Abstract

We describe a compiler and run-time system that allow data-parallel programs to execute on a network of heterogeneous UNIX workstations. The programming language supported is Dataparallel C, a SIMD language with virtual processors and a global name space. This parallel programming environment allows the user to take advantage of the power of multiple workstations without adding any message-passing calls to the source program. Because the performance of individual workstations in a multi-user environment may change during the execution of a Dataparallel C program, the run-time system automatically performs dynamic load balancing. We present experimental results that demonstrate the usefulness of dynamic load balancing in a multi-user environment.

1 Introduction

Distributed computing environments consisting of workstations connected over a local area network have become commonplace. They provide high quality personalized computing along with resource sharing, and they offer unmatched price/performance ratios.

In the first half of the 1980s, most of the research in the area of distributed computing was concentrated on building distributed operating systems such as V-Kernel [6], Accent [16], Amoeba [19] and Eden [15]. Gosinski [11] describes a main goal of a distributed operating system to be the allocation of network resources that allows their use in the most effective way. More recently, however, significant work has been done to develop programming systems that would support development and execution of parallel applications across a network of workstations. Amber [5] is an object-based system with primitives for thread management and object mobility. Munin [3] is a distributed shared memory system that allows parallel programs written for shared memory multiprocessors to be executed in a distributed environment. Several software packages, such as PVM from Oak Ridge National Laboratory [2] and TCGMSG from Argonne National Laboratory, provide low-level functions for communication between processes on different machines. These low-level functions are similar to message-passing primitives for a distributed memory multicomputer.

This paper focuses on data-parallel programming in a distributed computing environment. Data parallelism has proven to be the natural paradigm for a large number of problems in science and engineering [9, 10]. The programming language used is Dataparallel C, derived from the original C* language developed by Thinking Machines Corporation. Dataparallel C is a high-level SIMD language, free of message-passing primitives. Prior publications have described the implementation of Dataparallel C compilers for Sequent shared memory multiprocessors and Intel and nCUBE distributed memory multicomputers [12, 13]. We have modified the multicomputer-targeted compiler and written a new run-time system that allows Dataparallel C programs to execute on a network of heterogeneous UNIX workstations. Since this is a multi-user environment, where the performance of individual workstations may vary while programs execute, the run-time system automatically performs dynamic load balancing.

Section 2 of this paper gives an overview of the Dataparallel C language. Section 3 describes how the existing Dataparallel C compiler for distributed memory machines works and how we had to modify it. Modifications to the existing routing library for hypercube multicomputers and the implementation of dynamic load balancing are discussed in Section 4. Section 5 presents our preliminary experimental results, and Section 6 summarizes the contributions of this work.

2 Dataparallel C

Dataparallel C is a variant of the original data-parallel programming language C* [17]. The Dataparallel C language is described in detail in [12]. In
I. #define Lp 1024 /* Dimension of matrices to be multiplied */
2. #define DIM 128 /* Size of submatrices */
3. #define ID (this-&x CO] [Ol)
4. #define ROU(i) (i+ID/(N/DIM)*DIM)
5. #define COL(i) (i+ID%(N/DIM)*DIM)
6. domain block submatrix {
7.   float a[DIM][DIM], b[DIM][DIM], c[DIM][DIM]; } x[N/DIM][N/DIM];
8. main()
9. {
10.   int m;
     /* Matrix initialization omitted */
11.   start_timer();
12.   [domain submatrix].{
13.     int skew;
14.     /* Skew matrices a and b to align submatrices properly */
15.     for (skew = 0; skew < ID/(N/DIM); skew++) west()->a = a;
16.     for (skew = 0; skew < ID%(N/DIM); skew++) north()->b = b;
17.     /* Perform matrix multiplication */
18.     for (m = 0; m < N/DIM; m++) {
19.       matm(a, b, c);
20.       west()->a = a;
21.       north()->b = b;
22.     }
23. }
24.   stop_timer();
25. }

Figure 1. A Dataparallel C program to multiply two 1024x1024 matrices. The line numbers are not part of the program; they are for reference purposes only. Function matm, called in line 18, is an ordinary C function that multiplies matrices a and b and adds the product into matrix c.

this section we summarize the principal features of the language, using a matrix multiplication program as an example (see Figure 1).

Dataparallel C is a SIMD superset of the C programming language; it extends C to provide the programmer access to a virtual parallel machine. The programming model is based upon virtual processors, global name space, and synchronous execution of a single instruction stream. The Dataparallel C programmer envisions a SIMD computer consisting of a front-end uniprocessor attached to an adaptable back-end parallel processor. The sequential portion of the Dataparallel C program (consisting of conventional C code) executes on the front end. The parallel portion of the Dataparallel C program (delimited by constructs not found in C) executes on the back end.

The back end is adaptable in that the programmer selects the number of processors to be activated. This number is independent of the number of physical processors that may be available on the hardware executing the Dataparallel C program. For this reason the Dataparallel C program is said to activate virtual processors when a parallel construct is entered.

Virtual processors are allocated in groups. Each virtual processor in the group has an identical memory layout. The Dataparallel C programmer specifies a virtual processor's memory layout by using syntax similar to the C struct. A new keyword domain is
used to indicate that this is a parallel data declaration. As in C structures, the names declared within the domain are referred to as members.

Instances of a domain are declared using the C array constructor. Each domain instance becomes the memory for one virtual processor. The array dimension therefore indicates the size of the virtual back-end parallel processor that is to be allocated. Lines 6-7 of Figure 1 declare a domain type called submatrix and a two-dimensional array of instances of that domain type. Note that all domain arrays can be multidimensional. The number of virtual processors allocated is the product of the array dimensions.

Data located in Dataparallel C's front-end processor are termed mono data. The variable declared in line 10 of Figure 1 is a mono variable. Data located in a back-end processor is termed poly data. The variables declared in lines 6-7 and 13 are poly variables.

Lines 12-21 of Figure 1 illustrate the Dataparallel C domain select statement. The body of the domain select is executed by every virtual processor allocated for the particular domain type selected. The virtual processors execute the body synchronously. The domain members are included within the scope of the body of the domain select. These names refer to the values local to a particular virtual processor.

Dataparallel C provides a global name space. The code executing in a virtual processor can reference a variable in the front-end processor by referring to the variable by name. Similarly, the members of a domain instance are accessible everywhere in a program.

Like C++, Dataparallel C has a keyword this. In Dataparallel C this is a pointer to the domain instance currently being operated on by a virtual processor. Dataparallel C programs often use this in expressions so that individual virtual processors can determine their positions in the virtual processor array (see lines 3-5 of Figure 1).

For low dimension domain arrays the compiler recognizes keywords that provide virtual processors with convenient access to their neighbors. In the case of two-dimensional arrays, the keywords north(), northeast(), east(), southeast(), south(), southwest(), west(), and northwest() provide access to mesh neighbors. The program of Figure 1 uses these keywords in lines 14-15 and 18-19 to move data among virtual processors. Note that virtual processors interact without explicit message passing.

The Dataparallel C programmer can provide information to the compiler to help it map virtual processors to physical processors. For example, the keyword block in line 6 of Figure 1 directs the compiler to map nearly square submatrices of the domain element matrix to the same physical processor.

3 Compiler Technology

The starting point of our work was the implementation of Dataparallel C for hypercube multicomputers [12, 14]. The Dataparallel C compiler accepts a Dataparallel C program and produces a C program which is replicated on all physical processors. In that sense the compiler can be viewed as translating a SIMD specification into a SPMD implementation. Standard C code that the compiler emits is augmented with calls to a routing library. This approach has two important benefits. First, it allows us to take advantage of optimizing C compilers tuned for specific processors. Second, it makes the compiler more portable. To port the compiler to a new machine it is enough to re-implement the routing library to reflect the target machine's architecture and communication protocols. However, in order to exploit fully the computing power of heterogeneous workstations, we had to make some modifications to the compiler.

In most Dataparallel C programs the number of virtual processors will far exceed the number of physical processors. The compiler generates code so that a single process executing on a physical processor emulates a set of virtual processors. Since all the processors in a hypercube multicomputer are identical, and the user can acquire them for dedicated use, the compiler allocates an equal number of virtual processors to every physical processor. In a network environment this is usually not the case; workstations can differ significantly in their computing power. If all the workstations were to get the same share of virtual processors this would cause noticeable load imbalance, and the execution time would be determined by the slowest workstations. To avoid this, we have modified the compiler to allow users to specify the relative speeds of individual workstations at compile time. The compiler then distributes virtual processors according to this specification.

The compiler needs to implement standard mappings between a domain array index and a pair of values: physical processor number, virtual processor offset. While this is done with simple macros in the original compiler, our modification sometimes requires array lookup, since each physical processor has to know the number of virtual processors on all other physical processors. Although this introduces some extra overhead, the benefit of non-uniform distribution of virtual processors is much greater, because it allows for the potential use of all workstations to their full power.
In a multi-user environment the load on a workstation can vary significantly over time. For this reason our run-time system performs dynamic load balancing which has led to another compiler modification. While the original compiler used the initial static distribution to allocate enough memory to hold the maximum number of virtual processors stored in any physical processor, our modified compiler allocates memory for the maximum possible number of virtual processors that any workstation may have to emulate during the execution of the program. Inside that large chunk of memory domain array locations are dynamically addressed, based on the current number of virtual processors on a given physical processor.

4 Run-Time System

Our run-time system consists of the routing library, which has evolved from the routing library for hypercube multicomputers, and the mechanism for automatic load balancing.

4.1 Routing Library

Since elements of a domain array are distributed across workstations, each reference to a domain element not located on the referencing processor requires an inter-process communication. The network implementation of the routing library is based on the 4.3BSD UNIX socket interface. Using connection-oriented sockets and the standard TCP/IP communication protocol, we build a fully-connected topology among the participating workstations. Each processor can directly communicate with any other processor.

Communication patterns and functions in the routing library for hypercube multicomputers are described in [1]. Initially, our goal was to use the library as is and only re-implement hypercube communication primitives for sending and receiving messages. While this approach provided almost immediate functionality of the system, further experiments have shown more efficient ways to implement some of the communication routines. In particular, we have implemented new algorithms for the reduction, random-read, and random-write routines.

A common operation in Dataparallel C is a reduction of multiple poly values into a single mono variable. Because the compiler spreads poly values throughout the memories of the individual processors, communication is needed to perform a reduction. In the routing library for hypercube multicomputers, processors first perform local reduction of the values they own, and then exchange locally reduced values across each of the hypercube dimensions. In our network version of reduction operation, after performing local reductions, all processors send their locally reduced values to one processor (fan-in), which then performs global reduction and broadcasts the reduced value to all other processors (fan-out). While it might seem that the processor performing global reduction would be a bottleneck in this operation, our experiments have shown that this algorithm outperforms hypercube-like style of communication when the number of workstations is relatively small.

Global name space in Dataparallel C allows every virtual processor to store or fetch a value belonging to any other virtual processor. This is achieved through general-purpose communication functions called random-write and random-read. The hypercube routing library implements random-write by first having every processor sort its communication packets according to their destination processor. After that, processors exchange packets across each hypercube dimension, with each processor storing the values destined for itself, and further exchanging the values destined for other nodes. The fully connected topology that we build in the network implementation significantly simplifies this algorithm. Each processor first sorts communication packets, and then directly sends corresponding packets to their destination processor.

4.2 Automatic Load Balancing

Our modification of the compiler to allow non-uniform distribution of virtual processors has significantly increased the potential to exploit fully the computing power of all workstations. However, in an environment where each workstation can be accessed by multiple users, there is a need to control the load of all workstations dynamically and remove any significant imbalance.

Virtual processors are the basic units of parallelism in Dataparallel C, and their number is usually much larger than the number of physical processors. It seems natural that load balancing could be achieved by dynamically changing the number of virtual processors that each physical processor has to emulate.

According to the taxonomy given in [4], our algorithm falls in the class of dynamic physically distributed cooperative load balancing algorithms. There is no one location where load information of all the processors is collected, but rather, all the processors cooperate in making the decision about load redistribution.

There are four parts to every dynamic load balancing algorithm: monitoring local load on each processor (load screening), exchange of the load information between the processors, making the decision about process migration, and performing process migration it-
As the measure of each processor's load we take the average computation time per virtual processor emulated. A potential problem with this measure is the situation where the variance in the amount of work performed by the virtual processors is large, but in most data-parallel algorithms this is not the case. Also, in order for this measure to reflect accurately load of each workstation, the period of time between two load screening must not be too short, since this could cause rapid fluctuations in the load of the processors.

Exchanging the load information is always followed by the decision on whether to do load redistribution or not. The critical question is how often to exchange the load information between processors. If not enough time has elapsed between two information exchanges and possible load redistributions, the program can spend considerable amount of time just migrating the data. On the other hand, if this period is too long, the system might suffer from a serious load imbalance. Our algorithm estimates the time to perform an average load redistribution, and then sets the time between two load information exchanges in such a way that the estimated load redistribution time represents only a small fraction of the time between two information exchanges, e.g., 5%.

The example in Figure 2 illustrates the load balancing algorithm. Suppose we run an application with 90 virtual processors on three workstations, and suppose each workstation initially emulates 30 virtual processors.

At the first load screening the system records the amounts of time that workstations have spent emulating their own virtual processors. It is important to note that this time reflects only the computation, and not the communication, and that the faster processors have spent more of their time waiting for the messages from the slower processors. Given these times, we find the number of virtual processors that each workstation has been able to emulate per unit of time. These numbers represent the relative computing power of the workstations since the program began execution. Adding the three computing power coefficients we see that the aggregate system performance is 13.0 virtual processor emulations per second. The run-time system distributes virtual processors based on each workstation's speed, relative to the aggregate speed. For example, the number of of virtual processors assigned to workstation 1 until the next load balancing step is 

\[
90 \times \left( \frac{6.54}{13.0} \right) = 45.
\]

In the second load balance call we again record the computation time for each workstation, and find the coefficients of available computing power. According to those coefficients, adjustment of the load should be performed by moving one virtual processor from workstation 2 to workstation 1. However, under assumption that an average virtual processor migration takes 5% of the computation time between two load balancing calls, we make the decision not to redistribute the load, if the adjustment is not greater than 5% on at least one of the workstations. Therefore, numbers in parentheses represent "ideal" numbers of virtual processors for workstations 1 and 2, but the actual numbers of virtual processors have not been changed.

The final step of our load balancing algorithm is load redistribution itself. Since every processor executes exactly the same program (SPMD model) and since mono variables in Dataparallel C are replicated on all physical processors, load redistribution can be done by simply moving domain array elements between the processors. In that way we change the num-
ber of virtual processors each physical processor is to emulate and effectively balance the relative amount of work it has to do. Since Dataparallel C parallel code is replaced by virtual processor emulation loops, where the loop limit is given by the number of virtual processors that particular physical processor owns, load balancing cannot be done inside virtual processor emulation loops. Thus, load balancing is always initiated from within routing library functions. These functions record the time on each of the workstations, and when necessary call the load balancing routine that will take care of load information exchange and virtual processor migration.

5 Experimental Results

Communication costs in a network of workstations are relatively much higher then similar costs in a distributed memory multicomputer. Our system is currently implemented on a local area network of Sun SPARCstations. Ethernet based LAN with baseband technology provides a relatively low maximum transfer rate of 10Mbps, which introduces two limitations. First, the number of workstations participating in a program execution cannot be very large, and second, the grain size of the program should not be too small, since the benefit of multiple workstations power would be nullified by high communication cost. An important advantage of running an application distributed over several workstations is the likelihood of increased data locality, which in some cases can make up for high communication costs.

We have run several Dataparallel C benchmark programs and initial results are very encouraging. Here we describe three of those programs and present performance results.

Program matrix multiplies two 1024x1024 matrices containing single-precision floating point numbers. The program is based on block matrix multiplication (See Figure 1). The algorithm divides factor matrices A and B into 128x128 blocks. After properly aligning the blocks of A and B, each virtual processor repeatedly performs sequential matrix multiplication on the blocks of A and B it currently controls, then participates with the other virtual processors to rotate the blocks of A to the previous column and the blocks of B to the previous row. It is interesting to note that a sequential implementation of block matrix multiplication executes faster on a single SPARCstation than the standard "textbook" matrix multiplication algorithm. We believe this is because the cache hit rate is higher. For this reason the time of the best sequential program and the time of the Dataparallel C program executing on one processor are identical.

Program g-jordan solves a dense system of 1000 linear equations using the Gauss-Jordan method. Although the Gauss-Jordan algorithm requires more floating-point operations than Gaussian elimination, it requires fewer communication steps, because it does not have a back substitution phase. For that reason we have found that Gauss-Jordan executes faster on distributed memory parallel computers when the system size is relatively small. To compute the speedup achieved by program g-jordan, we compare its execution time against that of our best sequential Gaussian elimination program.

Our third program, shallow, is a Dataparallel C implementation of the shallow water benchmark program developed at the National Center for Atmospheric Research in Boulder, Colorado. The model consists of the nonlinear shallow-water equations in two horizontal dimensions. The program solves the equations using finite difference approximations. In its original form the program used a 64x64 grid. We have increased the grid size to 128x128 to increase the running time of the program.

Program sharks implements the time-driven simulation algorithm proposed in [7]. The Dataparallel C implementation of this problem is presented in detail in [12]. We have measured the performance for the case when the number of sharks and fish is 2048, the total simulation period is 100000, and the time step is 10000. There is no processor interaction during the simulation, and therefore this program can achieve almost perfect efficiency.

<table>
<thead>
<tr>
<th>Program</th>
<th>Seq. Time</th>
<th>Speedup - 4</th>
<th>Speedup - 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix</td>
<td>530.6s</td>
<td>3.5</td>
<td>5.6</td>
</tr>
<tr>
<td>g-jordan</td>
<td>397.5s</td>
<td>2.8</td>
<td>2.2</td>
</tr>
<tr>
<td>shallow</td>
<td>602.8s</td>
<td>3.7</td>
<td>6.3</td>
</tr>
<tr>
<td>sharks</td>
<td>261.2s</td>
<td>4.0</td>
<td>7.8</td>
</tr>
</tbody>
</table>

Table 1. Execution time in seconds of sequential programs and speedups on 4 and 8 SPARCstation 2s

It is important to note that performance was benchmarked with dedicated processors, but we did not have exclusive access to the network. Competing with other users for the network bandwidth has certainly increased the communication cost. As can be seen from the performance figures recorded in Table 1, our benchmark programs run with considerable efficiency. Program g-jordan does not perform well on 8 workstations due to an exceptionally high overhead produced when a workstation has to broadcast a large
message to many other workstations. We attribute this overhead to Sun Microsystems' implementation of the TCP/IP protocol suite.

The previous benchmarks statically assigned the same number of virtual processors to every workstation and did not perform dynamic load balancing. In order to test our non-uniform distribution of virtual processors and automatic load balancing algorithm, we have run a long series of experiments. The test program was the shallow water model, which runs long enough that its performance can depend on load imbalance. Configuration used consists of four Sun workstations: 2 SPARCstation 2s, 1 SPARCstation 1+, and 1 SPARCstation 1. We have experimented with four different policies for allocating virtual processors to workstations:

1. Allocate the same number of virtual processors to all workstations. Do not perform run-time load balancing.
2. Distribute virtual processors to workstations based upon their relative speeds. Do not perform run-time load balancing.
3. Allocate the same number of virtual processors to all workstations. Perform run-time load balancing.
4. Distribute virtual processors to workstations based upon their relative speeds. Perform run-time load balancing.

To get reliable performance data, we executed program shallow 400 times, 100 times for each policy. For each policy, 50 executions occurred with very light load on all the workstations, and in the other 50 executions one of the workstations was given an extra, CPU intensive, job. Each workstation was chosen about the same number of times to be the one with the extra job. The results of the experiments are summarized in Tables 2 and 3.

The experiment has clearly shown that allocating virtual processors based on the relative speeds of the workstations is superior to a uniform work distribution under both light and heavy load. Since dynamic load balancing introduces some overhead, static load balancing based upon the speed of workstations performs slightly better than dynamic load balancing when the load is very light, or the user has workstations for dedicated use. However, for this experiment we can say with the level of confidence 0.95 that the average difference is 2% or less in the case when both static and dynamic load balancing are used and 6% or less in the case when dynamic load balancing is used with a uniform initial distribution.

The big advantage of dynamic load balancing can be seen under heavy load conditions. For this experiment we can say with confidence level 0.95 that the average execution time will be at least 25% shorter if dynamic load balancing is used. On the other hand, there is only a slight difference in performance between policies 3 and 4; with confidence level 0.95 we can state that they will perform within 4% of each other. These two results lead us to believe that dynamic load balancing is a much more powerful tool than static load balancing. This is good, because it means that the compiler need not know about the speeds of the machines that will participate in a program execution, since the dynamic load balancing algorithm will automatically adjust the load. The user does not have to recompile the program before executing it on workstations with different performance characteristics.

### Table 2.

<table>
<thead>
<tr>
<th></th>
<th>No static load balancing</th>
<th>Static load balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dynamic load balancing</td>
<td>384.5 (90.5)</td>
<td>237.2 (65.7)</td>
</tr>
<tr>
<td>Dynamic load balancing</td>
<td>247.1 (82.0)</td>
<td>238.7 (74.2)</td>
</tr>
</tbody>
</table>

**Table 2.** Mean execution time in seconds of program shallow on 4, lightly loaded heterogeneous workstations. Parenthesized number is variance in seconds.

### Table 3.

<table>
<thead>
<tr>
<th></th>
<th>No static load balancing</th>
<th>Static load balancing</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dynamic load balancing</td>
<td>551.6 (20901.4)</td>
<td>438.4 (218.8)</td>
</tr>
<tr>
<td>Dynamic load balancing</td>
<td>327.7 (199.2)</td>
<td>320.3 (266.5)</td>
</tr>
</tbody>
</table>

**Table 3.** Mean execution time in seconds of program shallow on 4 heterogeneous workstations, when one of the workstations is executing another CPU intensive job. Parenthesized number is variance in seconds.

6 Conclusions

It is widely recognized that networks of workstations hide large amounts of unused computing power. Writing parallel applications to run across networks of workstations using message-passing primitives is at least as difficult as programming distributed memory.
multicomputers using their low-level languages. We have developed a system for running data-parallel programs efficiently on a network of workstations. Our run-time system is built on the top of the standard SunOS operating system. Despite the lack of a specialized kernel that would enable faster interprocessor communication, and despite relatively slow Ethernet LAN technology, our initial experiments have shown reasonable performance. We are currently working on porting our system to a network of IBM RS/6000 workstations, which we believe will provide a high performance computing environment.

To take full advantage of the computing power of heterogeneous workstations in a multi-user environment, we have developed an algorithm for dynamic load balancing. Dynamic load balancing is automatically performed by the run-time system, thus freeing the user of any load considerations, while providing nearly optimal execution time.

The current environment assumes that all workstations have the same architecture, but minor enhancements to our software would allow users to execute data-parallel programs on networks with workstations from multiple vendors.

Acknowledgements

Anthony Lapadula wrote the multicomputer targeted Dataparallel C compiler, which we used as a starting point for our work. Ray Anderson wrote the multicomputer routing library which was used at the very beginning of this research. Doran Wilde was the first to implement multicomputer communication primitives in the network environment.

This work was supported by National Science Foundation grant CCE-8906622.

References


