines.

- **Mapping**: As pointed out earlier both the programs and machines have certain characteristics which require the subtasks to be mapped to specific machines, to obtain the best performance. The work assigned to each processor must exploit the processor characteristics, and we must be careful to also consider the network heterogeneity while placing communication.

- **Load Balancing**: Homogeneous static load balancing algorithms must be adapted to work for heterogeneous NOMs. In [3] we present compile-time optimal and sub-optimal loop scheduling algorithms for heterogeneity. But since such systems support a multiuser environment, run-time dynamic load-balancing may be required. We have to trade-off the task-switching cost versus the load-imbalance cost. There are many dynamic load balancing schemes [13, 9, 5], but these cannot be used for problems with subtasks of various capabilities. There has been recent work looking at non-constant switching costs in [15], but this is a static scheme. Cho and Park [1] do this for Linear Array Networks. Since NOMs 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 differ-

ent machines. Although this introduces the overhead of data conversion while sending messages among the machines, it is generally not that significant [7].

- **Different parallelizations**: In heterogeneous environments, problem decomposition and task placement can have dramatic effects on performance. While the homogeneous machine and homogeneous network case is well understood for regular problems with no communication, such as matrix multiplication, even this case poses interesting issues when dealing with different machines. For example, let's consider a homogeneous collection of Sun SPARCstations connected via Ethernet, and a shared memory SGI multiprocessor, which has a dedicated interconnecting bus with low latency and high bandwidth. Let us further consider a parallelizable loop which has a high communication-to-computation ratio. It might not be worthwhile to parallelize this loop on the SPARCs due to the high latency and low bandwidth of the Ethernet, and we might want to simply replicate this loop on all the nodes. On the other hand, we may be able to get a performance improvement by parallelizing this loop on the SGI multiprocessor, due to its architectural features. The important point here is that depending on the underlying machine characteristics, a different parallelization may be required for good performance on different machines. As we consider the heterogeneous case, the complexity of the problem increases substantially.

- **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 NOM must take into account the contention that may be caused in the network. Modeling this is a very complex task. Besides load balancing at the machine level, we might need to monitor the load on the network.

- **Software Issues**: Differences in the host operating systems, file systems, database systems, interprocess 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 applications across a wide range of platforms.

- **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. We will discuss this point in more detail in section 5.

In this paper we will concentrate on two aspects of heterogeneity: different processor speeds and different memory capacities. Sections 4 and 5 discuss these two issues, which are important for any distributed application running in a heterogeneous environment, and we feel that they should be addressed first.

3 Methodology

To gain a better understanding of these issues, we study a suite of four parallel applications on a heterogeneous network of workstations. This section briefly describes the hardware and the parallel programs used in our experiments.

All the experiments were performed on a network of Sun workstations (SPARC 1, LX, 5 and 10), interconnected via an Ethernet LAN. Applications used C as the source code language, and were run on dedicated machines, i.e., there were no other users on the machines.

PVM [6] was used to parallelize the applications. PVM (Parallel Virtual Machine), is a message passing software system mainly intended for network based distributed computing on heterogeneous serial and parallel computers. PVM supports heterogeneity at the application, machine and network level, and supports coarse grain parallelism in the application.

We look at a number of applications, which include the following:

- Matrix Multiply (MxM): Multiplication of two square matrices.

- 2D-FFT: Two dimensional Fast Fourier Transformation.

- Cholesky Factorization (CHO): This program finds a lower triangular matrix L with positive diagonal elements such that $A = LL^T$, where A is a dense symmetric positive definite matrix.

- Economics (ECO), a commodity trade model [10]: 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.

4 Effect of Heterogeneous Processors

In this section we will discuss how the processor heterogeneity influences scheduling, i.e., data partitioning and distribution for near-optimal performance of the program. In our approach, we summarize the heterogeneity in one parameter — *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.

Knowledge of accurate processor speeds is crucial for load balancing in a heterogeneous environment. Generally, faster machines should receive more work to do. We will show that no single approximation of processor speeds is sufficient for load balancing purposes since speeds vary from one application to another.

The scheduling problem involves a tradeoff between the task granularity and efficient load balancing. As we increase the granularity of the parallelism or the number of tasks we get better load balance among the processors, but at the same time we increase the overhead, which negatively impacts the performance. We should choose the right value for the granularity by considering the potential for load imbalance, communication cost and the inherent parallelism in the application. The data partitions should match the processor characteristics.

A number of criteria have been presented to evaluate the performance of computers. For example, the MIPS (*million instructions per second*), MFLOPS (*million floating-point operations per second*), Whetstone, and Dhrystone ratings. Besides these, a number of kernels and benchmarks have been proposed to test the processor performance. 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.

Our approach was to measure the execution time for different applications on the network of workstations we have, and to take those speeds as an indication of the performance of the machine on a particular application.

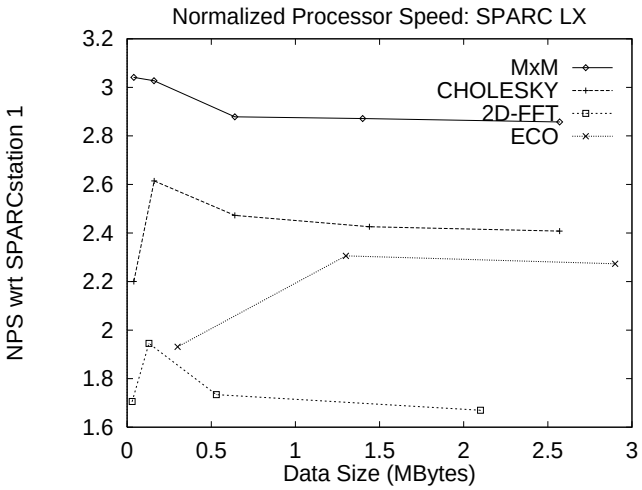

Figure 1: NPS: Sun SPARC LX vs. SPARC 1

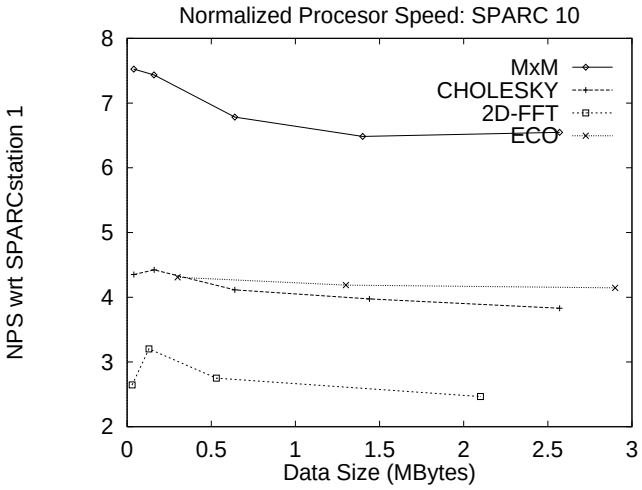

Figure 2: NPS: Sun SPARC 10 vs. SPARC 1

We now summarize our experimental results to show the performance of specific machines on different applications. Figures 1 and 2 show the normalized processor speeds of SPARCstation 10 and LX respectively. They also show the variance in the NPS as a function of the data size. All the speeds are with respect to the slowest machine, i.e., the SPARC 1 workstation. We can observe that as the data size increases, there is a slight decrease in the speeds till it reaches an almost constant level. The initial variations for small data sizes can be attributed to caching effects.

Our experiments indicate that machine performance varies for different applications. This has

important implications when computing in heterogeneous environments. For example, for optimal load balancing we might need to do a trial run of the application on the NOM, obtain the normalized processor speeds, and then distribute work among the processors based on these speeds. On the other hand, we may obtain these ratios by doing compile-time performance prediction. In [14], the author describes a detailed, architecture-specific, compile-time performance prediction framework. Porting to different architectures and compilers is quite involved, though possible.

We must also study the effects of varying the data size. This might require running the application for various sizes of input and storing these speeds in a table. This is certainly not feasible, therefore, we might need to extrapolate the speed ratios. However, for the programs we considered, we did not observe much variance with different data sizes.

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

Table 1: Performance Ratios: SPARC 5 wrt SPARC 1

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

Table 2: Running Time (SPARC 5 + SPARC LX)

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. In [3] we show how to use the NPS or other processor speed ratios in scheduling. For all of the above applications we parallelized the outermost doall loop, and distributed its iterations based on the NPS values. At this stage we ignore the effects of communication.

5 Effect of Heterogeneous Memory

As pointed out earlier, heterogeneous NOMs 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. Obviously, the largest data size is limited by the amount of combined memory present in the system. Several interesting issues arise when we try to efficiently drive the system to its maximum limit.

We mentioned in the last section that to obtain efficient load balancing we must distribute work among the processors based on their normalized speeds, which are application dependent. While doing the data distribution we must be careful so that we do not exceed the memory capacity of the machines, i.e., we must allocate work to the machines based on both the processor speed and available memory. We also need the memory capacity to determine how large an input size can be used for the programs.

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: Memory Capacities

One way of determining the available memory is to use the actual physical memory values in the system. Table 3 shows the the amount of actual physical memory and the amount that is available to user applications. 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 memory capacity on a given machine. 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[1]. We therefore introduce a new parameter, *Resident Memory Size (RMS)*, defined as the minimum number of pages of physical memory required to ensure that all page fault misses are cold misses (i.e.,

[1] see experiments in section 6.

due to the first reference), using a particular page replacement algorithm. We believe that this notion gives a better indication of the largest problem size we can run.

Note that 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:

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

The above RMS is calculated using an ideal page replacement scheme. Using the LRU page replacement instead, would give

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

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* [2] 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 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 will not be a good approximation for irregular problems.

6 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 NOM. 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 NOM.

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: Effect of NPS & total memory

Our experiments indicate that using the total memory requirement of the program may not be a good estimate of the size of the program instances that we can run. 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. This distribution causes the total memory requirement for the SPARC 5 (given in column 3) to exceed the user available memory (approximately 24.5Mb) for it. 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), showing that we can't use the total memory requirement as the criterion for choosing a problem size.

Table 5 shows the results obtained for MxM(2788×2788) on a configuration of SPARC 10 and SPARC 5, using the RMS value instead of the total memory requirement. We first ran the program by distributing the work based on the NPS values, but the RMS exceeded the memory on the SPARC 5, and caused the machine to thrash, and we stopped 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 (approx. 4:1, from table 3) to schedule the work (see under MEM). 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.

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

Table 5: Effect of NPS & RMS

We now discuss our scheduling algorithm. We first try to distribute the data among the processors proportional to their NPS values for the particular application in consideration. 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.

7 Conclusion

In this paper we have looked at the general issues that arise when computing in a heterogeneous network of machines environment. We specifically looked at the consequences of heterogeneity at the processor and memory level.

For efficient load balancing we must distribute work proportional to the processor capabilities or speeds. We found that a reliable method to find out the processor performance is to do sample runs of the pro-

gram, and obtain the *normalized processor speed*. This is highly application dependent as we observed a variance in the performance of the machines for different applications.

Typically we would like to solve large problem instances, limited only by the total combined memory in the NOM. We observed that the *resident memory size*, which may be application dependent, gives a good estimate of the memory requirement of an application. Finally, we point out that for efficiently solving large instances of an application, we must use both the *normalized processor speeds* and *resident memory sizes*, to achieve near-optimal performance.

References

[1] S.-Y. Cho and K. H. Park. Dynamic task assignment in heterogeneous linear array networks for metacomputing. *Proceedings of the Heterogeneous Computing Workshop '94*, pages 66–71, April 1994.

[2] M. Cierniak and W. Li. Unifying data and control transformations for distributed shared-memory machines. In *Proceedings of PLDI '95*, June 1995. Also available as Tech. Report 542, Computer Science Dept., Univ. of Rochester.

[3] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. Technical Report 540, Computer Science Dept., Univ. of Rochester, October 1994.

[4] J. C. DeSouza-Batista, M. M. Eshaghian, A. C. Parker, S. Prakash, and Y. C. Wu. A suboptimal assignment of application tasks onto heterogeneous systems. *Proceedings of the Heterogeneous Computing Workshop '94*, pages 9–16, April 1994.

[5] Derek L. Eager and John Zahorjan. Adaptive guided self-scheduling. Technical Report 92-01-01, Department of Computer Science, University of Washington, January 1992.

[6] 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.

[7] A. S. Grimshaw, J. B. Weissman, E. A. West, and E. C. Loyot. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 21(3):257–270, 1994.

[8] C. Leangsuksun and J. Potter. Designs and experiments on heterogeneous mapping heuristics. *Proceedings of the Heterogeneous Computing Workshop '94*, pages 17–22, April 1994.

[9] E.P. Markatos and T.J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *Proceedings Supercomputing '92*, pages 104–113, November 1992.

[10] 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 *Recent Advances in Spatial Equilibrium Modeling: Methodology and Applications*. Springer-Verlag, Heidelberg, 1995. forthcoming.

[11] B. Narahari, A. Youssef, and H.-A. Choi. Matching and scheduling in a generalized optimal selection theory. *Proceedings of the Heterogeneous Computing Workshop '94*, pages 3–8, April 1994.

[12] M. Parashar, S. Hariri, A. G. Mohamed, and G. C. Fox. A requirement analysis for high performance distributed computing over LAN's. *Proceedings of the First International Symposium on High Performance Distributed Computing*, pages 142–151, September 1992.

[13] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[14] Ko-Yang Wang. Precise compile-time performance prediction for superscalar-based computers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, June 1994.

[15] D. W. Watson, J. K. Antonio, H. J. Siegel, and M. J. Atallah. Static program decomposition among machines in an SIMD/SPMD heterogeneous environment with non-constant mode switching cost. *Proceedings of the Heterogeneous Computing Workshop '94*, pages 58–65, April 1994.