In Proceedings of the Fourth International Symposium on High Performance Distributed Computing, August 1995

Loop Scheduling for Heterogeneity [*]

Michał Cierniak, Wei Li, Mohammed Javeed Zaki
Computer Science Department, University of Rochester, Rochester, NY 14627

Abstract

In this paper, we study the problem of scheduling parallel loops at compile-time for a heterogeneous network of machines. We consider heterogeneity *in three aspects of parallel programming:* program, processor *and* network. *A heterogeneous program has parallel loops with different amount of work in each iteration; heterogeneous processors have different speeds; 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* sub-optimal *schedules of loops for load balancing, communication optimizations and network contention. Experiments show that a significant improvement of performance is achieved using our techniques.*

1 Introduction

With the rapid advances in the new high speed computer network technologies such as ATM (Asynchronous Transfer Mode), a network of workstations is becoming increasingly competitive compared to expensive parallel machines. However, it introduces heterogeneity in the virtual parallel machine, since it may consist of possibly many different types of processors. For example, we may have many HP, SUN, or DEC workstations, and a multiprocessor SGI machine in our network. The processors have different speeds and obviously the communication between processors on the SGI is faster than between one processor on the SGI and a SPARC.

The inherently dynamic nature of the configuration of the virtual machine, depending on what 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. In this paper, we study the problem of scheduling parallel loops at compile-time for a heterogeneous network of machines.

We consider *heterogeneity* in three aspects of parallel programming: *program*, *processor* and *network*. 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 consider heterogeneity in both the processor and the network dimensions.

  We take a very different approach than the work on stochastic models [6], which are intended to model the performance behavior of the whole system with possibly many jobs running at the same time in an unpredictable way, and are, therefore, more detailed and complicated. We develop this model in the context of compiler code generation and optimizations. Since compiler analysis can provide more information and insight into the program access pattern and behavior, the machine model can be simpler and more deterministic. Simplicity of the machine model is also required so that it can be used in the compiler for deciding optimizations.

- We develop a set of *architecture-conscious* compile-time scheduling algorithms for generating optimal or sub-optimal scheduling of loops for load balancing and communication, for a network of heterogeneous machines. We describe the optimal way to partition the iterations of parallel loops on the available processors. The parallel loops may be *homogeneous* or *heterogeneous*, and with or without communication.

  We show that for simple homogeneous loops without communication, the straightforward way of distributing iterations according to the relative speeds of the machines works well. However, for heterogeneous loops, and loops with communication and contention, new techniques must be employed to achieve good performance.

- 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 [9].

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 [12]. For NUMA (Non-Uniform Memory Access) parallel machines, loop scheduling has to take data distribution into account [7]. 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. Self-scheduling [13] is the simplest approach in which processors ask for additional work from the task queue when they become idle. Guided self-scheduling [11] reduces the runtime cost by allocating a block of iterations every time. Affinity scheduling [8] is a hybrid static and

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

dynamic scheduling algorithm that takes data locality into account.

Research in heterogeneous computing environments has focused on homogeneous applications. The problem of load balancing on a parallel machine with nodes of different performance has been considered in [1], [3] and [5]. An approach for scheduling in a machine with heterogeneous memories has been presented in [14]. Requirements for distributed computing over LAN's have been analyzed in [10].

The rest of this paper is organized as follows. In Section 2, we introduce our program model, which is followed by our machine model in Section 3. In Section 4, we consider scheduling for heterogeneous programs (on homogeneous machines). In Section 5, we look at the case of heterogeneous processors, with the same communication links. Section 6 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 7, we extend our model to handle the case of scheduling for load balancing while avoiding network contention. The experimental results are presented and discussed in Section 8. Finally, we conclude in Section 9.

2 Program Model

In this paper we will look at parallel loops, that is, loops whose iterations do not depend on one another. 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 messages are sent after the computation terminates and messages to the same destination are combined into one larger message.

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 network of heterogeneous machines.

2.1 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. For the heterogeneous loop case, we restrict our attention to loops where the computation is an affine function of the loop index, which captures a large set of real programs. 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 $x_i = ai + b$, that is the amount of computation is an affine function of the normalized loop index.

2.2 Communication

In the above section, we considered loops without communication. As mentioned earlier, by the nature of parallel loops, there is no communication necessary between their iterations. When we talk of communication, we mean messages that are sent at the end of the execution of the parallel loop, because of the need to communicate data to subsequent loops. 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.

Thus, we have 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.

3 Machine Model

We have been looking at a network of workstations and parallel machines. This introduces heterogeneity at two levels: one due to the potentially different processor speeds, and the other due to differences in communication cost between any two machines in the network. We want our model to take these factors into account.

3.1 Processor 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, γ, to describe the speed of a machine: γ_i — time for one operation on processor i. This synthetic parameter reflects the average operation time in an application with a typical operation mix.

The machines in the network can be homogeneous or heterogeneous. In the former case, the speed of all machines is given by the same parameter, γ. In the heterogeneous case that cost can vary depending on the machine. Therefore, we need a different value of that parameter for every machine. We will use γ_i to denote the speed of the ith machine in the considered configuration. The number of machines, p, is constant throughout the execution of the application.

3.2 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[1], and thus have the following two parameters: α_i — startup time for a message on processor i, and β_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 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 7.

3.3 Virtual Machine Models

Based on our discussion above we have the following four combinations of machines and networks:
(1) Homogeneous Processors and Homogeneous Network.
(2) Heterogeneous Processors and Homogeneous Network.
(3) Homogeneous Processors and Heterogeneous Network.
(4) Heterogeneous Processors and Heterogeneous Network.

For Sections 4, 5 and 6 we assume no contention in the network. Section 7 presents a scheduling algorithm for the case 1 above, that can be used in presence of contention.

4 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 3, the following machine parameters describe this type of machines: p (number of processors in the system), γ (time to execute one operation), α (communication initialization time), and β (time to send one byte of a message). As usual, n denotes the number of iterations of the loop.

As a simple introduction to loop scheduling, we first consider homogeneous parallel loops without communication. This is the simplest case of all — it is the first type of a loop described in Section 2.1. 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.

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 $i = 1, \ldots, n$. 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.

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,

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

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 iterations 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.

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 4. When $2p$ does not divide n, we have to use an approach similar to the algorithm described above. 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 $(\gamma a + \beta c)$[2], because this constant determines whether the time spent on computation and communication increases or decreases with the iteration number.

[1]We will consider a more complex communication model in Section 7.

[2]Recall that $y = ci + d$ is the communication for iteration i.

5 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), γ_i (time for processor i to execute one operation), α (communication initialization time), and β (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.

5.1 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 [5].

5.2 Homogeneous parallel loops, with communication

When there is communication, the algorithm in Section 5.1 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 $i = 1, \ldots, n$. 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 βy 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 $i, j = 1, \ldots, p$. 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 α 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 5.1.

5.3 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.

6 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), γ_i (time for processor i to execute one operation), α_i (communication initialization time on processor i), and β_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 5.1 and 5.3 can be applied to the case of a heterogeneous network.

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, α_i, we find a processor with the largest value of α_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 5.2 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.

We can similarly schedule a heterogeneous loop with communication. Details are given in [2].

7 Scheduling for Contention Avoidance

Sections 4, 5 and 6 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.

7.1 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:

- γ — time to execute one operation,
- α', β' — communication parameters for the part of the send operation performed locally (this is the part of communication that is not sequentialized),
- α'', β'' — 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 processor are identical and their ordering is arbitrary. With this assumption, we can give the 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}).

7.2 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.

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

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 $i = 2, \ldots, p$. 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.

$\square$

We will show below, an algorithm that will find a contention-free schedule if it exists. We can use Theorem 1 to simplify the formula for the completion time of the ith 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 $i = 1, \ldots, p$. By Theorem 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 $i = 2, \ldots, p$. 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.

$$\left\{ \begin{array}{rcl} vz_1 - wz_2 & = & -\alpha'' \\ vz_2 - wz_3 & = & -\alpha'' \\ \vdots & & \\ vz_{p-1} - wz_p & = & -\alpha'' \\ z_1 + z_2 + \ldots + z_p & = & n \end{array} \right.$$

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

We have shown in [2] that the above system of equations has a unique solution.

8 Experiments

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.

All experiments were performed on Sun workstations (SPARCstation 1, SPARCstation LX and SPARCstation 10) connected with an Ethernet network. The Programs were written in C and Fortran, and PVM [4] was used to parallelize them.

8.1 Heterogeneous programs

TRIANG [2] is a program with a heterogeneous loop used in the experiments presented in this section.

Figure 1 shows the speedups for two different parallelization of TRIANG. Label `bitonic` marks the results for the parallelization from Section 4. We compare our approach with the round-robin scheduling. The round-robin technique schedules a doall loop on p processors by assigning iterations $0, 0+p, 0+2p, \ldots$ to processor 0, iterations $1, 1+p, 1+2p, \ldots$ to processor 1 and so on. This approach is very popular in practice. It is very simple and yields acceptable performance.

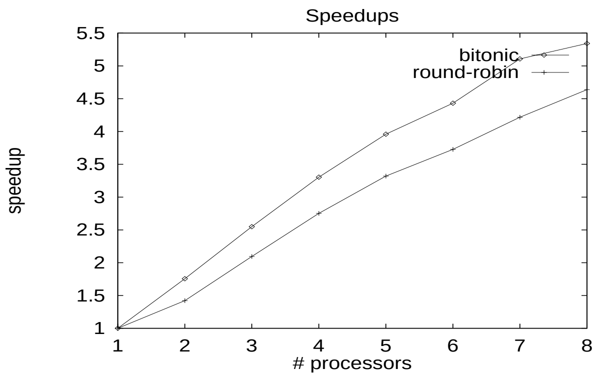

Figure 1: Speedups — TRIANG

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 1 demonstrates that the bitonic schedule consistently outperforms the round-robin technique.

8.2 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.

To find a static schedule on a network of heterogeneous computers, we have to know the processor speeds. In our discussion in Section 5, we used the parameter γ_i, to denote time for processor i to execute one operation. This is useful for analytical reasoning about distributed computing systems, but does not fully capture the complexity of modern processors. Not only do the different operations have different times, but also pipelining and multiple instruction issue make finding a combined time for a sequence of operations very hard. Consider two of the workstations used in our experiments: SPARCstation 1 and SPARCstation LX. An average operation time depends on the instruction mix. This is reflected in relative speeds of those two machines. For matrix multiply, SPARCstation LX is 1.85 times faster than SPARCstation 1, but for the heterogeneous loop example, this ratio is 2.15. The difference in relative speeds of a SPARCstation 10 and a SPARCstation 1 is even larger for those two applications. It is 3.00 and 5.65 respectively, for the two programs.

Therefore, a better approach to load balancing is to first measure execution times of the sequential program that we are about to parallelize on small data sizes on every machine which we want to use in our heterogeneous system. The times can be used to obtain relative speeds of those machines for this particular application [14].

In this section we consider programs without contention. In this case our goal is to obtain the best possible load balance using the algorithm in Section 5. We will use the following approach to evaluate the parallelization. First we will normalize the speeds of all machines in the configuration relatively to the speed of one of the processors — the *base processor*. Any machine can become the base processor and it always has the normalized speed of 1. For this experiment, our parallel machine consists of a network of three types of workstations. We normalize the speeds relatively to the slowest of our computers, SPARCstation 1.

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.

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

Table 1: Speedups — matrix multiply

The results for matrix multiply are given in Table 1. The program multiplies two square matrices of the size 600×600. 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 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 1 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.

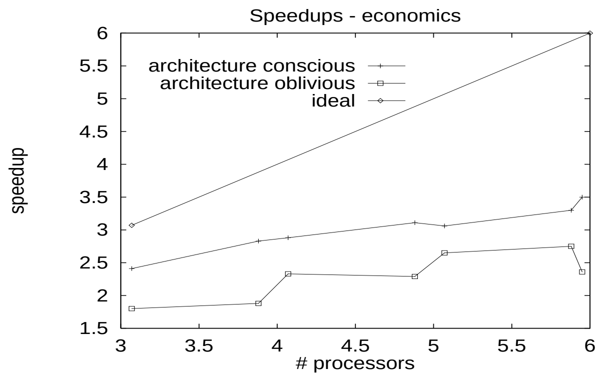

Figure 2: Speedups — economics

The second example of the homogeneous loop case is a program for *spatial price equilibrium modeling* in economics [9]. The program applies the methodology of the theory of variational inequalities for formulation and computation of spatial price equilibrium models with discriminatory ad valorem tariffs, which is a widely-used trade policy instrument. It provides solutions to a variety of medium-scale and large-scale problems relevant for empirical modeling of international commodity trade.

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 2 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.

8.3 Contention Avoidance

We use LFK10, a program [2] based on loop 10 from the Livermore Fortran Kernels, to demonstrate our contention avoidance algorithm.

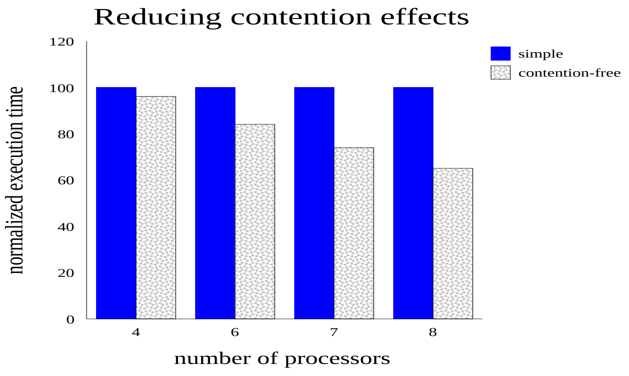

Figure 3: Execution times — LFK10

The outermost loop is a doall loop and it is being parallelized. We assume, however, that for the next stage of computation array must be broadcast to all processors. This causes high level of contention in our Ethernet network. We can use the algorithm developed in Section 7 to maximize the speedup by minimizing contention.

Figure 3 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 contention-free schedule results in a significantly faster program.

For this example the speedups achieved by the contention-free 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 7 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.

9 Conclusions

In this paper, we studied the problem of scheduling parallel loops at compile-time for a heterogeneous network of machines. We considered *heterogeneity* in three aspects of parallel programming: *program*, *processor* and *network*.

We have proposed a simple yet comprehensive model for a network of processors. The model was designed for use in compiling for load balancing and communication. We developed compiler algorithms for generating *optimal* and *sub-optimal* schedules of loops for load balancing, communication optimizations and network contention. Our experiments showed that the new techniques can significantly improve the performance of parallel loops over existing techniques.

References

[1] A. L. Cheung and A. P. Reeves. High performance computing on a cluster of workstations. *Proc. of the 1st Int. Symposium on High Performance Distributed Computing*, pages 152–160, September 1992.

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

[3] P. E. Crandall and M. J. Quinn. A decomposition advisory system for heterogeneous data-parallel processing. *Proc. of the 3rd Int. Symposium on High Performance Distributed Computing*, August 1994.

[4] 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.

[5] 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.

[6] Joseph L. Hammond and Peter J.P. O'Reilly. *Performance analysis of local computer networks*. Addison-Wesley, 1986.

[7] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 11(4), November 1993. An earlier version appeared in Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems, October, 1992.

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

[9] 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.

[10] M. Parashar, S. Hariri, A. G. Mohamed, and G. C. Fox. A requirement analysis for high performance distributed computing over LAN's. *Proc. of the 1st Int. Symposium on High Performance Distributed Computing*, pages 142–151, September 1992.

[11] C. Polychronopoulos and D. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, 36:1425–39, December 1987.

[12] C. D. Polychronopoulos. *Parallel Programming and Compilers*. Kluwer Academic Publishers, 1988.

[13] P. Tang and P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proc. of '86 International Conference On Parallel Processing*, August 1986.

[14] M. J. Zaki, W. Li, and M. Cierniak. Performance impact of processor and memory heterogeneity in a network of machines. In *Proc. of the Fourth Heterogeneous Computing Workshop*, April 1995.