A Technique for Adaptation to Available Resources on Clusters Independent of Synchronization Methods Used *

Umit Rencuzogullari and Sandhya Dwarkadas

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

Abstract

Clusters of Workstations (COW) offer high performance relative to their cost. Generally these clusters operate as autonomous systems running independent copies of the operating system, where access to machines is not controlled and all users enjoy the same access privileges. While these features are desirable and reduce operating costs, they create adverse effects on parallel applications running on these clusters. Load imbalances are common for parallel applications on COWs due to: 1) variable amount of load on nodes caused by an inherent lack of parallelism, 2) variable resource availability on nodes, and 3) independent scheduling decisions made by the independent schedulers on each node. Our earlier study has shown that an approach combining static program analysis, dynamic load balancing, and scheduler cooperation is effective in countering the adverse effects mentioned above. In our current study, we investigate the scalability of our approach as the number of processors is increased. We further relax the requirement of global synchronization, avoiding the need to use barriers and allowing the use of any other synchronization primitives while still achieving dynamic load balancing. The use of alternative synchronization primitives avoids the inherent vulnerability of barriers to load imbalance. It also allows load balancing to take place at any point in the course of execution, rather than only at a synchronization point, potentially reducing the time the application runs imbalanced. Moreover, load readjustment decisions are made in a distributed fashion, thus preventing any need for processes to globally synchronize in order to redistribute load.

*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 the U.S. Department of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460.

1. Introduction

Clusters of Workstations (COWs) are attractive since they provide high compute power at low cost. COWs provide lower maintenance cost by allowing heterogeneous hardware to be put together, enabling partial upgrades and providing the opportunity to spread an upgrade over a period of time. COWs also have lower operating cost as they use readily available hardware and software. Each machine is generally autonomous and runs an independent copy of the operating system, allowing all users equal and uncontrolled access privileges.

Unfortunately, the advantages we enumerated above turn into disadvantages when COWs are used as a parallel computing platform. Since COWs are mostly operated as autonomous systems, scheduling decisions are made independently. Furthermore, since access to individual machines is not regulated, it is likely that some of the nodes running a parallel program will have other programs running on them. Heterogeneous clusters imply variance in resources among nodes. Each of these reasons contribute to the likelihood that some nodes will be overloaded while others are underutilized. Combining these factors with program characteristics, such as inherent lack of parallelism or unpredictable processing requirements, makes running parallel programs in such an environment much less efficient than expected.

In the past, each of these problems has been addressed in isolation. Load balancing has been studied for both COWs and parallel machines [8, 12, 22, 17, 10, 4]. Lowenthal et. al. [12] and Morris et. al. [7] 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.

Coscheduling has also been studied extensively, in the form of explicit coscheduling [16, 23] for parallel machines, and in the form of implicit coscheduling [20, 3, 15], for

COWs.

Coscheduling works best when the amount of work and the available resources are equal across all nodes. On an autonomous COW it is highly likely that nodes have different numbers of processes running, which reduces the effectiveness of the coscheduling approach. If implicit coscheduling is used, the resulting schedule might also no longer be fair. On the other hand, load balancing takes its motivation from the assumption that each of the nodes has a different load/resource ratio, and tries to make that ratio equal across nodes. Even when load balancing is perfect across nodes, delays caused by multiprogramming will not be addressed, since communicating or synchronizing processes would have to wait for their communication/synchronization partner to be scheduled before they are able to finish their operation.

An obvious solution would seem to be to combine load balancing with a coscheduling scheme. Unfortunately, a trivial combination does not work since it still suffers from message delays, or possible unfairness. For example, consider a parallel application running on two nodes, where one of the nodes, say A, is dedicated, and the other, B, is running another application along with the parallel application. Assume the load is balanced, i.e. the parallel process on node A is getting twice as much work to do as its peer on node B. A coscheduler cannot do anything, since the parallel application is naturally coscheduled. However, in this case it is possible for the process running on A to send a message and wait for its partner to be rescheduled, should the partner process on B be de-scheduled. The wait time could be as high as one quantum.

Our solution to the above problems has been developed in the context of software distributed shared memory (SDSM). An SDSM protocol provides the illusion of shared memory to the programmer. The benefit in our case is that the complexity of the load balancing middleware is reduced since data is automatically moved to where it is accessed rather than having to be explicitly coded. Our earlier study [18] has shown an effective way of coping with load imbalances and scheduling discrepancies for applications using barriers to synchronize. A "barrier" requires all involved processes to arrive at the barrier before anyone may depart from it.

Our new system extends our previous work by being oblivious to the synchronization method used. This permits the exclusive use of locks and flags, which are much more relaxed methods of synchronization. A lock requires an acquirer to wait only if the lock to be acquired is being held by another party. Flags, on the other hand, are an event-based synchronization mechanism that can be used to signal the completion of an event. Further, we explore the scalability of our system and show that our combination of cooperative scheduling and load balancing scales up to 32 processors and provides excellent performance benefits. Our new extensions benefit even barrier-based applications by providing them with ways of balancing load without waiting for a barrier, in cases where the time between barriers is large.

```
int     sh_dat1[N], sh_dat2[N],a,b,c,d;

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 the prefix `sh_`.

The rest of the paper is organized as follows: Section 2 provides information about our former system and how our new system differs and what capabilities are added. Section 3 evaluates the system separating the effects of each component. Finally, Section 4 concludes and outlines our plans for future work.

2. Design And Implementation

Our programming environment is software distributed shared memory (DSM). DSM hides the details of communication from the user by providing a layer below the application that implicitly manages data movement. Data movement due to load reassignment, which would otherwise be needed to be somehow explicitly added to the program, is handled implicitly. Even though it is relatively easy to explicitly code data movement when the data to be moved is known statically, it is much harder when the decision is made dynamically, since the source and destination are hard to determine. We use Cashmere-2L (CSM) [21], as our base DSM system.

Due to relaxing the requirements of the use of barriers, the monitoring and redistribution methods we used in our earlier work must be modified. We elaborate on each of the components and their modifications in the following sections.

2.1. Baseline Load Balancing System

Our baseline system consists of three separate subsystems: static program analysis, runtime system, and operating system scheduling support. Our earlier work [18] presented a scheme to balance load and increase the throughput of COWs for "barrier-based" parallel applications. In that implementation, all synchronizations were pushed into the runtime system, and load was balanced only at synchronization points, which were barriers. Our current extensions, in addition to supporting other classes of applications, also allow barrier-based applications to balance at arbitrary points,

```
Initialize
  sharing types /*STENCIL/INDEPENDENT*/
  load types    /*FIXED/VARIABLE*/
  list of arrays,      /*sh_dat1,sh_dat2*/
  list of access types,  /*read/write*/
  list of upper/lower bounds and strides
  coefficients and constants /*a,b,c,d*/

taskSet = partition_tasks();

get a task while there are Tasks in the taskSet
  set lowbound, highbound, stride for that Task

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

Figure 2. Parallel loop with pseudocode that serves as an interface to the runtime system. The runtime system can then change the amount of work assigned to a process.

hence making it possible to balance load between barriers when the time between successive barriers is large. This in turn reduces the amount of time an application runs unbalanced.

The following sections describe each of the three components separately and explains what had to be changed in order to extend our system to work with a larger class of applications.

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 runtime-system hooks that monitor the process activity. 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 [9] 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 a sample parallel loop. Figure 1 shows pseudo-code for the original loop. There are several pieces of information that the compiler supplies to the runtime system. 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 [6]) 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 only once, with only those variables that change are updated on each invocation.

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 the parallel loop index implies that in order to achieve a balanced distribution of load while preserving access locality, 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 intended to influence the type of load distribution used, and to determine the type of redistribution used. *Access* can potentially be updated by the runtime system 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 heavily loaded processor's tasks directly 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 system, we use the Stanford University Intermediate Format (SUIF) [1] compiler. The SUIF system is organized as a set of compiler passes built on top of a kernel that de-

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

fines 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 system with information about each parallel region's access patterns. The second pass works on parallelized programs and modifies the loop structure by inserting the required runtime system hooks and modifying the loop structure to use runtime-system provided execution parameters.

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. Runtime System

The runtime system is the main component that uses the information provided by the static analysis to partition, distribute and redistribute the work among all cooperating processes. Also, it is the component that using the provided interface, cooperates with the scheduler and other processes. Load distribution is based on the concept of `RelativePower`, and guided by the statically provided information and runtime feedback.

2.3.1. Relative Processing Power

As described in [9, 18], in order to partition the load according to available resources we try to estimate available computational resources and communication overhead. In general it is reasonable to assume a node with more resources is capable of doing more work in a given amount of time. We base our load distribution decisions on our estimation of the computing capability of a node, which we call `RelativePower`. Intuitively, `RelativePower` is proportional to w/t, where w is the `amount of work done` and t is `the elapsed time`. For applications that employ barriers, all processes work in the same parallel region, on their appointed work, until they are done and reach the next barrier. In this scheme, it is trivial to know the amount of work done, since all assigned iterations of all parallel regions have to be executed before a barrier is reached and `RelativePower` is recomputed. The other component, time, can also be measured easily as well, hence, making the estimation of `RelativePower` fairly straightforward.

However, we do not rely on existence of barriers in applications in our new runtime system. When no barriers are employed, it is likely that multiple parallel regions are spanned and different number of iterations are executed in each of these parallel regions by each of the processes. In most cases, not all processes have finished their assigned work. This makes estimation of the amount of completed work, harder to obtain. For the purposes of computing the `RelativePower`, we use the minimum execution time for an iteration as the basis for determining the amount of work for that iteration. Prior to computation of the `RelativePower`, all processes exchange their number of executed iterations, and the time it took to execute these iterations, for each of the parallel regions. Figure 3 shows the pseudocode of how to compute `RelativePower`.

```
static float RelativePower[NumProcs]
  //Initialized to 1/NumProcs
float IterTime[NumParRegs][NumProcs]
  //Execution time of parallel region
float NumIters[NumParRegs][NumProcs]
  //Executed Tasks in Each Region
float WorkPerProc[NumProcs]
float PerProcExecTime[NumProcs]
float Power,SumOfPowers=0

//Calculate the amount of work done by each
//process and the time it took to do so.
for all Parallel Regions i
  float AvgTime,WorkDone
  float MinIterTime = LARGENUMBER

  for all Processes j
    PerProcExecTime[j] += IterTime[i][j]
    AvgTime = IterTime[i][j]/NumIters[i][j]
    if (AvgTime < MinIterTime)
       MinIterTime = AvgTime
  for all Processes j
    WorkDone = NumIters[i][j] * MinIterTime
    WorkPerProc[j] += WorkDone

//Calculate RelativePower
for all Processes i
  Power = WorkPerProc[i]/PerProcExecTime[i]
  RelativePower[i] =  Power
  SumOfPowers += Power

//Normalize The RelativePower
for all Processes i
  RelativePowers[i] /= SumOfPowers
```

Figure 3. Computing "RelativePower".

2.3.2. Task Distribution Strategy

Upon entrance for the first time to a parallel region, the runtime system 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 their size is a multiple of the coherence unit. 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.

At any time the runtime system has a notion of perceived `RelativePower`. Program execution starts by assuming all processors have equal amount of work to do and they are all equally powerful, i.e. their `RelativePower` is equal. The `RelativePower` is updated regularly as explained in Section 2.3.1. Load reassignment occurs when a significant change in the `RelativePower` is detected[2].

While tasks are being created, if the access pattern is *STENCIL*, then a single task per process, sized proportional to the perceived `RelativePower` of that process is created. If the access pattern is not *STENCIL*, however, many equal-sized fixed tasks are created and they are distributed among processes with numbers proportional to their perceived `RelativePower`. For *STENCIL* regions, load is balanced by changing the size of the task via shuffling its boundaries. Otherwise, load is balanced by moving tasks from processors with decreased `RelativePower` to those with increased `RelativePower`. Even though balancing itself might involve more communication when the access pattern is *STENCIL*, the steady-state communication is reduced by making sure all assignments have the least number of boundaries possible.

Task assignment and execution take the topology of the processors into account. For a cluster of SMPs, work is 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 [14]. 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 [11]). 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.

When any of the processes is not able to proceed due to not finding anything in its task queue, the first attempt is to steal tasks from processes running on the same SMP node. If a *suitable* task is not found there either, a balancing request message is issued. Upon this request, all processes exchange their execution statistics and compute the new `RelativePower`. Exchanged statistics also include the number of times a process iterated over a parallel region. For the purposes of reducing the number of messages and message assembly time, statistics are written directly into network mapped memory.

To accommodate applications without barriers, it is required to decide when a new load reassignment should take effect. Since some processes might be lagging behind, every reassignment of tasks does not take effect immediately for all processes. A process may move a task from another process only when the source and destination processes are at the same phase with respect to the region being processed. If the old owner is lagging behind (the most likely scenario) and a task is moved, a naive approach might skip some computation stages (parallel regions or iterations), resulting in incorrect computation. It is also possible for the old owner to be ahead, for example, when the parallel region's access type is *STENCIL*, in which case work might erroneously be replicated. In order to simplify the implementation and reduce the need for synchronization, for each of the parallel regions we determine the current computation state of the fastest process in terms of the current parallel region and the number of times each parallel region has been executed. Old owners of tasks that are moved use this information to stop processing them when this computation state is reached. The new owners take over at that time.

Guaranteeing the coherence of a task to be moved requires the old and new owner of a task to synchronize after the old owner operates on the data for the last time, if they have not done so already. To guarantee coherence, consistency operations and task modifications are timestamped. The movement of a task is legal if the timestamp of the consistency operation is larger than the modification timestamp of the task being moved. Otherwise, the old owner and the new owner of the task need to exchange updates to bring the copy of the new owner up-to-date.

2.4. Cooperative Scheduling Support

Multiprogramming adds an additional dimension to the problem of imbalanced load. Communication among cooperating processes can result in significant delay if one of the cooperating processes is de-scheduled and unable to respond. Coscheduling [16, 20, 3, 23, 15] approaches have been used in the past, where cooperating processes are scheduled to execute simultaneously on all processors. This

[2] At least one of the `RelativePowers` must change enough to make sure some load movement would actually happen

approach is good 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 reduce the wait time experienced 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. Our implementation is on Compaq's Tru64 (formerly known as DEC Unix) version 4.0F.

2.4.1. 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, and charging that amount to the process), but is guaranteed not to grow larger than one quantum. This prevents a process from taking over the processor for long periods of time by yielding often. When a scheduling request is received, the scheduler uses the time in the piggy-bank, if any, to schedule the intended process.

2.4.2. OS-Runtime Interface

For a process of a parallel application to be scheduled on demand, the desire to schedule it needs to be communicated to its scheduler. For many networks, receiving a message involves executing some code on the recipient end, in the driver. This code could be modified to implicitly communicate with the scheduler, to express desire for immediate scheduling. Our network, Compaq's Memory Channel, however, is a low latency remote write network, and it does not execute any code on the recipient host CPU, upon receipt of a message. Hence the desire for scheduling a peer need to be communicated explicitly. For that purpose we send a signal. However, sending a signal after each message or at every synchronization point is expensive, and in some cases, where the peer is already scheduled or the message is asynchronous, it is not needed. For the purposes of eliminating excessive signals, we employ other features provided by our network to communicate among cooperating processes.

We provide a system call that allows each process to register a signal and a memory location. The registered signal is used by a cooperating process as a wakeup signal. Upon receiving that signal, rather than delivering it to the application, if the scheduler can schedule the process using the equity in the piggy-bank, while continuing to guarantee fairness, it does so.

The registered memory location has two boolean words. The first ("scheduling status") is written by the scheduler and gives hint to other processes regarding the scheduling status of the registering process. It is set by the scheduler when the process is de-scheduled. The second word ("signaled") is set by signal-sending processes to inform others that a scheduling request has been sent to a particular process, preventing them from sending yet another signal. When the process is scheduled, both words of the registered memory are reset, indicating that the process is scheduled and no scheduling request is pending. These memory locations are placed in network mapped memory, and modifications to these locations are broadcast to all other processes. 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.

2.4.3. Runtime 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 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. The yield 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 spin-block strategy [15] is used 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.

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 our platform. If a process yields frequently enough, it might either not be charged at all for its use, or it could be over-charged depending on the rela-

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

tive timing of a clock-tick and the yield call. In order to fix this, we used hardware cycle counters as the basis for accounting.

3. Evaluation

3.1. Experimental Platform

Our experimental environment is a cluster of Compaq AlphaServer 4100 workstations. Each workstation is equipped with four 21164A processors operating at 600 MHz, 2 GB of shared memory, and a Memory Channel network interface. The Memory Channel [5] 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. The one-way latency for a 64-bit remote-write operation is 3.3 μsecs.

All the programs, the runtime library, and Cashmere were compiled with **gcc** version 2.8.1 using the **-O2** optimization flag. 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.

3.2. Experimental Results

In order to evaluate our system, we used a set of eight applications as our benchmarks. These benchmarks exhibit a range of sharing patterns and types of parallel regions. Figure 4 shows the execution results of our applications. We ran our applications using 32 processors, under 4 different load schemes as shown in Table 1. For the purposes of loading a processor, we use a program that executes in a tight loop incrementing a variable (a pure computational load). For each load scheme, we present execution times with no support (*None*), with support for task stealing within a node enabled (*Steal*), and with task stealing within a node as well as load balancing across nodes enabled (*Balance*). The case labeled “No Load” is the base case. It is intended to show baseline execution time and the runtime system overhead, and in some cases how the runtime system might help offsetting the anomalies due to the application algorithm, even in the absence of any other load. The “Load-4” case is intended to show the adaptation of the system by shifting the parallel program tasks from an overloaded node and distributing these tasks among other nodes. The “Load-8” case is intended to show the effectiveness of intra node load balancing, by task stealing. Finally, the “Load-16” case shows the effectiveness and scalability of load balancing schemes as well as how scalable the cooperative scheduling is. Note that cooperative scheduling is enabled in all the experiments. Hence these performance improvements are on top of what cooperative scheduling could achieve. Section 3.2.1 evaluates the effect of cooperative scheduling on performance. Following is a description of our benchmarks and discussion of our results.

Label\Nodes	0	1	2	3	4	5	6	7
No-Load	0	0	0	0	0	0	0	0
Load-4	0	4	0	0	0	0	0	0
Load-8	1	1	1	1	1	1	1	1
Load-16	0	0	0	0	4	4	4	4

Table 1. Number of processors running a sequential program along the parallel program for each of the configurations.

Transitive Closure: A graph algorithm that checks reachability from one vertex to others. The main intuition in the implementation is that if vertex A is reachable by vertex B, then every vertex reachable by vertex A is also reachable by vertex B. We used a random input with any pair of vertices having 60% likelihood of having an edge. The amount of computation depends on whether the pair of vertices picked up could reach each other. Hence, even though the loop structure is regular, the computation within the loop is conditional, creating an application induced (short-term) load imbalance.

Gaussian Elimination: 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 8Kx8K floating point numbers. Flags are used for synchronization purposes. A processor sets a flag upon computing the pivot, which in turn signals availability of the pivot to other processors. This implementation has more relaxed synchronization than a barrier implementation, because it allows two processes to be working with different pivots at any time of the computation. Furthermore, flags are known to be less affected by multiprogramming than barriers [13].

Jacobi: An iterative method for solving partial differential equations with nearest neighbor averaging as the main computation. We used a matrix of 8Kx8K floating point numbers. At each iteration, neighboring processors need to exchange their boundary rows of data. Two arrays are employed, where one is used as a scratch pad. Two barriers are required at each iteration, one after doing the averaging and the other after copying the data from the scratch pad to the main array. Since it exhibits nearest neighbor sharing, a single task is created per process to reduce steady state communication. Load balancing is achieved by resizing assigned task rather than changing the number of fixed tasks assigned to each processor.

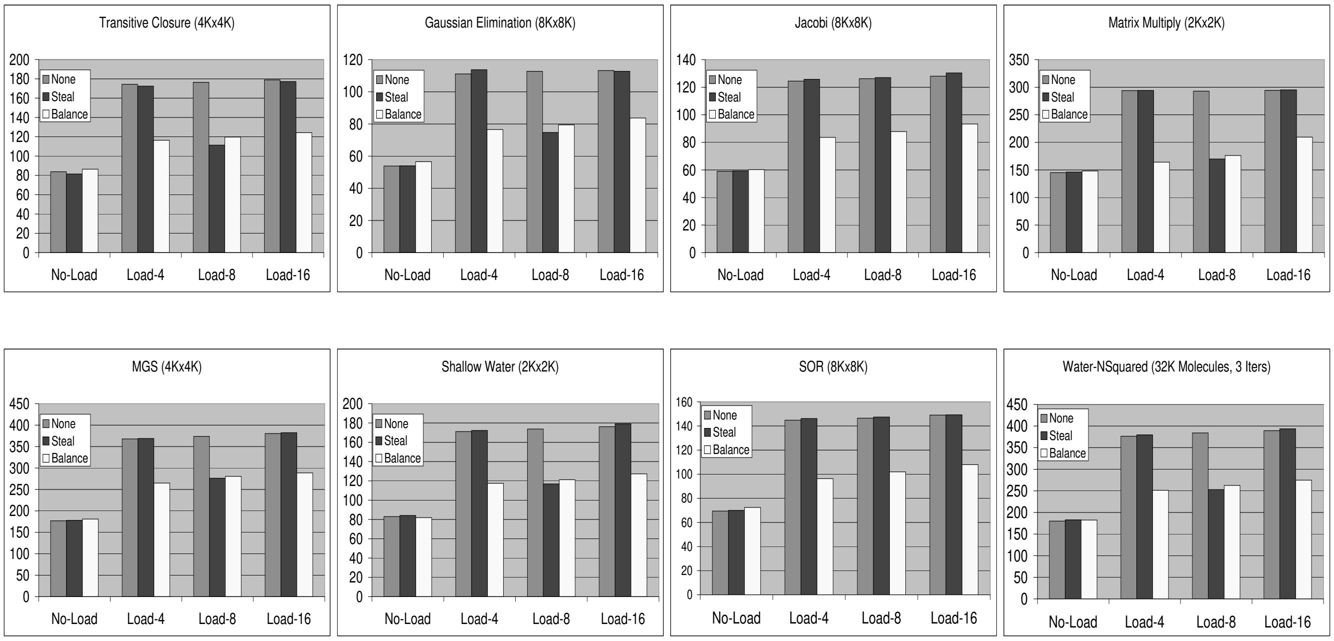

Figure 4. Effectiveness of load balancing. The Y-axis is the execution times in seconds. X-axis labels indicate the load as explained in Table 1. "None" indicates no balancing was done, "Steal" indicates only intra-node task stealing was allowed, and finally "Balance" indicates both intra-node stealing and inter-node balancing were allowed. In all cases scheduling support was on.

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 2048x2048 matrices of integers — one each for the multiplier, multiplicand, and result. This application has very long periods of computation and very little communication or synchronization. As a result, it is oblivious to lack of coordination.

Modified Gramm Schmidt (MGS): This application computes an orthonormal basis for a set of N-dimensional vectors. At each iteration i, the algorithm first sequentially normalizes the i^{th} vector, then makes all vectors $j > i$ orthogonal to vector i, in parallel. Since the application leaves out a row at each iteration, the created distribution is cyclic.

Shallow: The shallow water benchmark from the National Center for Atmospheric Research. This code is used in weather prediction and solves differential equations on a two dimensional grid. During the execution, 11 parallel regions are spanned, with some of the regions taking only a few milliseconds, involving a single row of the matrix.

SOR: Successive-over-relaxation is a nearest neighbor averaging algorithm from the TreadMarks [2] distribution, that is also used to solve partial differential equations. A matrix of 8Kx8K floating point numbers is used in our experiments.

Water-NSquared: A molecular dynamics simulation from the SPLASH-1 [19] benchmark suite. It is run for 3 steps. 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 32K molecules. The application acquires a lock to update each of the molecules. Our SDSM system has a maximum number of 4K locks, causing 8 molecules share one lock. Even though contention for a lock is likely, the critical region is very short. Furthermore, if any process holding a lock happened to be de-scheduled, the cooperative scheduling mechanism would reschedule it, reducing the wait time. Despite the high number of locks, the number of barriers executed in the course of the run is small.

As Figure 4 demonstrates, having both intra-node task stealing and inter-node load balancing enabled (i.e. "Balance") reduces the execution time across all application when there is any load, and the overhead in the absence of load is minimal. The reduction in execution time is no less than 26% and it is as high as 44%. Several results are worth noting. First of all, for all the applications with the exception of jacobi and SOR, intra-node task stealing alone achieves slightly better runtime reduction for the "Load-8" (no inter-node imbalance) case. This is mainly due to the fact that task stealing alone has less overhead than inter-node load balancing, and in some cases, load balancing might cause one piece of work to move from one node to another causing slightly more communication in the system. For **transitive closure**, even in the absence of load the execution time is reduced 3% when intra-node task stealing is used. This is due to smoothing the short-term load imbalance induced by application characteristics, by stealing tasks from processes lagging behind. **Gaussian elimination** uses the most relaxed form of synchronization, and the results show that our system handles this type of synchronization correctly and effectively. **Jacobi and SOR** are applications with loops marked as being *STENCIL*. This causes creation of a single task per process, and makes task-stealing impossible. However, load balancing is very effective for both applications. **Matrix multiply** has been shown, in our earlier study [18], to exhibit a reduction of 33% at most when all features were turned on. In this set of experiments the reduction is as high as 44%. This improvement is mainly the result of the ability to balance load before reaching a barrier (there are roughly 20 secs between barriers), and hence reducing the time the application runs imbalanced. **Shallow** benefits from the runtime system even in the absence of any other load, by cutting the execution time by about 1%, despite the overhead of the runtime system. This improvement is the result of localizing computation for loops which involve very little work over a small amount of data. This cuts the cost of communication by having the nodes that currently cache the data perform all the computation.

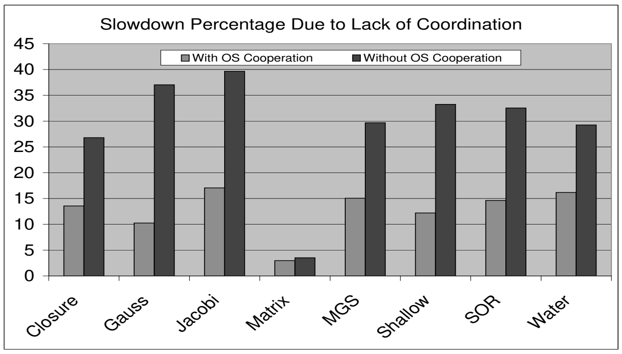

Figure 5. Effectiveness of cooperation. The left bar shows the percentage of overhead due to multiprogramming when the runtime system cooperates with the OS as described in Section 2.4. The right bar is when there is no coordination among parallel processes.

3.2.1. Effectiveness of Cooperative Scheduling

In order to determine how effective our cooperative scheduling is, we conducted experiments with and without cooperation. Results are shown in Figure 5. In all cases, applications were run with "Load-16" configuration as shown in Table 1. None of the load balancing or locality management features were used. Ideally the execution time of each application should be exactly twice as much as the case when all the nodes are dedicated. Any amount above this is a slowdown due to multiprogramming. Not cooperating with the OS leaves the scheduling uncoordinated and hence adds a slowdown of up to 39%. Just turning the cooperation on, which in effect causes interacting processes to be scheduled on demand, reduces this overhead by 50% on average, and in some cases as much as 72%. 72% reduction is achieved in Gaussian Elimination, where it is important to have the process producing the pivot be scheduled while others are fetching the pivot from it. Even though we reduce the effects of lack of coordination in all cases, there are several reasons for not achieving the ideal: 1) There is a delay of several microseconds between sending a signal and actually the signal reaching its destination. 2) Since events are polled by the OS at each hardware clock tick, which is about 833μs, a signal waits on average half of this time to be processed, after being received by the target node. 3) The context switch adds cost. 4) In some cases, it is possible for a scheduling request to be ignored if the recipient process has already used its fair share, and granting that request would be unfair to other processes on the same node. This is an indication that load balancing is needed, and in most cases the runtime system, when enabled, would redistribute the load moving some of the work to other processes.

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 applications running on an autonomous COW. The system works regardless of the synchronization method used by the parallel application and at the same time is fair to all processes. Reducing the execution time of the parallel program while being fair to other applications improves the throughput of the COW. Our previous work addressed similar issues for applications that used barrier synchronization. The extensions described in this paper are important not only because of extending the application base we address, but also due to providing opportunities for applications using barriers to be able to adapt to available resources before a barrier is reached. Furthermore, since barriers are known to be more vulnerable to resource mismatches, it provides applications with the opportunity to be implemented using synchronization primitives other than barriers.

For parallel applications that use our system, we have shown that using all the features helps to reduce the execution time by as much as 44% on top of the reduction achieved by cooperative scheduling and on up to 32 processors. Features such as the ability to reshuffle work on the fly without the need to be at a synchronization point benefit applications with long periods of computation between two synchronization points (e.g. matrix multiply). Our runtime system's awareness of the underlying topology of the cluster reduces the amount of communicated data considerably by moving the data within the SMP node first. Data is moved across nodes only when the aggregate load of the collection of processors within the node exceeds the pool of resources within that node.

References

[1] 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.

[2] 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.

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

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

[5] Richard B. Gillett. Memory channel network for PCI. *IEEE Micro*, 16(1):12–18, Feb 1996.

[6] 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.

[7] 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.

[8] Sotiris Ioannidis and Sandhya Dwarkadas. Compiler and run-time support for adaptive load balancing in software distributed shared memory system. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 107–122, May 1998.

[9] Sotiris Ioannidis, Umit Rencuzogullari, Robert Stets, and Sandhya Dwarkadas. Craul: Compiler and run-time integration for adaptation under load. *Journal of Scientific Programming*, August 1999.

[10] C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, Oct 1985.

[11] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth C. Sevcik. Locality and loop scheduling on NUMA multiprocessors. In *1993 International Conference on Parallel Processing*, pages 140–147, August 1993.

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

[13] E. Markatos, M. Crovella, P. Das, C. Dubnicki, and T. LeBlanc. The effects of multiprogramming on barrier synchronization. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, pages 662–669, December 1991.

[14] 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.

[15] Shailabh Nagar, Ajit Banerjee, Anand Sivasubramaniam, and Chita 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.

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

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

[18] U. Rencuzogullari and S. Dwarkadas. Dynamic adaptation to available resources for parallel computing in an autonomous network of workstations. In *Proceedings of the 8th Symposium on the Principles and Practice of Parallel Programming*. ACM, June 2001.

[19] 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.

[20] Patrick Gregory Sobalvarro. *Demand-based Coscheduling of Parallel Jobs on Multiprogrammed Machines*. PhD thesis, M.I.T., January 1997.

[21] 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.

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

[23] Andrew Tucker and Anoop 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.