Dynamic Adaptation to Available Resources for Parallel Computing in an Autonomous Network of Workstations *

Umit Rencuzogullari, Sandhya Dwarkadas

Department of Computer Science
University of Rochester
Rochester, NY 14627–0226
umit,sandhya@cs.rochester.edu

ABSTRACT

Networks of workstations (NOWs), which are generally composed of autonomous compute elements networked together, are an attractive parallel computing platform since they offer high performance at low cost. The autonomous nature of the environment, however, often results in inefficient utilization due to load imbalances caused by three primary factors: 1) unequal load (compute or communication) assignment to equally-powerful compute nodes, 2) unequal resources at compute nodes, and 3) multiprogramming. These load imbalances result in idle waiting time on cooperating processes that need to synchronize or communicate data. Additional waiting time may result due to local scheduling decisions in a multiprogrammed environment. In this paper, we present a combined approach of compile-time analysis, run-time load distribution, and operating system scheduler cooperation for improved utilization of available resources in an autonomous NOW. The techniques we propose allow efficient resource utilization by taking into consideration all three causes of load imbalance in addition to locality of access in the process of load distribution. The resulting adaptive load distribution and cooperative scheduling system allows applications to take advantage of parallel resources when available by providing better performance than when the loaded resources are not used at all.

1. INTRODUCTION

Networks of workstations (NOWs), which are generally composed of autonomous compute elements (whether uniprocessors or symmetric multiprocessors (SMPs)) networked together, are an attractive parallel computing platform since they offer high performance at low cost. There are several factors that interfere with the efficient utilization of autonomous NOWs for parallel computing. The parallelization strategy used by an application could result in load imbalances due to unequal load assignment or excess communication among some nodes. In addition, load imbalances can arise due to hardware inequalities. It is very likely that a NOW is made up of old and new hardware as machines are upgraded. Finally, multiprogramming in combination with independent scheduling decisions on each of the nodes can result in additional slowdown of a parallel application trying to take advantage of the distributed autonomous resources.

The default programming paradigm that is supported in hardware is message passing across the nodes, and shared memory among processes within a node. Unfortunately, the message passing paradigm requires explicit communication management by the programmer or parallelizing compiler. This communication management can be very complex, especially for applications with dynamic access patterns, or for multiprogrammed platforms or platforms with unequal resources. The most efficient workload and communication schedule can be impossible to predict statically.

An alternative programming paradigm is software-based distributed shared memory (SDSM). An SDSM protocol (e.g., [3, 17, 26]) provides the illusion of shared memory across a distributed collection of machines, providing a uniform and perhaps a more intuitive programming paradigm. A shared memory paradigm provides ease-of-use and additionally leverages an SMP workstation's available hardware coherence to handle sharing within the SMP. SDSM has been shown to be an effective target for a parallelizing compiler [5]. Since data caching and communication is implemented by the run-time system, compile-time complexity is reduced. Previous work [7, 19] has integrated compile-time information within the run-time system in order to improve performance. Access patterns from the compiler are used by the run-time system to optimize communication, providing a significant improvement in performance. Our goal is to leverage the flexibility afforded by the SDSM system to effect load balancing in autonomous environments.

Load balancing and/or locality management has been extensively studied by many researchers especially in the context of loop scheduling [12, 18, 27, 23, 15, 8]. All these studies deal with the issue of either load balance or locality or both but not with scheduling issues. Ioannidis et al. [12] propose a method of assigning loops to each of the pro-

*This work was supported in part by NSF grants EIA-9972881, EIA-0080124, CCR–9702466, CCR-9988361, and CCR–9705594; by an external research grant from DEC/Compaq; and by DARPA/AFRL contract number F29601-00-K-0182.

PPOPP'01, June 18-20, 2001, Snowbird, Utah, USA.

cesses based on the observed relative power and the locality of data (in order to minimize steady-state communication). Lowenthal et. al. [18, 11] use a global strategy for optimizing the execution path through the data distribution graph of a program, which is executed at runtime and directed by the number of incurred page faults and computation time of each parallel region. The other studies mentioned suggest various heuristics for selecting tasks from a task queue. Task queue techniques inherently assume a tightly coupled environment where synchronization is fairly inexpensive compared to the amount of computation. In addition, none of these studies address the problem of communication delay introduced by scheduling in a multiprogrammed environment.

On the other hand, many researchers have considered the problem of scheduling a parallel application in a parallel or distributed system. The goal here is essentially to reduce communication delay by ensuring that cooperating processes are scheduled at the same time [22, 25, 6, 28]. Ousterhout [22] shows that coscheduling is desirable and describes different algorithms for accomplishing it. Ousterhout's work assumes that scheduling decisions are made centrally, by a single operating system running across all processors. Sobalvarro et. al. [25] and Dusseau et. al. [6] assume a loosely coupled system and make scheduling decisions based on communication patterns. Tucker et. al. [28] propose keeping the number of runnable processes of a single application equal to the number of available processors.

We argue that neither pure load balancing techniques nor the scheduling mechanisms proposed thus far allow for efficient use of autonomous NOWs. Load balancing and locality management techniques are effective when the imbalance is caused by different hardware or disproportionate data distribution, but they tend to be ineffective in multiprogrammed environments. Wait time of a sender or receiver could be as high as one scheduling quantum (the time between consecutive context switches) for a message and significantly more for synchronization. Furthermore, as the number of cooperating processes increases, the wait time may increase linearly in multiprogrammed situations. Coscheduling techniques, on the other hand, assume equal levels of multiprogramming and work on all nodes. In the presence of unequal levels of multiprogramming, trying to get the same proportion of the processor from an overloaded node would result in either starvation for other processes on the overloaded node or wasted processor cycles on an underloaded node.

The solution we propose for effective utilization in the presence of load imbalance is a combination of compile-time program analysis, runtime load balancing in combination with SDSM, and operating system scheduling support to reduce communication and synchronization delay. At compile time, we analyze the program to capture the access patterns and instrument the code with calls to the runtime library. Our static instrumentation feeds loop and access information to our runtime system. The runtime system uses this information to partition the available work based on locality of data access as well as resource availability. The runtime library is the glue between the application and the operating system, continuously monitoring ongoing activity and making decisions as to when to willingly yield the processor in order to give the scheduler more flexibility. The operating system responds to application-specific information on scheduling needs while respecting fairness. The operating system also provides feedback to the application about the scheduling status of cooperating processes, allowing the runtime to make resource management decisions based on this information.

The remainder of the paper is organized as follows. Section 2 describes our approach to the problem, Section 3 presents our experimental setup and an evaluation of our system, and finally Section 4 concludes and discusses future work.

2. DESIGN AND IMPLEMENTATION

As mentioned in Section 1, our programming environment is software distributed shared memory (SDSM). The use of SDSM reduces the complexity of the compiler (or hand-coded) parallelization, since communication is managed by the runtime system. Our system uses a combination of static program analysis, runtime monitoring and load redistribution, and operating system scheduling support in order to intelligently maximize available resource utilization. We elaborate on each of the components in the following subsections.

2.1 The Base Software DSM Runtime System

Our run-time system, Cashmere-2L (CSM) [26], is a page-based software DSM system that has been designed for SMP clusters connected via a low-latency remote-write network. The system implements a multiple-writer [4], "moderately" lazy release consistency protocol [14], and requires applications to adhere to the data-race-free, or properly-labeled, programming model [1]. Effectively, the application is required to use explicit synchronization to ensure that non-local changes to shared data are visible. The moderately lazy characteristic of the consistency model is due to its implementation, which lies in between those of TreadMarks [3] and Munin [4]. Invalidations in CSM are sent during a release and take effect at the time of the next acquire, regardless of whether they are causally related to the acquired lock.

A unique point of the CSM design is that it targets low-latency remote-write networks, such as DEC's Memory Channel [9]. These networks allow processors in one node to directly modify the memory of another node safely from user space, with very low (microsecond) latency. CSM utilizes the remote-write capabilities to efficiently maintain internal protocol data structures. As an example, CSM uses the Memory Channel's remote-write, broadcast mechanism to maintain a replicated *directory* of sharing information for each page (i.e., each node maintains a complete copy of the directory). The per-page directory entries indicate who the current readers and writers of the page are.

Under CSM, every page of shared data has a single, distinguished home node that collects modifications at each release, and maintains up-to-date information on the page. Initially, shared pages are mapped only on their associated home nodes. Other nodes obtain copies of the pages through page faults, which trigger requests for an up-to-date copy of the page from the home node. Page faults due to write accesses are also used to keep track of data modified by each node, for later invalidation of other copies at the time of a release. If the home node is not actively writing the page, then the home node is *migrated* to the current writer by simply modifying the directory to point to the new home node. If there are readers or writers of a particular page on a node other than the home node, the home node downgrades its

writing permissions to allow future possible migrations. As an optimization, however, we also move the page into *exclusive* mode if there are no other sharers, and avoid any consistency actions on the page. Writes on non-exclusive and non-home-node pages result in a *twin* (or pristine copy of the page) being created. The *twin* is later used to determine local modifications.

As mentioned, CSM was also designed specifically to take advantage of the features of clusters of SMPs. The protocol uses the hardware within each SMP to maintain coherence of data among processes within each node. All processors in a node share the same physical frame for a shared data page. The software protocol is only invoked when sharing spans nodes. The hardware coherence also allows software protocol operations within a node to be coalesced, resulting in reduced data communication, as well as reduced consistency overhead.

2.2 Static Program Analysis

We use static program analysis to identify the access pattern of our parallel program as well as to insert the library hooks that monitor the process activity and cooperate with the operating system (OS). Once a parallel region is identified, there are two dimensions along which load distribution decisions can be made. The first is the amount of work per subtask (where a subtask is identified as the smallest independent unit of work that can be performed in parallel, e.g., a single iteration of a parallel loop). The second is the data accessed by each subtask. For many regular access patterns, the compiler can identify the data accessed by each parallel loop. In addition, the compiler can also attempt to predict whether each parallel loop performs the same or different amounts of work. Our static analysis [13] provides information on the above two dimensions wherever possible.

We illustrate the interface between the compiler and the runtime, as well as the information extracted by the compiler, through an example parallel loop. Figure 1 shows pseudo-code for the original sequential loop. There are several pieces of information that the compiler supplies to the runtime. For every shared data structure, the compiler initializes data structures indicating its size and the number and size of each dimension. In addition, for each parallel region, the compiler supplies information regarding the shared data accessed (in the form of a regular section [10]) per loop (or subtask) in the parallel region. The loop is then transformed as shown in the pseudo-code in Figure 2. In reality, much of the information passed to the runtime task partitioner is initialized in static data structures, with only those variables that change on each invocation being updated.

```
int    sh_dat1[N], sh_dat2[N];

for (i = lowerbound; i < upperbound; i += stride)
    sh_dat1[a*i + b] += sh_dat2[c*i + d];
```

Figure 1: Initial parallel loop. Shared data is indicated by `sh_`.

Once the information on the loop bounds and array dimensions is available, the amount of computation and the locality of access can be deduced (heuristically) for several important classes of applications. For instance, detecting that the amount of work per parallel loop is a function of

```
int    sh_dat1[N], sh_dat2[N];

Initialize
    parallel loop identifier,   /* i           */
    list of shared arrays, /* sh_dat1, sh_dat2 */
    list of types of accesses,  /* read/write  */
    list of lower bounds,       /* lower_bound */
    list of upper bounds,       /* upper_bound */
    list of strides,            /* stride      */
    list of coefficients and
    constants for array indices /* a, c, b, d  */

taskSet = partition_tasks( );

while there are Tasks in the taskSet
    lowerbound = new lower bound for that Task;
    upperbound = new upper bound for that Task;
    stride = new_stride;

    for (i = lowerbound; i < upperbound; i += stride)
        sh_dat1[a*i + b] += sh_dat2[c*i + d];
```

Figure 2: Parallel loop with added pseudo-code that serves as an interface with the run-time library. The run-time system can then change the amount of work assigned to each parallel task.

the parallel loop index implies that in order to achieve a balanced distribution of load while preserving locality of access, a cyclic distribution of the parallel loops would be useful[1]. Similarly, detecting a non-empty intersection between the regular sections of adjacent parallel loops implies a stencil-type computation with nearest-neighbor sharing, while detecting an empty or loop-independent intersection among loops implies loop-independent sharing.

Two variables in the data structure for each parallel region encode this information — *load* and *access*. *load* is currently defined to be one of *FIXED* or *VARIABLE*, the default being *FIXED*. A *VARIABLE* load type is currently used as an indication to use a cyclic load distribution, while a *FIXED* load type is used as an indication to use a block load distribution. *access* is currently defined to be one of *STENCIL* or *INDEPENDENT*. *access* is used to influence the type of load distribution used, and to determine the type of redistribution used. *access* can potentially be updated by the runtime based on information about data currently cached by the process. An access type of *STENCIL* is treated as a signal to use a blocked load distribution as well as a blocked re-assignment of load (i.e., load is re-assigned by shifting loop boundaries in proportion to the processing power of the individual processors). Using this type of load re-assignment minimizes steady-state communication due to nearest-neighbor sharing. However, the redistribution results in data being communicated among all neighboring processors during each redistribution. An access type of *INDEPENDENT* signals the ability to minimize this communication by assigning a loaded processors' tasks directly

[1] In the presence of conditional statements, variable load per parallel loop cannot always be detected at compile-time. Application-specific knowledge could also be easily encoded by the user.

to the lightly loaded processors. Since data sharing among loops is iteration-independent, there is no resulting increase in steady-state communication.

For source-to-source translation from a sequential program to a parallel program that is compatible with our runtime library, we use the Stanford University Intermediate Format (SUIF) [2] compiler. The SUIF system is organized as a set of compiler passes built on top of a kernel that defines the intermediate format. Each of these passes is implemented as a separate program that reads its input from a file and writes its output to another file. SUIF files always use the same format.

We added two passes to the SUIF system for our purposes. The first pass works before the parallel code generation and inserts code that provides the runtime library with information about each parallel region's access patterns. The second pass works on parallelized programs and modifies the loop structure so that a task queue is used.

The standard SUIF distribution can generate a single-program, multiple-data (SPMD) program from sequential code for many simple loops but lacks the more complex transformations essential to extract parallelism from less easily analyzable loops. While our SUIF passes provide an easy translation mechanism for many programs, it is straightforward to insert the required data structures by hand into an already parallelized program.

2.3 Relative Processing Power

As described in [13], in order to partition the load according to available resources, we need to be able to estimate the available computational resources and communication overheads. Intuitively, elapsed time is a good measure of the available resources, since it captures a processor's perceived load, whether due to processor speed, multiprogramming, or contention for memory and/or the network. Each process queries the operating system to obtain the elapsed time for the execution of each parallel region. The `RelativePower` of a processor is inversely proportional to this time, i.e., the faster the processor, the larger its power relative to the other processors. We track time over several parallel regions prior to updating the relative powers in order to smooth out transient spikes in performance. The high-level algorithm for computing relative processing power is shown in Figure 3. `TaskTime` is a shared array that is updated prior to computing the relative powers. `RelativePower` is initialized according to the task distribution strategy used and is always normalized.

2.4 Task Distribution Strategy

On the first execution of any parallel region, the initial load assignment made by the runtime is guided by the information provided by the static analysis, the currently perceived relative power of each processor, and the size of data elements as well as the size of the coherence unit. The information provided by the static analysis includes loop boundaries, size of data elements, and predicted access patterns. The runtime library keeps statistics about the perceived compute power of each processor (which could change over time based on the level of multiprogramming) and makes initial assignments proportional to the perceived compute power.

Task assignment and execution takes the topology of the processors into account. For a network of SMPs, work is

```
float RelativePower[NumOfProcessors];
    // Initialized at program start
    // to 1/NumOfProcessors
float TaskTime[NumOfProcessors];
    // Execution time of parallel region
float SumOfPowers=0;

// Calculate new RelativePower
for  all Processors i
    RelativePower[i] /= TaskTime[i];
    SumOfPowers += RelativePower[i];
endfor

// Normalize based on sum of the RelativePowers
// to ensure the sum of powers is 1.0
for  all Processors i
    RelativePower[i] /= SumOfPowers;
endfor
```

Figure 3: Algorithm to determine relative processing power.

partitioned in a hierarchical manner in order to account for the fact that intra-node communication is cheaper than inter-node communication. Task redistribution is performed across SMPs. Task stealing is allowed within each SMP. Locality has been shown to be more important than load balancing [20]. Given the continuously increasing speed gap between processors and memory and the use of deeper memory hierarchies, locality management is an even bigger issue in today's processors. In order to preserve locality within an SMP, each processor maintains task affinity — it must finish its own task assignment prior to stealing a task from another processor (similar to [16], except using fixed-size tasks). This is done by using a per-processor task queue, and having a processor retrieve tasks from the head of its queue but steal from the tail of another processor's queue. Once a task is stolen from another processor's task queue, it is moved and owned by the stealing processor. Using a per-processor task queue (rather than a shared one for all processes on a multiprocessor node) helps not only by maintaining locality but also by reducing contention for the lock to access the shared task queue.

The runtime library partitions the parallel region into tasks based on the access pattern, the load per parallel loop, and the size of the coherence unit. The size of the data elements along with the size of the coherence unit are used to determine the partitioning in an attempt to reduce false sharing. Work is partitioned so that accesses by each individual process are in multiples of the coherence unit in order to avoid false sharing across processors. Consecutive loop iterations are blocked together until the data accessed is the least common multiple of the coherence unit and the data accessed per loop. This defines the minimum task size. Once the minimum task size has been determined, a fixed number of tasks per parallel region are created and assigned to processors using either a block or cyclic distribution based on whether the load is defined to be *FIXED* or *VARIABLE*, respectively, or whether the access pattern is *STENCIL*. The size of each task is an integral multiple of the minimum task size and enough tasks are created to allow later redistribution when relative processing powers change.

The goal of the runtime library is to dynamically determine and partition work in a locality-preserving manner based on the computation to communication ratio. Partitioning work to be performed on a single coherence unit will result in excessive communication due to false sharing, unless the amount of computation per data unit is high relative to the communication cost. If all the data to be processed by the loop resides on a single node and the parallel region's execution time is shorter than the time to communicate the data, work is performed on the node that caches the data. Within the node, the work is distributed among the processes since intra-node communication is fairly inexpensive.

Similarly, the task distribution strategy attempts to preserve locality across parallel regions. If the same array is accessed across multiple parallel regions, a data structure is maintained for the array to indicate the `current` partitioning that has been used. If a parallel region accesses the same shared array and `approximately` the same data range within that array (determined by the task size chosen for the region since the runtime creates a fixed number of tasks per parallel region), the same partitioning is used. If the access patterns in the two regions vary considerably, communication is assumed to be large due to the necessary redistribution, and a partitioning is chosen appropriately. At the same time, the `current` partitioning is updated. This strategy is localized, requiring less computation than the more global strategy proposed in [11].

Reassignment of tasks occurs when a significant change in the `RelativePower` is detected[2]. Task redistribution is performed on the basis of locality. For a *STENCIL* access pattern, tasks are redistributed among neighboring processors in order to minimize steady-state communication. For an *INDEPENDENT* access pattern, tasks are redistributed by reallocating tasks from the slowest processors to the faster processors. For *INDEPENDENT* access patterns, this type of reallocation will result in the minimum overall communication.

2.5 Cooperative Scheduling Support

Multiprogramming adds an additional dimension to the problem of imbalanced load. Communication among cooperating processes can result in significant additional overhead because of waiting time caused by one of the processes being descheduled and unable to respond. Coscheduling [22, 25, 6, 28, 21] approaches have been used in the past, where cooperating processes are scheduled to execute simultaneously on all processors. This approach is useful when the load on all processors is equal. However, in the presence of autonomous nodes with unequal levels of multiprogramming at each processor, a more distributed and cooperative approach is required in order to improve efficiency while retaining autonomy.

Our goal is to improve the response times seen by parallel applications in the presence of multiprogramming through the use of a cooperative scheduler. We modified a priority-based scheduler to achieve this goal while retaining the fairness and autonomy of the individual schedulers on each node. The cooperative interface has several components. Our implementation is on Compaq's Tru64 (formerly known as DEC Unix) version 4.0F.

[2]The difference between the smallest and largest `RelativePower` must differ by more than a hysteresis factor from its previous value (10% in our case).

2.5.1 Scheduler Modifications

In order to improve response times, the scheduler must be willing to schedule an application's process on demand. However, this cannot be accomplished in traditional schedulers without compromising fairness. To provide the scheduler with the flexibility to handle these conflicting requirements, each process, upon declaration of its interest in cooperating with remote processes, is charged a scheduling quantum of time. This time is held in a "*piggy-bank*" for future use by the process. The "*piggy-bank*" is replenished any time the process voluntarily yields the processor prior to the expiration of its scheduling quantum (by adding an amount less than or equal to the remainder of the quantum), but is guaranteed not to grow larger than one scheduling quantum of time. This guarantee prevents a process from taking over the processor for long periods of time by yielding often. The process is pre-charged for the amount of time put into the piggy-bank rather than being charged when the piggy-bank is used. The scheduler can then use the time in the piggy-bank to schedule the process on request. Such a request is honored only if the process has some balance in its piggy-bank. The time used by the process from the piggy-bank is subtracted and the balance is saved for future use.

2.5.2 OS-Runtime Interface

For a parallel application to request the cooperation of the scheduler, information about the process's scheduling state is required. One way of providing this information is through a variable shared between the operating system and the application. In addition, once the scheduling status of a process is determined, remote processes need a mechanism to wake up the process if it is not currently running.

We provide a system call that allows each process to register a memory location and a signal. The registered memory location has two words. The first ("scheduling status") is written by the operating system and provides feedback to cooperating processes regarding the scheduling status of the registering process. It is set by the operating system when the process is descheduled. The second ("signaled") can be written by cooperating processes and is used to avoid excessive signaling overhead. It is set by a cooperating process when a signal has been sent to this particular process for the purposes of being woken up.

The registered signal is used by a cooperating process as a wakeup signal. Upon receiving the signal, rather than delivering it, if the operating system can schedule the process while continuing to guarantee fairness using the piggy-bank (i.e., if resource utilization is within limits), it does so. When the process is scheduled, both words of the registered memory are reset, indicating that the process is scheduled and no signal is pending.

Ideally, the values written to the registered memory locations must be available to all cooperating processes. Hence, these memory locations are placed in shared memory and broadcast to all sharers. Since our network is a memory-mapped, low latency remote write network, these memory locations are mapped into the network address space and the scheduler accesses them using ordinary reads and writes. We believe the additional communication overhead resulting from this sharing is minimal in comparison to the rest of the protocol and data communication overhead for the application. This is especially true for the medium-scale clusters used for such parallel applications.

2.5.3 Application Cooperation

In order to give the scheduler the flexibility to respond to on-demand scheduling requests, an application must voluntarily yield the processor in order to build up its piggy-bank. A yield system call[3] is used to free up resources preemptively in order to build up this future "equity".

The yield system call takes one argument to indicate the lowest priority that the caller is willing to yield to. The argument specifies a priority relative to the priority of the caller. If no other process within the given priority is available, the call returns immediately with no effect. Otherwise, a runnable process with the highest priority is picked and scheduled. A complication in implementing this system call is accounting for resource usage. In many operating systems, processes are charged at the granularity of a clock-tick, which is about 1 msec on Tru64 Unix (our experimental platform). If a process yields frequently enough, it is fairly easy for that process either not to be charged at all for its use or to be over-charged depending on the relative timing of a clock tick and the scheduling event. In order to fix this problem, hardware counters are used as the basis for accounting.

The yield call is made by a process whenever the process would otherwise spin waiting for an external event such as communication or synchronization with a remote process. A spin-block strategy [21] is used in order to avoid unnecessary yields. The spin time is set to be at least twice the round-trip communication time and is doubled each time the yield is unsuccessful.

It is important to keep in mind that, unless the newly added features are used by the applications, the overhead of the new scheduling is negligible. Even when these features are exploited extensively, the overhead is still minimal. The only overhead incurred is to set/clear the registered memory locations at each context-switch and keep some extra accounting information about resource usage.

3. EVALUATION

3.1 Experimental Platform

Our experimental environment is a cluster of Compaq AlphaServer 4100 servers. Each AlphaServer is equipped with four 21164A processors operating at 600 MHz, 2 GB of shared memory, and a Memory Channel network interface. The Memory Channel [9] is a PCI-based crossbar network, with a peak point-to-point bandwidth of approximately 83 MBytes/sec. The network is capable of remotely writing to memory mapped areas, but does not have remote read capability. One-way latency for a 64-bit remote-write operation is 3.3 μsecs.

The 21164A has two levels of cache on chip. The first level consists of a split direct-mapped 8 KB instruction and data cache, with a line size of 32 bytes. The first-level data cache is write through. The second-level cache is a 96 KB 3-way set associative unified cache, with a line size of 64 bytes. Our platform has an 8 MB direct-mapped board level cache, with a line size of 64 bytes. Each AlphaServer runs Digital Unix 4.0F, with TruCluster v. 1.6 extensions. All the programs, the runtime library, and Cashmere were compiled with **gcc** version 2.8.1 using the **-O2** optimization flag.

[3]While some operating systems already provide this ability, we had to add this system call to Tru64.

On our platform, a scheduling quantum is approximately 10ms and a process runs until the quantum expires unless there is a higher priority process. A null system call takes approximately 0.5 μs and a context switch takes approximately 6 μs.

In Cashmere, a page fetch operation takes approximately 220 μs on an unloaded system, twin operations require 68 μs, and a diff operation ranges from 100–245 μs, depending on the size.

3.2 Experimental Results

In order to evaluate our system, we used a set of six kernels as our benchmarks. These benchmarks exhibit a range of sharing patterns and types of parallel regions. We briefly describe each application below.

- **Matrix Multiply:** A simple matrix multiplication algorithm parallelized by forming tasks with groups of rows and distributing these tasks among processes. The dataset consists of three 512x512 matrices of long integers (8 bytes each) — one each for the multiplier, multiplicand, and result.

- **Jacobi:** An iterative method for solving partial differential equations with nearest neighbor averaging as the main computation. We used a matrix of 2048x2048 single precision floating point numbers (4 bytes each).

- **SOR:** Successive-over-relaxation is a nearest neighbor algorithm from the TreadMarks [3] distribution, which is also used to solve partial differential equations. A matrix of 4096x1024 of double precision floating point numbers (8 bytes each) is used in our experiments.

- **Shallow:** The shallow water benchmark from the National Center for Atmospheric Research. This code is used in weather prediction and solves difference equations on a two dimensional grid. 13 Matrices of size 2048x2048 are needed where each element is a double precision floating point number, totaling 436MB.

- **Water:** A molecular dynamics simulation from the SPLASH-1 [24] benchmark suite. The bulk of the interprocessor communication occurs during a phase that updates intermolecular forces using locks, resulting in a migratory sharing pattern. Between each update phase, a barrier operation is performed. We use an input set of 9261 molecules, with the size of each molecule's data structure being 672 bytes.

- **Gauss:** A parallel gaussian elimination algorithm. The solution is computed by using partial pivoting and back substitution, and the row elimination is parallelized. The dataset size in our experiments is a matrix of 2048x2048 double precision floating point numbers (8 bytes each).

All our runs were conducted on 4 nodes with 4 processors on each node, for a total of 16 processors. For the purposes of loading a processor, we use a program that executes in a tight loop incrementing a variable (a pure computational load). At periodic increments, the elapsed time is recorded in order to provide a measure of the progress of the background load and to ensure scheduling fairness. With two processes running simultaneously, the elapsed time between

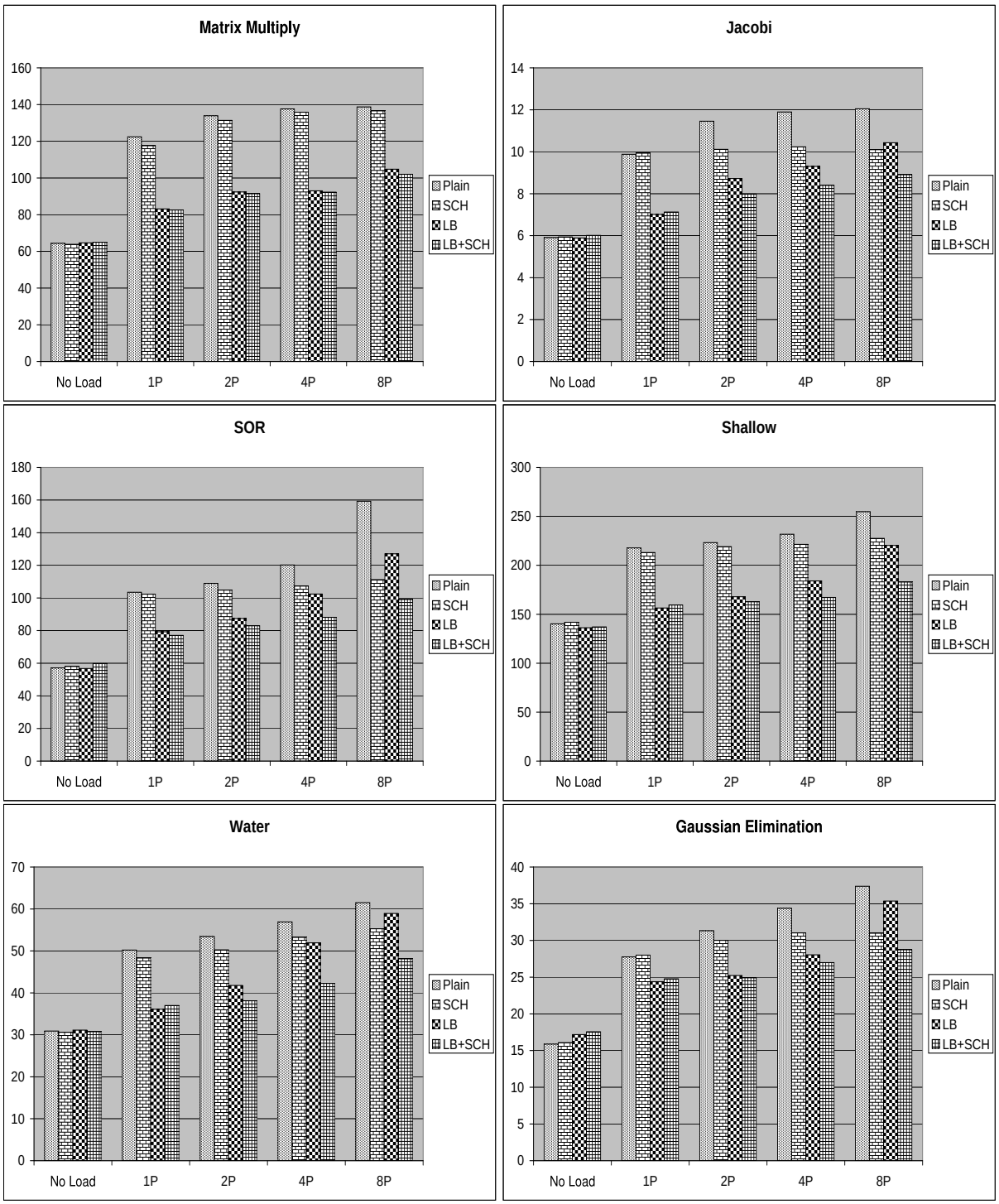

Figure 4: Execution times with and without load balancing and cooperative scheduling support as the number of loaded processors (each with 50% load, 1P = 1 loaded processor on 1 node, 2P = 2 loaded processors on 1 node, 4P = 2 loaded processors on 1 node and 1 loaded processor on each of 2 nodes, and 8P = 2 loaded processors on each node) is increased. The X-axis shows the number of multiprogrammed processors. "Plain" indicates no scheduling or load balancing support. "LB" indicates load balancing support, "SCH" indicates cooperative scheduling support, and "LB+SCH" indicates the use of both load balancing and cooperative scheduling supports.

Program	Seq. Time (secs)	Load Type	Access Type	Global Sync. Rate (barriers/sec at 16p.)
Matrix	578.76	FIXED	INDEPENDENT	3.17
Jacobi	151.5	FIXED	STENCIL	68.02
SOR	2389.2	FIXED	STENCIL	70.01
Shallow	1510.8	FIXED	STENCIL	10.03
Water	795	FIXED	INDEPENDENT	2.46
Gauss	366.3	VARIABLE	INDEPENDENT	258.19

Table 1: Relevant Program Characteristics

two consecutive intervals should be approximately twice that when running on a dedicated processor.

We conducted 5 sets of experiments for each of the applications with varying numbers of multiprogrammed processors, where each multiprogrammed processor had a single additional computational load as described above. Figure 4 presents the results. Each set of experiments includes four bars representing the execution time of the program with various features turned on or off. "Plain" implies an execution with no scheduling or load balancing support. "LB" represents an execution with runtime load balancing support turned on. "SCH" represents an execution without load balancing but with the use of a cooperative scheduler in the kernel. Finally, "LB+SCH" represents an execution where both the load balancing support and the cooperative scheduler are turned on. The "No Load" case demonstrates the generally minimal overhead from adding support for dynamic load balancing and cooperative scheduling. In fact, as is demonstrated by the graph for Shallow (and explained below), execution time can actually be improved.

Table 1 presents the sequential execution times, global synchronization rates at 16 processors in the absence of load, and dominant load and access type for each of the parallel applications. We discuss each application below.

Matrix Multiply: As seen is Table 1, this application has one of the lowest rates of global synchronization relative to the applications in our benchmark suite. The total number of global synchronizations is 204. Hence, the time between two synchronization points (315 ms) is much longer than a single scheduling quantum (10 ms). Further, the amount of communication is very low. This makes the application indifferent to cooperative scheduling, and allows the load balancing to be effective even in the absence of cooperative scheduling. Overall reduction in execution time varies between 26 and 33%, in comparison to no support at all. Since the type of sharing in this application is labeled as *INDEPENDENT*, i.e., it is independent of the region being parallelized, the load balancer uses a blocked task distribution and is able to minimize the communication while redistributing tasks by moving tasks directly from the heavily loaded to the lightly loaded processors.

Jacobi and SOR: Both of these applications exhibit nearest neighbor sharing as indicated by the access type of "STENCIL". Hence, the load balancer uses a blocked task distribution, and redistribution is performed by moving tasks between neighboring processors in order to minimize sharing-based communication (If redistribution is performed assuming that the loops are independent, the resulting steady-state communication (page transfers) is doubled, and the execution time increased by a factor of 1.5). As demonstrated by the "no-load" case in Figure 4, the overhead of using our features even in the absence of load is negligible for both applications. Ideally, in the absence of load balancing, a 50% load on any process should result in a doubling of the execution time. As we increase the number of loaded processors, however, the execution time for the base case ("plain") continues to increase and is more than double the no-load case when 8 of the processors are loaded. This is due to the independent scheduling decisions made by the underlying operating system. Examining the "SCH" bars in the figures shows that cooperative scheduling is highly effective in eliminating waiting time due to descheduled processes — the execution time as the load is increased stays constant, as one would expect in the absence of load balancing. As the number of loaded processors increases, the effectiveness of load balancing alone ("LB") decreases substantially, because the computation between two rendezvous points in both applications is relatively low. Combining load balancing and scheduling support results in an improvement in the execution time of at least 24% for all cases.

Shallow: Shallow is an application that demonstrates the effectiveness of the locality optimizations that take coherence unit size into account. Shallow performs several boundary condition initializations on its matrices. Since this computation is performed on a single row of the matrix, and a single row fits in one coherence unit, parallelization across nodes would result in excessive communication due to false sharing. The runtime library is able to eliminate this communication by performing the computation on the node that caches the data. The benefits of this optimization are demonstrated by the reduction in execution time (3%) with the load balancer turned on even in the absence of load. Load balancing results in up to a 28% reduction in the execution time in the presence of load.

Water: A unique characteristic of this application is the fact that it acquires a per-molecule lock to update each of the molecules, since a molecule could be updated by each of the processes to show the interaction between all the molecules. While the number of global synchronizations is low (76 over the course of execution), a lock acquisition attempt could also incur delays when a process holding the lock is preempted. The likelihood of having a process descheduled while holding a lock increases with the number of multiprogrammed processors. As can be seen in Figure 4, pure load balancing reduces the execution time by only 4% when the number of loaded processors is 8. However, for all loaded cases the reduction in execution time is between 22% and 29% when both load balancing and cooperative scheduling are used.

Gauss: For Gauss, the amount of work in a parallel region decreases as the computation progresses. Hence, this is

an example where distributing iterations cyclically is necessary in order to preserve locality while balancing load (the compiler algorithm is able to detect this behavior using the loop indices). For this application, because of the potential variance in load among processors, accurate estimates of the relative power as well as appropriate yield times are difficult to predict. Hence, there is a slight performance degradation in the absence of load when using load balancing and/or the cooperative scheduler. As can be seen, load balancing support without a cooperative scheduler fails to improve performance significantly (improves performance by only 5%) when there are many loaded processors. The combination of the load balancer and the cooperative scheduler is able to reduce execution time by up to 23% in the presence of 8 loaded processors. For this program, there are 4101 global synchronization operations and 392K messages exchanged in roughly 16 seconds of execution time. In other words, a global synchronization is executed every 4 msecs, i.e., at a granularity much smaller than the scheduling quantum of 10 msecs. Even if a request for scheduling is sent to the remote node, it takes up to 833μs (one hardclock tick) for that request to be honored (the signal is delivered by Tru64 only at the end of the clock tick). Hence, it is likely that as the load is increased, the parallel application is forced to wait at the global synchronization for descheduled processes despite the cooperative scheduler.

Finally, Figure 5 compares the execution times of a 16 processor execution with 8 loaded processors (a load of 50%) with that of running the application on 8 and 12 processors without load. The 12 processor execution time in the absence of load represents the (in practice unattainable due to communication overheads) ideal execution time we would like to achieve with 16 processors, given the load we have imposed. The 8 processor execution time represents a space sharing approach, where the application only uses the unloaded processors. Our expectation is that the execution time on 16 processors with 8 loaded processors is close to the 12 processor execution time. As the figure shows, this expectation is realized for Matrix Multiply and Water. Both these applications have access patterns where the sharing is independent of the parallelization strategy used for the parallel region. Hence, the use of 16 processors (as opposed to 12 or 8) does not result in additional steady-state communication. However, for Shallow, SOR, and Jacobi, the three applications with a *STENCIL* access pattern, steady-state communication at 16 processors is higher than at 12 processors. Hence, while performance is improved with load balancing, it is not close to the 12 processor case. As the results in Figure 5 show, Gauss is one application for which although performance is improved relative to "plain" (Figure 4), once more than four processors are loaded, the application would actually be better served by abandoning the loaded processors and executing all of the computation on the unloaded processors (a space sharing approach). We are currently also exploring this avenue of adaptation.

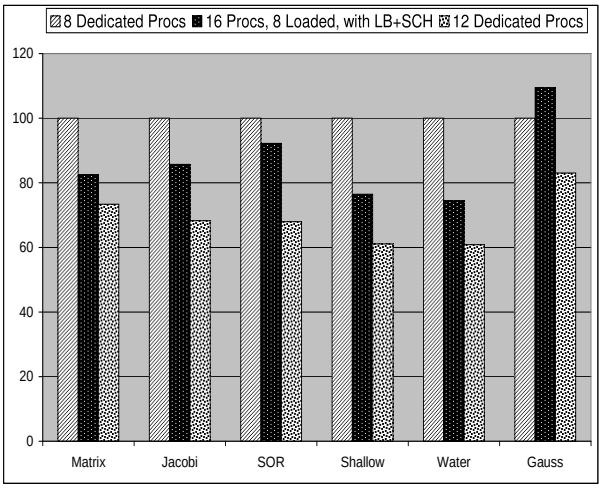

Figure 5: Comparison of execution times (normalized to the execution time on 8 dedicated processors) on 8 dedicated processors, 16 processors with 8 loaded processors, and 12 dedicated processors.

4. CONCLUSIONS

We have presented a system that combines compile-time analysis, runtime load balancing and locality considerations, and cooperative scheduling support for improved performance of parallel programs on autonomous distributed hardware. Our results show that while load balancing alone is effective in improving resource utilization when the number of loaded processors is low, combining load balancing with cooperative scheduling is essential to improving performance when the number of loaded processors is high. The cooperative interface to the kernel is minimally intrusive, and takes advantage of shared data between the kernel and cooperating processes. Our cooperative scheduling mechanism ensures fairness of resource allocation for all running processes. Our runtime load balancing techniques are able to take advantage of locality and communication cost information to minimize overall execution time by balancing locality and load considerations. Further, our task scheduling approach takes the variance in communication latency among processors into account. The result is a system that is able to adapt to dynamically changing resource availability without changes to the application.

Future work includes an evaluation of the scalability of the system, both in terms of being able to handle multiple simultaneously executing parallel applications and in terms of being able to scale to a larger number of processors. In addition, further experimentation on the frequency of allowed redistribution as well as the sensitivity to variance in load is needed.

5. REFERENCES

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, pages 66–76, December 1996.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.

[3] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel.

Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[5] A.L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 474–482, April 1997.

[6] A. D. Dusseau, R. H. Arpaci, and D. H. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of SIGMETRICS 1996*, pages 25–36, PA, USA, May 1996. ACM.

[7] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 186–197, October 1996.

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

[9] R. Gillett. Memory channel: An optimized cluster interconnect. *IEEE Micro*, 16(2):12–18, February 1996.

[10] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[11] D. G. Morris III and D. K. Lowenthal. Accurate data redistribution cost estimation in software distributed shared memory systems. In *Proceedings of the 8th Symposium on the Principles and Practice of Parallel Programming*, June 2001.

[12] S. Ioannidis and S. Dwarkadas. Compiler and run-time support for adaptive load balancing in software distributed shared memory systems. In *Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, pages 107–122, May 1998.

[13] S. Ioannidis, U. Rencuzogullari, R. Stets, and S. Dwarkadas. CRAUL: Compiler and run-time integration for adaptation under load. *Journal of Scientific Programming*, pages 261–273, August 1999.

[14] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[15] C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, October 1985.

[16] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *1993 International Conference on Parallel Processing*, pages 140–147, August 1993.

[17] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[18] D. K. Lowenthal and G. R. Andrews. An adaptive approach to data placement. In *10th International Parallel Processing Symposium*, April 1996.

[19] H. Lu, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, pages 48–56, June 1997.

[20] E. P. Markatos and T. J. LeBlanc. Load balancing versus locality management in shared-memory multiprocessors. *1992 International Conference on Parallel Processing*, pages 258–267, August 1992.

[21] S. Nagar, A. Banerjee, A. Sivasubramaniam, and C. R. Das. A closer look at coscheduling approaches for a network of workstations. In *Proceedings of 11th ACM Symposium on Parallel Algorithms and Architectures*, pages 96–105, June 1999.

[22] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30. IEEE, October 1982.

[23] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. In *IEEE Transactions on Computers*, December 1987.

[24] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. *Computer Architecture News*, 20(1):2–12, March 1992.

[25] P. G. Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Machines.* PhD thesis, M.I.T., January 1997.

[26] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M.L. Scott. Cashmere-2L: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170–183, October 1997.

[27] P. Tang and P. C. Yew. Processor self-scheduling for multiple nested parallel loops. In *1986 International Conference on Parallel Processing*, August 1986.

[28] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM SIGOPS Symposium on Operating Systems Principles*, pages 159–166. ACM, December 1989.