Automatic Generation of Parallel Programs with Dynamic Load Balancing

Bruce S. Siegell
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Peter Steenkiste
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Existing parallelizing compilers are targeted towards parallel architectures where all processors are dedicated to a single application. However, a new type of parallel system has become available in the form of high performance workstations connected by high speed networks. Such systems pose new problems for compilers because the available processing power on each workstation may change with time due to other tasks competing for resources. We argue that it is possible for a parallelizing compiler to generate code that can dynamically shift portions of the application's workload between processors to improve performance. We have implemented a run-time system that supports automatically generated programs with dynamic load balancing. In this paper we describe this system and present performance measurements. We also describe the compiler functionality needed to generate parallel programs with dynamic load balancing.

1 Introduction

There has been a fair amount of success in developing parallel languages, e.g., [1] [2], and parallelizing compilers, e.g., [3] [4], for MIMD distributed memory machines. This research has primarily targeted tightly-coupled machines, such as the Intel Paragon or the Cray T3D. However, the most prevalent distributed-memory system is a group of workstations connected by a local area network. As high-speed networks become more common and workstations become faster, it becomes practical to treat a network of independent computers as a multicomputer. The goal of this research is to explore the issues in targeting parallelizing compilers for such network-based multicomputers.

A common method of parallelization used by parallelizing compilers is the distribution of iterations of loops. Each node executes the same program (SPMD programming model) but executes only a subset of the loop iterations. The Fortran D [3], Fx [5], Vienna Fortran [6] and AL [4] compilers distribute DO loops with the aid of data alignment and distribution directives provided by the programmer. Loop iterations are executed using the *owner computes* rule: each node does the computations associated with the data assigned to it. In most cases, the distribution of loop iterations is static. However, on systems where processing requirements or processing capabilities change over time, static distribution of work will lead to inefficient execution due to load imbalance. Some languages, e.g. Fortran D, include dynamic data distribution directives to balance load and reduce communication requirements as data access patterns change; however, these optimizations are performed at compile time and do not address load imbalances due to a dynamic processing environment.

Our research explores mechanisms for *dynamic load balancing* of automatically generated parallel code, and how these mechanisms can be incorporated into a parallelizing compiler for a distributed memory machine. Dynamic load balancing is defined as the allocation of the work of a single application to processors at run time so that the execution time of the application is minimized. Many researchers have explored the problem of dynamic scheduling of parallel loop iterations [7] [8] [9] [10] [11] [12] [13], but most have not considered the problems of a distributed memory environment. A few researchers [11] [12] have suggested approaches that take locality issues into account for machines with non-uniform memory access times, but their main target is still shared memory machines. Researchers [13] have also looked at dynamic scheduling of loop iterations on a network of workstations using Dataparallel C, an explicitly parallel language. Our approach to load balancing is similar to the work with Dataparallel C, but our emphasis is on issues relating to automatic parallelization.

The remainder of the paper is organized as follows. We first discuss the features of our application domain and runtime environment that have an impact on load balancing (Section 2). In Section 3 we discuss the dynamic load balancer that we used as a starting point, and we present the extensions that are needed to both the compiler and the

This work was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract MDA972-90-C-0035.

load balancer for automatically generated code in Section 4. We present performance results for an implementation of the load balancer over Nectar [14] in Section 5, and we end with a discussion on related work and conclusions.

2 Load Balancing Considerations

Automatically generated code for dynamic load balancing must adjust to the characteristics of the application and the run-time environment without involvement of the programmer. In this section we discuss the features of applications and the run-time environment that have an impact on the load balancer.

2.1 Application features

Several application features impose constraints on the design of an efficient load balancing system. Table 2.1 summarizes the presence or absence of these properties for three commonly-used routines that we use as examples throughout the paper: matrix multiplication (MM), successive overrelaxation (SOR), and LU decomposition (LU).

Data dependences in the application have a big impact on load balancing. First, dependences in the sequential code correspond to communication, and thus, synchronization, in the parallelized code. The goal of the load balancer should be to balance the computation times between these synchronization points. If a distributed loop has loop-carried dependences, the mapping of iterations to processors affects the amount of communication. The load balancer should consider the impact of the mapping on communication when moving work.

If a distributed loop is nested, work movement can be more efficient than if it is the outermost loop. When the distributed loop is an inner loop, each iteration of the distributed loop may access the same distributed data on successive iterations of the enclosing loops [11]. Thus, if this data is shifted between processors to balance load, more work is moved per data element than if the distributed loop were the outermost loop.

Finally, many scientific computations are irregular, and the load balancer cannot always assume that both the number and the size of work units will remain constant. This makes it harder for the load balancer to predict the effect of work movement. The bounds of the distributed loop may depend on values of outer loop indices, in which case the load balancer will have to keep track of the iterations that must be balanced at run time. If the bounds of loops inside the distributed loop vary, then the size of the work associated with iterations of the loop will vary. Similarly, the presence of conditionals in the distributed loop makes it difficult to predict the cost of different iterations. Note that these features also pose problems for a static load distribution on a dedicated system.

Property (of distributed loop)	MM	SOR	LU
loop-carried dependences	no	yes	no
communication outside loop	no	yes	yes
repeated execution of loop	yes	yes	yes
varying loop bounds	no	no	yes
index-dependent iteration size	no	no	yes
data-dependent iteration size	no	no	no

Table 1: Application properties.

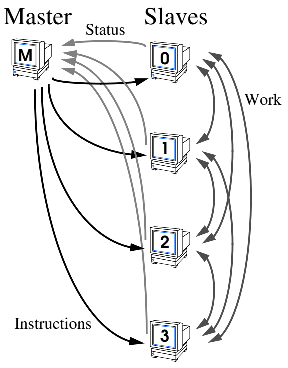
2.2 Environment features

Dynamic load balancing shifts workloads to adapt to changes in the environment. Ideally, load balancing should be performed as frequently as possible to track load changes as closely as possible. However, load balancing costs make this impractical. First, there is overhead associated with collecting the information to make a load balancing decision, and increasing the load balancing frequency might make the overhead unacceptable. Second, there is overhead associated with moving work, which means that it is impractical to trace load changes that happen very quickly and trying to do so will result in unnecessary overhead. These two sources of overhead place an upper limit on how frequently load balancing should be performed. The scheduling granularity used by the operating system—the *time quantum*—also affects load balancing frequency selection, as we describe in more detail in Section 4.3.

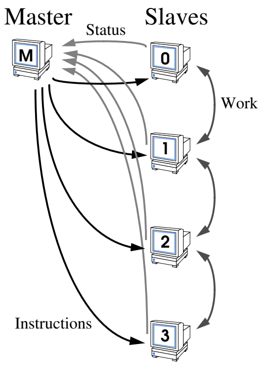
Synchronization required by the parallelized application also affects the load balancing decision making process. If synchronization occurs frequently, short-term skews in processing times accumulate and degrade performance. If possible, the code should be restructured, e.g., by strip mining, loop interchange, etc., to minimize the frequency of these synchronizations. Also, to avoid creating additional synchronization points, load balancing code should be placed at existing synchronization points in the parallelized application.

3 Load Balancer Design

In designing the load balancing system, some of the features are fixed at design time based on the knowledge of the application area and target system, and some decisions are made at compile time, startup time or run time so that application and dynamic environment characteristics can be considered. In this section we describe a load balancer that is appropriate for our application domain and target system. Its design is largely independent of whether the load balancer operates on automatically generated or hand-written



(a) Unrestricted



(b) Restricted by dependences

Figure 1: Communication for load balancing.

code. In the next section we focus on compiler and load balancing features that are needed to support automatically generated code.

3.1 High-level architecture

Load balancing is based on *global information*, allowing quicker response to fluctuations in system performance than if it were based on local information. Global knowledge allows the load balancer to instruct overloaded processors to move load directly to processors with surplus processing resources in a single step. We use a *central load balancing process* since it is simpler and sufficient for the relatively small number of nodes in our system. We call the central load balancing process the *master* and the processing nodes *slaves* (Figure 1).

Because the target is a distributed memory system, the cost of moving work and corresponding data back and forth from a central location to the slaves would become a bottleneck. Therefore, the *work is distributed among the slave processors*, and load balancing is done by shifting the work directly between the slaves (Figure 1a). Work movement may be constrained (Figure 1b) to reduce the cost of communications required by the parallelized application.

3.2 Load balancing algorithm

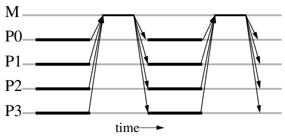
In our load balancing system, the slave processors periodically exchange messages with the load balancer. At selected load balancing points, the slaves send information about their performance since the last information exchange and receive instructions on how to redistribute work. At these points, the load balancer generates instructions for redistributing the distributed data in proportion to the slaves' relative processing capabilities, assuming that the remaining work associated with each data slice is equal. Slave performance is specified in work units per second, where the work units are iterations of the distributed loop. With this application-specific measure, there is no need to explicitly measure the loads on the processors or to give different weights to different processors in a heterogeneous processing environment. The load balancing algorithm relies on the assumption that previous computation rates are indicative of future computation rates [15] [13]. The algorithm has many similarities to the algorithm used by [13].

Using the information provided by the slaves, the load balancer calculates the aggregate computation rate of the entire system and computes a new work distribution where the work assigned to each processor is proportional to its contribution to the aggregate rate. The load balancer then compares the new work distribution to the current work distribution and computes instructions for redistributing the work which it sends to the slaves. For applications with loop-carried dependences, the instructions only move work between logically adjacent slaves so intermediate processors may be involved in a shifting of load (Figure 1b); this restriction is necessary to maintain a block distribution so that communication costs due to loop-carried dependences are minimized [15]. For applications without such restrictions, work may be moved directly between the source slave and the destination slave (Figure 1a). After redistributing work, the slaves continue computing their assigned iterations until the next information exchange.

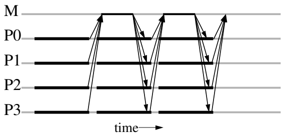
Several refinements are made to the above algorithm to prevent excessive work movement. First, to prevent oscillations and to reduce sensitivity to short load spikes, new rate information for each slave is filtered by averaging it with older rate information, with relative weights set according to trends observed in the rates. Second, work movement instructions are not generated unless the projected reduction in execution time is at least 10%, to account for work movement costs [13] and to avoid tracking normal system fluctuations. Finally, after work movement instructions are generated, a more detailed *profitability determination* phase [16] compares the estimated cost of the work movement with the projected benefits and cancels the work movement if it can not pay off.

3.3 Master-slave interactions

It is important to minimize the cost of interactions between the load balancer and the slaves since this overhead is incurred even if the system is well balanced. In the simple *synchronous* mechanism described in the previous section, each slave sends performance information to the load balancer and blocks waiting for instructions based on that information during each load balancing phase (Figure 2a). This mechanism responds immediately to measured changes in performance, but the entire interaction with the load balancer is placed in the critical path of the application.



(a) Synchronous



(b) Pipelined

Figure 2: Master-Slave interactions for load balancing.

Task	Section
Generate control for central load balancer	4.1
Determine grain size and block communication	4.4
Insert code in slaves for interaction with load balancer	4.2
Supply dependence information for restricting work movement	3.2
Generate application-specific routines for work movement	4.5
Generate code for arbitrary communication	4.6

Table 2: Compiler tasks in support of load balancing.

One way to reduce load balancing costs is by *pipelining*: at each load balancing point, the instructions received by the slaves are based on performance information sent at the previous load balancing point (Figure 2b). Pipelining prevents interactions with the central load balancer from becoming a bottleneck, especially as the number of slaves is increased, but delays the effects of load balancing instructions. In our target environment, network delays can vary significantly so we use pipelining to hide the costs of communication with the load balancer. Experiments comparing the pipelined and synchronous approaches confirm that pipelining is important.

4 Automatic load balancing

Our load balancing system consists of application-specific code generated by a parallelizing compiler and a common run-time library that supports task creation, communication, and load balancing. The compiler generates code for the master and slave processes. The application-specific code includes code for the computation, for input and output, for load balancing interactions, and for moving work.

This section discusses how the compiler supports dynamic load balancing and how the run-time system can adjust load balancing parameters automatically to improve performance. Table 2 summarizes the functions that must be added to a parallelizing compiler to support dynamic load balancing of the code it generates.

4.1 Preserving application knowledge

A central element of our approach is that *application information is preserved* implicitly in the generated code by retaining the loop structure of the sequential source code (e.g., Figure 3). Preserving this knowledge pays off in a number of ways. First, since task distribution is based on distributing loop iterations across the slaves, and iterations that operate on the same data will be executed on the same slave unless work is moved, we maximize data locality. Second, knowledge about the loop structure and data dependences makes it possible to reduce communication since work movement can be restricted to minimize the number of data dependences that cross processor boundaries. Finally, we exploit the fact that the tasks are loop iterations to minimize the cost of bookkeeping and task switching. Specifically, managing a "task queue" on a slave requires keeping track of a range of loop indices (i.e., two values), and task switching consists of incrementing a loop index. Context switching between tasks, i.e., loop iterations, is unnecessary since the entire context is captured in the loop structure.

Most of the code in the master is application-independent and can be included in the run-time library. However, the compiler must generate control code for the master that mimics the loop structure of the slaves so that both execute the same number of load balancing phases, and so that the program can terminate properly. For example, when the distributed loop is nested inside a data-dependent WHILE loop, the master must invoke the central load balancing code the correct number of times before receiving the data for testing the WHILE loop conditions.

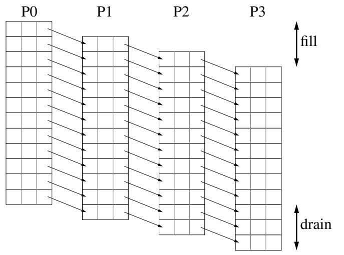
4.2 Placement of load balancing code

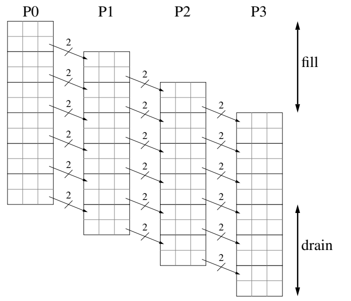
Load balancing hooks, conditional calls to the load balancing code, must be inserted at appropriate points in the parallelized code so that load balancing can occur periodically during the distributed computation. When the hook conditions are true, e.g., a counter has exceeded some value, the load balancing code measures the time spent in the computation, sends status information to the load balancer, receives instructions, and, if necessary, moves work between the processors. Hooks must be placed so that they do not add excessive overhead yet occur frequently enough for the load balancer to be responsive to load fluctuations. Figures 3b and 3c show possible hook insertion points (in bold) for a simplified version of SOR.

A compiler uses simple rules to place the load balancing hooks. First, if the distributed loop is an outermost loop, then a load balancing hook is inserted at the end of each iteration. If the distributed loop is an inner loop, the hook is placed at the deepest loop nesting level for which its cost is a negligible fraction (e.g., less than 1%) of the estimated

```
for (iter = 0; iter < maxiter; iter++) {
    for (i = 1; i < n - 1; i++) {
        for (j = 1; j < n - 1; j++) {
            b[j][i] = 0.493 * (b[j][i-1] + b[j-1][i] + b[j][i+1] + b[j+1][i]) + (-0.972) * b[j][i];
        }
    }
}
```

(a) Sequential






```
for (iter = 0; iter < maxiter; iter++) {
    if (pid != 0) send(left, &b[firstcol][0], n);
    if (pid != pcount-1) receive(right, &b[lastcol][0], n);
    for (i = 1; i < n - 1; i++) {
        if (pid != 0) receive(left, &b[firstcol-1][i], 1);
        for (j = firstcol; j < lastcol; j++) {
            b[j][i] = 0.493 * (b[j][i-1] + b[j-1][i] + b[j][i+1] + b[j+1][i])
                    + (-0.972) * b[j][i];
            lbhook2();              /* overhead too high.  */
        }
        lbhook1();                  /* ok.                 */
        if (pid != pcount-1) send(right, &b[lastcol-1][i], 1);
    }
    lbhook0();                      /* not frequent enough.*/
}
```

(b) Pipelined

```
for (iter = 0; iter < maxiter; iter++) {
    if (pid != 0) send(left, &b[firstcol][0], n);
    if (pid != pcount-1) receive(right, &b[lastcol][0], n);
    for (i0 = 0; i0 < n / blocksize; i0++) {
        if (pid != 0) receive(left, &b[firstcol-1][i0*blocksize], blocksize);
        for (i = i0 * blocksize; (i < (i0+1) * blocksize) && (i < n-1); i++) {
            for (j = firstcol; j < lastcol; j++) {
                b[j][i]= 0.493 * (b[j][i-1] + b[j-1][i] + b[j][i+1] + b[j+1][i])
                        + (-0.972) * b[j][i];
                lbhook2();          /* overhead too high.  */
            }
            lbhook1();              /* ok.                 */
        }
        lbhook1a();                 /* better control.     */
        if (pid != pcount-1) send(right, &b[lastcol-1][i0*blocksize], blocksize);
    }
    lbhook0();                      /* not frequent enough.*/
}
```

(c) Blocked and pipelined

Figure 3: Parallelization options for SOR (simplified version; error computation not shown). Figures show pipelined execution of a single relaxation phase. Code portions affected by strip mining and communication blocking are shaded.

execution time for the code between successive instances of the hook. The hook is placed either at the end of each iteration of the distributed loop (e.g., *lbhook2* in Figure 3b), immediately following the distributed loop (e.g., *lbhook1*), or immediately following a loop containing the distributed loop (e.g., *lbhook0*). The execution time for the hooks and the code between hooks can be estimated at compile time using a simple measure, such as instruction count, since only orders of magnitude need to be compared. For cases where loops with variable bounds are involved in the execution time estimate, the programmer can supply an approximation for the bounds or the compiler can make an educated guess. Loops can also be strip mined so that loop bounds can be controlled at run time to control the hook overhead. For example, in Figure 3c, *blocksize* can be adjusted to control the amount of code executed between instances of *lbhook1a*. If there is communication code at the same level of loop nesting as the hook, the hook position is shifted to a point next to the communication code so that load balancing does not add synchronization points.

4.3 Selection of load balancing frequency

For load balancing to be effective, the system must respond to performance changes as quickly as possible. However, as explained in Section 2.2, several factors in the execution environment limit the period between load balancings. Specifically:

- *load balancing interaction* – Collecting performance information and interacting with the load balancer adds overhead, even if the system is balanced. This overhead must be limited to a small fraction of the execution time. In our prototype over Nectar, the overhead is measured when the application starts, and the load balancing period is set so that the overhead is less than 5%.

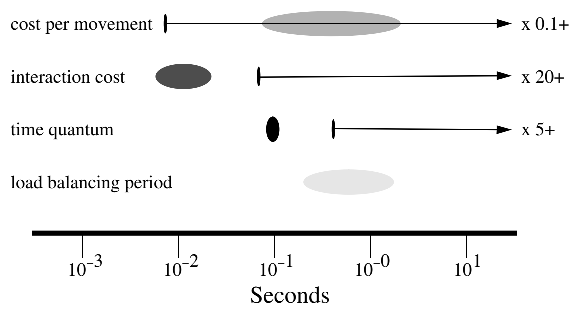


Figure 4: Periods affecting selection of load balancing frequency. The arrows and multipliers describe the acceptable ranges for the frequency based on the periods.

- *cost of moving work* – The cost of work movement limits how often it is profitable to move work [13]. Although it might be useful to track performance more frequently than work can be moved to react quickly to load changes, it does not pay off to take this to an extreme. Our system limits the load balancing period to no less than 0.1 times the cost of moving work. The cost of moving work is measured each time work is moved.

- *OS scheduling on the slaves* – The load balancing frequency should be selected so that the scheduling mechanism used by the operating system does not interfere with performance measurements. If the measurement period is near the time quantum for the system, context switching between processes will cause dramatic oscillations in the performance measurements causing work to be moved back and forth between slaves unnecessarily. To avoid this, the load balancing period must be at least several times the time quantum so that the context switching effects average out. Our system requires the load balancing period to be at least 5 scheduling quanta or 500 msec.

At run time, the target period between load balancings is set to the highest of these three lower bounds, as pictured in Figure 4. Each time the load balancer is invoked, it predicts the amount of computation that can be performed during the target period, using the most recent information about computation rates. Based on this prediction, the load balancer computes the number of hook instances that should be skipped before the next invocation of the load balancing code and includes this information in the instructions for the slaves. The placement of the hooks limits how closely the actual period between balancings approximates the target period: the more frequently hooks occur, the closer the actual period can be to the target period.

4.4 Controlling application granularity

Applications with loop-carried dependences, e.g., SOR (Figure 3), require communication for each iteration of the pipelined loop. For small iteration sizes, this communication overhead can dominate the execution time, and the resulting synchronization will significantly increase the execution time if there is any load imbalance. For example, if the iterations are smaller than the scheduling quantum used by the operating system, the execution times between synchronization points on slaves with competing tasks will vary widely, resulting in poor performance and making it difficult to accurately assess available processing resources so that load can be balanced.

These problems can be avoided by increasing the grain size of the computation through strip mining of the loop (Figure 3c). Communication can then be moved outside of the resulting inner loop, resulting in fewer messages after grouping communications with the same destination. These transformations reduce the frequency of synchronization, make execution times more predictable, and make communication more efficient. This optimization is not specific to load balanced code and can pay off on any system where communication is relatively expensive. The optimization is however more important on non-dedicated systems, since the effects of load imbalance are amplified by frequent synchronization. These restructuring transformations are done at the expense of parallelism because longer times are needed to fill and drain the pipeline, e.g., Figure 3b vs. Figure 3c.

This optimization is performed jointly by the compiler and the runtime system. The compiler does the strip mining, and the number of loop iterations in a block is set automatically at startup time based on measurements of the time it takes to execute several iterations of the loop. In our experimental system, the loop count is set to the number of iterations required to equal 150 milliseconds, 1.5 times the scheduling quantum.

4.5 Work movement

The compiler adds to each slave code to send and receive work based on the instructions received from the load balancer. The compiler knows the layout of the data in memory and can generate code to gather the necessary data when sending work and to scatter the data across the data structures when receiving work. In order to facilitate work movement, the compiler must modify the data structures compared with those used by a traditional parallelizing compiler. Distributed arrays are usually stored in distribution-major form (e.g. row major if the array is distributed by row) so that elements of a distributed data slice are contiguous. With load balancing, since the data a slave works on can change over time, an index array is needed to keep track of which data is local. In some cases, this index array causes an extra level of indirection for accessing the distributed data.

For pipelined applications, e.g., SOR, it is necessary to perform additional computations on data that is moved by load balancing, to make sure that the moved iterations are at a state consistent with the iterations on the destination processor. Iterations shifted to a slave from the slave to its left are one pipeline phase ahead of the local iterations and must be set aside until the local iterations catch up. Iterations shifted from the slave to its right are one pipeline phase behind the local iterations and must be caught up. The compiler is responsible for generating the code to set aside or update the received work.

4.6 Locating distributed data elements

If statements outside of distributed loops reference distributed data, communication may be required to access the data. With a fixed data distribution, a compiler can generate code that can compute the location of any distributed data element using information local to each processor. However, with a data distribution that changes at run time due to load balancing, processors can not compute data locations using local information only, and additional communication may be necessary. For example, when an element of a distributed data structure is to be moved into another distributed data location, the source and target slaves may be unknown to each other. A solution is to have the sender broadcast the data and to have all other slaves receive the data, but discard it unless they own the target location. The compiler generates appropriate communication code to handle the different types of assignment statements involving distributed data elements.

4.7 Adapting to changes in the application

In applications such as LU decomposition, the number of loop iterations for the distributed loop decreases with each iteration of the outermost loop, so each invocation of the distributed loop references one less slice of the data. Load balancing is concerned with work rather than data so it is undesirable to move data for which there is no remaining work. Distributed data slices are distinguished by labeling those with future work as *active* and those without as *inactive*. Whether data slices are active or inactive at a point during the execution of the program can be determined by the compiler from analysis of the loop bounds or with the aid of directives from the programmer. This information is used by the slaves at run time to determine which work units to move.

Another feature of LU decomposition is that the amount of work associated with the iterations of the distributed loop decreases each time the loop is invoked. The load balancer still correctly handles this case because for each invocation of the distributed loop, all of the iterations require the same amount of computation, and proportional allocation of iterations still works. However, as the computation progresses, the ratio of the cost of invoking the load balancer to the cost of executing a loop iteration increases; to compensate, the frequency of load balancing should be reduced as the size of work units decreases. The algorithm that selects the load balancing frequency (Section 4.3) makes this adjustment automatically.

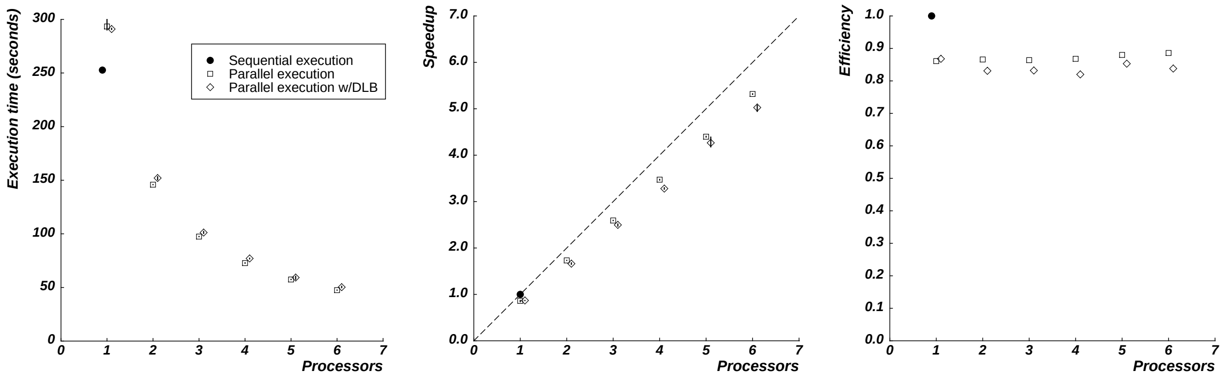
5 Measurements

We implemented a run-time system for automatically generated parallel programs for a loosely coupled system that includes the load balancing support discussed in this paper. Our target platform is the Nectar System [14], which consists of several Sun 4/330 workstations connected by a network providing 100 Mbyte/second links between any processors. We have evaluated the run-time system using hand-compiled (from Fortran to C) versions of the matrix multiplication (MM) and successive overrelaxation (SOR) applications mentioned earlier in the paper (Table 2.1).

5.1 Performance

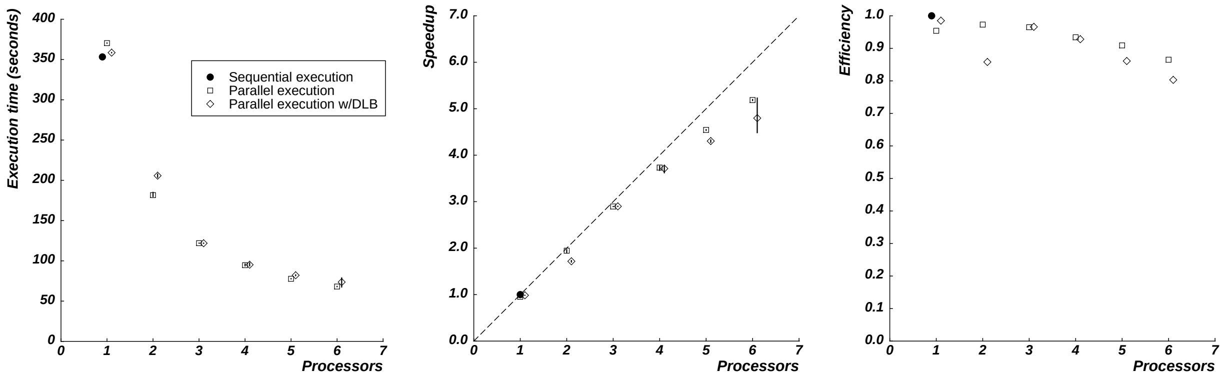
In an environment consisting of a homogeneous set of dedicated machines, load is balanced if work is distributed equally to the processors. Dynamic load balancing is not needed in such an environment, and should increase the execution time as little as possible. We evaluated the overhead of load balancing in a dedicated homogeneous environment by measuring the execution times with and without load balancing (Figures 5a and 6a) and calculating the speedups relative to the application running on a single processor (Figures 5b and 6b). For all measurements presented, each point represents the average of at least 3 measurements, and, when present, vertical bars show the range of the measurements. As desired, the overhead of load balancing is quite small, and load balancing has little effect on the speedup for the parallelized application.

When a constant competing load is added to one of the processors (processor 0 in our case), we expect work to be redistributed immediately and then to stabilize. Comparing execution times (Figures 7a and 8a) does not provide enough information to distinguish load balancing overheads and benefits because the execution times include time spent on competing tasks. Instead, we evaluate load balancing effectiveness using the *efficiency* of resource usage, i.e., the total CPU time spent on productive computation divided by the sum of the available CPU times on the slave processors. On a homogeneous set of processors, the productive computation time is the time taken by the sequential



(a) Computation time (b) Speedup (c) Efficiency

Figure 5: 500 $\times$ 500 matrix multiplication running in dedicated homogeneous environment.



(a) Computation time (b) Speedup (c) Efficiency

Figure 6: 2000 $\times$ 2000 successive overrelaxation running in dedicated homogeneous environment.

version of the program on a dedicated machine. The available CPU time on each processor is the elapsed time minus the CPU time spent on competing tasks (measured using the *getrusage* function provided by UNIX).

$$efficiency = \frac{time_{sequential}}{time_{elapsed} * processors - \sum time_{competing}}$$

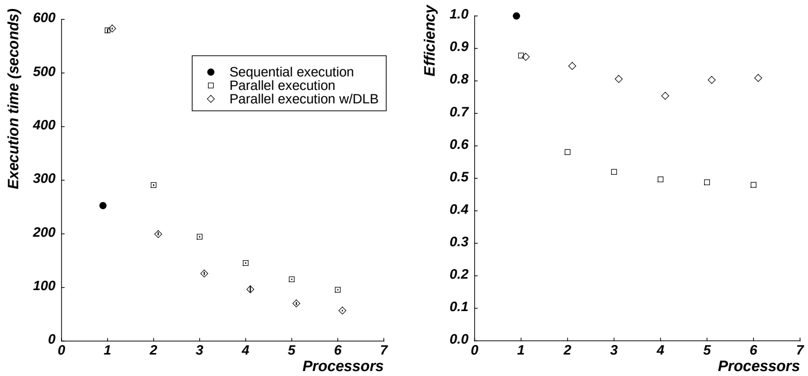
Even under ideal conditions, i.e., in the homogeneous, dedicated environment, the efficiency will be less than one as a result of communication and synchronization that is present in the distributed application. However, we would like load balancing to increase the efficiency, and bring it as close as possible to the ideal case. Figures 5c and 6c show efficiencies computed for the dedicated homogeneous environment. With one loaded processor (Figures 7b and 8b), the efficiencies for the load balanced applications are slightly lower than in the dedicated environment, but still higher than those of the applications without load balancing.

5.2 Work movement in response to load changes

Figure 9 shows how the work assigned to a slave processor follows the changes in its computation rate. A 500×500 matrix multiplication is run on a 4-slave system with an oscillating load on one slave; the remaining slaves have no competing loads. The graph shows the raw rate information, the adjusted rate information after filtering, and the work assignment for the slave with the competing load. The work assignment tracks the available processing power but lags behind by two load balancing periods: one for the load balancer to respond, and the other due to pipelining of the master-slave interactions. The lag is greater when the rate drops because the time between load balancing hooks increases as the rate decreases, temporarily lengthening the load balancing period.

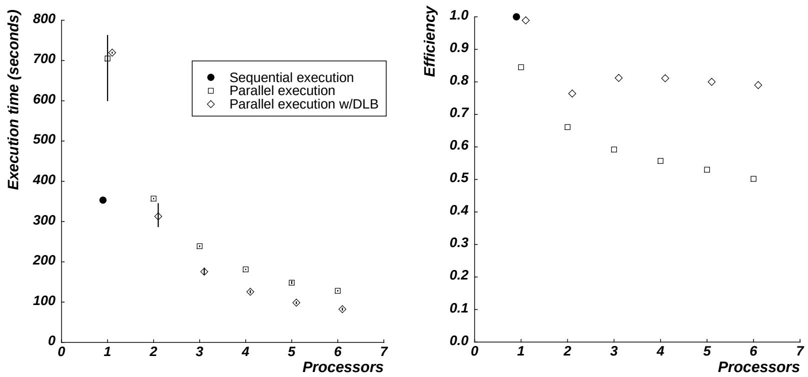
6 Related work on load balancing

Many of the approaches for scheduling of iterations of distributed loops are *task queue models* targeting shared



(a) Computation time (b) Efficiency

Figure 7: 500 × 500 matrix multiplication running in homogeneous environment with constant load on one processor.



(a) Computation time (b) Efficiency

Figure 8: 2000 × 2000 successive overrelaxation running in homogeneous environment with constant load on one processor.

memory architectures. Work units are kept in a logically central queue and are distributed to slave processors when the slave processors have finished their previously allocated work. Most of this research in this area, called *self-scheduling*, is concerned with minimizing the overhead of accessing this central queue while minimizing the skew between the finishing times of the processors [8] [7] [9] [10]. Information regarding interactions between the iterations is often lost due to the desire to have a single list of tasks.

Recent research [11] [12] has added consideration for *processor affinity* to the task queue models so that locality and data reuse are taken into account: iterations that use the same data are assigned to the same processor unless they need to be moved to balance load. In *Affinity Scheduling* [11], data is moved to the local cache when first accessed, and the scheduling algorithm assigns iterations in blocks. In *Locality-based Dynamic Scheduling* [12], data is initially distributed in block, cyclic, etc. fashion, and each processor first executes the iterations which access local data. Both of these approaches still assume a shared memory environment.

In *diffusion models*, targeted at tightly coupled distributed memory machines, all work is distributed to the processors at startup, and work is shifted between adjacent processors when processors detect an imbalance between their load and their neighbors'. The load balancing may be based on near-neighbor information only [16] or may be based on global information propagated through the processors [17]. These approaches usually assume independent loop iterations.

For the implementation of Dataparallel C on a network of workstations [13], loop iterations are mapped to virtual processors, and virtual processors are shifted between processors to balance load. As in our approach, relative computation rates are assessed periodically, and work is redistributed to processors in proportion to their rates. However, Dataparallel C requires the programmer to handle the program partitioning and communication explicitly; this may make pipelined execution of loops complicated to implement. Also, the virtual processor abstraction may add run-time overhead, and all processors communicate for load balancing so load balancing communication is in the critical

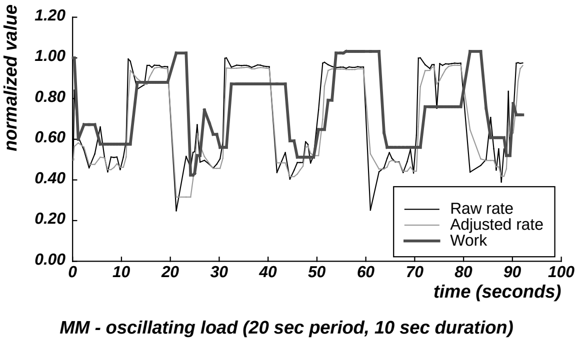


Figure 9: Measured rate (normalized with maximum), filtered rate, and work assignment (normalized with equal distribution assignment) for slave with oscillating load.

path for the computation.

Our work differs from previous approaches in that we explicitly consider application data dependences and loop structure in the load balancer. Also, we support automatically generated code and use compiler-provided information as well as run-time information to set load balancing parameters.

7 Conclusions

Parallelizing compilers for networks of processors that are shared with other users must generate efficient code that supports dynamic load balancing. In this paper we presented an architecture for a system that supports the automatic generation of parallel programs with dynamic load balancing. The concept is that the parallelizing compiler includes calls to a run-time load balancer in the generated code, taking advantage of application-specific features so that data locality and data reuse are maximized to minimize communication costs. Existing compilers are already capable of identifying most of the required application features. We identified the additional code the compiler has to generate and have presented an algorithm to determine placement of calls to load balancing routines. The runtime library relies on compiler-provided information and runtime measurements to select load balancing parameters.

We implemented a run-time library that supports automatically generated parallel programs and that includes load balancing. We presented results for two scientific routines with different application features that affect load balancing. Our measurements demonstrate that load balancing overhead can be kept low by proper adjustment of load balancing parameters, and that the load balancer can rapidly adjust the work distribution in a heterogeneous environment. Our results also show that techniques that overlap load balancing with computation are effective in reducing load balancing overhead.

References

[1] M. Rosing, R. B. Schnabel, and R. P. Weaver, "The Dino parallel programming language," *Journal of Parallel and Distributed Computing*, vol. 13, pp. 30–42, Sep. 1991.

[2] M. J. Quinn and P. J. Hatcher, "Data-Parallel Programming on Multicomputers," *IEEE Software*, vol. 7, pp. 69–76, Sep. 1990.

[3] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD Distributed-Memory Machines," *Communications of the ACM*, vol. 35, pp. 66–80, Aug. 1992.

[4] P.-S. Tseng, "A Parallelizing Compiler for Distributed Memory Parallel Computers," Ph.D. Thesis CMU-CS-89-148, ECE Department, Carnegie Mellon University, May 1989.

[5] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross, "Exploiting Task and Data Parallelism on a Multicomputer," in *Proc. of PPoPP*, pp. 13–22, May 1993.

[6] H. Zima, P. Brezany, B. Chapman, P. Mehrota, and A. Schwald, "Vienna Fortran – A Language Specification Version 1.1," Tech. Rep. ACPC/TR 92-4, Austrian Center for Parallel Computation, Mar. 1992.

[7] C. D. Polychronopoulos and D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Trans. on Computers*, vol. C-36, pp. 1425–1439, Dec. 1987.

[8] C. D. Polychronopoulos, "Toward Auto-scheduling Compilers," *The Journal of Supercomputing*, vol. 2, no. 3, pp. 297–330, 1988.

[9] S. F. Hummel, E. Schonberg, and L. E. Flynn, "Factoring: A Practical and Robust Method for Scheduling Parallel Loops," in *Supercomputing '91 Proceedings*, pp. 610–619, IEEE Computer Society Press, Nov. 1991.

[10] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, pp. 87–98, Jan. 1993.

[11] E. P. Markatos and T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors," in *Supercomputing '92 Proceedings*, pp. 104–113, IEEE Computer Society Press, Nov. 1992.

[12] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, "Locality and Loop Scheduling on NUMA Multiprocessors," in *Proc. of the 1993 Int'l Conference on Parallel Processing*, pp. II-140–II-147, CRC Press, Inc., Aug. 1993.

[13] N. Nedeljković and M. J. Quinn, "Data-Parallel Programming on a Network of Heterogeneous Workstations," in *Proc. of the First Int'l Symposium on High-Performance Distribution Computing*, pp. 28–36, IEEE Computer Society Press, Sep. 1992.

[14] E. Arnould, F. Bitz, E. Cooper, H. T. Kung, R. Sansom, and P. Steenkiste, "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers," in *ASPLOS-III Proceedings*, pp. 205–216, ACM/IEEE, Apr. 1989.

[15] H. Nishikawa and P. Stenkiste, "Aroma: Language Support for Distributed Objects," in *Proc. of the Sixth Int'l Parallel Processing Symposium*, pp. 686–690, IEEE Computer Society Press, Mar. 1992.

[16] M. Willebeek-LeMair and A. P. Reeves, "Dynamic Load Balancing Strategies for Highly Parallel Multicomputer Systems," Tech. Rep. EE-CEG-89-14, Cornell Univ. Computer Engineering Group, Dec. 1989.

[17] F. C. H. Lin and R. M. Keller, "The Gradient Model Load Balancing Method," *IEEE Trans. on Software Engineering*, vol. SE-13, pp. 32–38, Jan. 1987.