Dynamic Resource Management for Parallel Applications in an Autonomous Cluster of Workstations

by

Umit Rencuzogullari

Submitted in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy

Supervised by

Professor Sandhya Dwarkadas

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

2004
Curriculum Vitae

Umit Rencuzogullari was born in Samandag, Hatay, Turkey, on June 15, 1972. He attended the Middle East Technical University in Ankara, Turkey, where he received the Bachelor of Science degree in Computer Engineering in 1993. He joined the Computer Science Department of the University of Rochester in the Fall of 1997, where he has been conducting parallel computing research under the guidance of Professor Sandhya Dwarkadas. He received a Master of Science degree in Computer Science in 1999. His research concentrates on load balancing and locality management for parallel applications, exploring the integration of compile-time information with the run-time system, along with safe operating system interfaces for better interaction with applications.
Acknowledgments

It is impossible to express my convoluted feelings in a single sentence. May be it is a closure with a deep sense of loss of a very important part of my life. First and foremost, I am greatly thankful to my advisor, mentor and teacher Sandhya Dwarkadas for her very helpful and motivating guidance, and her positivity through the toughest times. I am very appreciative to my committee members David Albonesi, Tom LeBlanc and Michael Scott for their invigorating discussions, questions, and guidance.

I am also grateful to all my classmates for making these years more enjoyable and very memorable. I would like to thank my friend Cem for his continuous support during these years, Rob Stets for giving me first-hand experience in what later evolved in to a big tool of my dissertation, Grigorios for solving every single script problem I had throughout the years, Rajeev for always running simulations and giving me the best guilt-free excuse not to work :-), DeQing for numerous helpful discussions, Bijun and Yutao for doing all the late stage work, from thesis registration to collecting signatures to final submission.¹

¹This material is based upon work supported by a grant from The National Science Foundation under grant number CDA-0401142, The National Science Foundation under grant number R9702466, The National Science Foundation under grant 9705594, The U.S. Department of Energy Office of Inertial Confinement Fusion under Cooperative Agreement No. DE-FC03-92SF19460, and an external grant from Compaq. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views or opinions of the funders.
Abstract

Clusters of Workstations offer high performance compared to their cost. Unless specifically set up with job control, these clusters operate as autonomous systems, where each of the nodes runs its own copy of the operating system and makes its own scheduling decisions, access to machines is not controlled, and all users enjoy the same access privileges. Even though these features are desirable and reduce operating costs, they create adverse effects for parallel applications running on these clusters. Adverse effects are in the form of delays at communication or synchronization points (rendezvous points), which are caused by either load imbalance, i.e., disproportional load to resource assignment, or by scheduling skew, i.e., timing of allocated resources. While these two issues have been addressed in the past independently and in isolation, no prior study addresses these two issues simultaneously despite the fact that they are likely to occur together, particularly in multiprogrammed environments.

We present an automatic resource allocation and scheduling technique aimed at reducing wait time for parallel applications while maintaining the fairness of scheduling decisions. We use access information from the compiler along with fair and coordinated scheduling to aid the runtime system in dynamically balancing load (while maintaining locality) based on resource availability. The novelty in our approach, besides being the first to address both causes, is to introduce fairness in resource allocation. The system requires little or no programmer interaction for many classes of applications. The outcome of our techniques is not only to reduce the execution time of the parallel application, but also to increase the throughput of the cluster overall.
Table of Contents

Curriculum Vitae ii

Acknowledgments iii

Abstract iv

List of Tables viii

List of Figures x

1 Introduction 1

1.1 Challenges of Using Clusters of Workstations 2

1.2 Summary of Prior Work 3

1.3 Proposed Solution 5

1.4 Dissertation Organization 7

2 Background and Related Work 8

2.1 Load Imbalance 9

2.2 Coordinated Scheduling of Parallel Applications in the Presence of Multiprogramming 20

2.3 Summary 24
3 Design Overview

4 FCS: Fair Cooperative Scheduling
   4.1 Overview of the Unmodified Scheduler ........................................ 32
   4.2 FCS Design and Policy ............................................................. 33
   4.3 Communicating Scheduling Demands ............................................. 35
   4.4 FCS Interface And Initialization ................................................ 36
   4.5 Emphasis on Fairness in FCS ..................................................... 39
   4.6 Safety of FCS ........................................................................... 40
   4.7 Summary and Discussion ............................................................ 42

5 Static Program Analysis ..................................................................... 44
   5.1 Identifying Data Structures of a Program ...................................... 45
   5.2 Static Hint Generation About Loop Structures and Data Accesses ...... 48
   5.3 Parallel Code Generation ............................................................. 52
   5.4 Summary .................................................................................... 55

6 LADLE: Locality Aware Dynamic Load Balancing Environment .......... 57
   6.1 Initial Task Creation and Assignment .......................................... 57
   6.2 Task-Queues and Task Management in LADLE ............................... 60
   6.3 Load Re-assignment Strategy ...................................................... 61
   6.4 Relative Processing Power ........................................................... 63
   6.5 Execution of Newly Moved Tasks ................................................ 67
   6.6 Task Stealing for Intra-Node Load Balancing ................................. 71
   6.7 Deciding to Balance Load ......................................................... 72
   6.8 Locality Management in LADLE .................................................... 73
6.9 Interacting with FCS ................................. 74
6.10 Adjusting Spin Time .............................. 75
6.11 Handling Irregular Applications ............... 77
6.12 Summary and Discussion ....................... 80

7 Evaluation .............................................. 83
  7.1 Evaluation Platform .............................. 83
  7.2 Experimental Results ......................... 85
  7.3 Running Multiple LADLE/FCS Applications Simultaneously ........... 104
  7.4 Evaluating LADLE-IRR on Irregular Applications .................... 106
  7.5 Summary and Discussion ....................... 108

8 Conclusions ............................................ 110
  8.1 Contributions ................................... 111
  8.2 Future Work .................................... 114
List of Tables

4.1 The exported FCS interface. FCS calls are safe since they are untrusted by default, and they only provide hints about an application’s expectations rather than dictating policies. .......................... 37

6.1 An illustration of calculating $\text{RelativePower}$ in LADLE-IND. The parallel application has two processes and each of the processes span two parallel regions — Region 1 and Region 2. “Min Avg Per Task” chooses the minimum average time spent by each of the processes working on a task. This is used as the actual work content of a task, and the number of completed tasks by each process is multiplied by this value to compute the amount of completed work in each parallel region. The work of all parallel regions is summed to find the “Total Work”, which is then divided by the total “Exec Time” to find raw $\text{RelativePowers}$. The raw $\text{RelativePowers}$ are normalized to obtain $\text{RelativePowers}$. ................................. 67

7.1 Applications used for evaluation of our system. Their input size, the time it takes for each of the applications to execute on a uniprocessor (in seconds), and the number of different synchronization operations performed by each of them when they are run in parallel on 32 processors, are shown. ................................. 94
7.2 Names and meanings of different load scenarios used for experiments. "Node" indicates a 4-way SMP node of our experimental platform. The number underneath each node show how many processes, other than those of the parallel application, run on that node. The last column indicates the total number of these processes, on the whole cluster. In all cases, the parallel application is running on 32 processors.
List of Figures

5.1 Inserted code for identifying the shared data structure sh_arr to the runtime system LADLE. Initially, we get a unique id for the array, and register its starting address and sizes in each dimension. As the type of the array, we are only interested in the size of each element. . . . . . . 45

5.2 A simple sequential loop, operating on array “sh_arr”. The loop assigns some values to every element of each row between column “start” and column “end” of the array. We assume this loop is parallelizable. Even though the outer-loop boundary always goes from 0 to M, and is known statically, the inner-loop boundaries are received as arguments to the function, and are known only during execution. . . . . . . . . . . 46

5.3 The access information of the loop in Figure 5.2. “Sh_arr” is the only accessed shared data structure. This data structure is assumed to have already been registered as shown in Figure 5.1. The data structure is modified. Hence the access_type is set to WRITE. Dimension 1 of the array is always accessed from element 0 through element M with subscript “i” (“1 * i + 0 = i”). Dimension 0 of the data structure is always accessed between elements “start” and “end”, which are determined during the execution. Therefore, these values are set at each instantiation of the loop, while other values are set only upon the first entry. . . 47
5.4 A simple nearest neighbor averaging loop. For each element of $sh.a$, four neighbors of that element are averaged and the result is put in $sh.b$, then $sh.b$ copied back to $sh.a$. This operation is repeated for 100 times. 48

5.5 Parallelization of the loop in Figure 5.4 using cyclic distribution. Each process iterates over rows $i$, $i+P$, $i+2P$, ..., with $i$ starting at pid+1 of the respective process. Since each row is modified at each execution of the loop, rows $i-1$ and $i+1$ needs to be brought up-to-date before the loop can proceed. This would require communication to receive the two rows modified in the earlier iteration for each row a process works on. 49

5.6 Parallelization of the loop in Figure 5.4, using block distribution. Each process iterates over rows $start$ to $end$ and therefore needs to exchange at most two rows with other processes. 50

5.7 A simple loop operating on a two dimensional, $M \times M$ array. Since the loop at line L03 starts from "i" rather than a constant number, the number of rows operated on is decreased by 1 at each iteration of the outermost loop. Similarly, since the loop at line L04 also starts from "i", the number of columns being processed decreased by 1 at each iteration of the outermost loop. 51

5.8 Parallelization of the loop in Figure 5.7 using block distribution. Each process starts working on all rows between "start" and "end". However, once the index of the outer loop ("i") reaches the value of "start", the process loses one row at each iteration. Once the index reaches the value of "end", the process would not have any data to work on. This will create load imbalance among processes by having some processes spin idly, while others have to work on higher order rows. 51
5.9 Parallelization of the loop in Figure 5.7 using a cyclic distribution. Each process starts working at row "start" and skips P rows at each iteration. Each process loses one row of its assignment every P iterations of the outermost loop, making all processes finish execution of the loop at almost the same time. ............................................ 52

5.10 The LADLE formatted parallel version of the loop in Figure 5.7. The "while-loop" at line L03 is added for the loop to be executed in smaller tasks, making it possible to change assignments during execution. The entry point into LADLE is the call to LADLE.getNewTask() at line L03. 53

5.11 The LADLE formatted data definition for the loop in Figure 5.7. Every parallelized region is given a unique identifier shown here by "cur.region". The section between L04 and L25 is executed only once, at the first call of this function. The amount of data accessed by inner loops is variable across consecutive executions of the same parallel region and therefore it has to be set at each invocation. Therefore lines L26 and L27 set these bounds at each instantiation of the parallel region. The parallelized loop of the original function is shown in Figure 5.10 and called at line L29. At this point LADLE has all the information to divide the work, and execute the parallelized loop correctly. ....................... 54
6.1 An example task reassignment in a "STENCIL" parallel region. Initially all processes are of equal RelativePowers with (0.25; 0.25; 0.25; 0.25) as shown in (a). When RelativePowers change to (0.40; 0.25; 0.25; 0.10), indicating P0 is progressing faster, and P3 is progressing slower, the size of the chunk assigned to P0 is increased and the size of the one assigned to P3 is decreased, as shown in (b). Even though the sizes of chunks assigned to P1 and P2 do not change, their assignments change and they incur some communication overhead. However, considering the same data is likely to be iterated over multiple times, incurring one time overhead in the interest of reducing overall communication overhead due to steady state nearest neighbor communication could be well worth it.

6.2 An example reassignment in a region marked as having an "INDEPENDENT" sharing type and "FIXED" load. The distributed data structure is a two dimensional array, and two consecutive rows of this array are grouped together to form a task. Initial RelativePowers are (0.25; 0.25; 0.25; 0.25) for P0-P3 respectively, and each process is assigned 5 tasks (10 rows). When RelativePowers change to be (0.40; 0.25; 0.25; 0.10), 3 tasks of P3 are assigned to P0. Processes P1 and P2 are not affected by this move since their RelativePowers did not change and the sharing type is marked as "INDEPENDENT". Such reassignment would cause communication only between processes whose RelativePowers change.
A pseudocode algorithm to determine relative processing power for applications that use barriers for synchronization. Since work assignment to each process is proportional to its previously known relative processing power, and arrival at a barrier signals completion of that assignment, new relative processing powers could simply be calculated by dividing the earlier relative processing power of each process by the time it took for that process to finish its assignment. TaskTime has the time it took each of the processes to finish their earlier assigned work. RelativePower values are normalized to ensure their sum is 1.

Computing 'RelativePowers' without relying on any type of synchronization being used. The challenge is to determine the amount of completed work by each of the processes. The section between lines L07 and L16 does just that. The minimum average time spent in a parallel region working on one task is assumed to be the time tasks of that region can be completed at. Once that value is found, it is multiplied by the number of completed tasks by each of the processes. The same set of operations is repeated for all of the parallel regions, finding the amount of work completed by all the processes. Since it is known how much time is spent by each of the processes, computing RelativePowers is straightforward. Once RelativePowers are computed, they are normalized to ensure their sum is 1.

A simple program section that shows an application getting a pivot row, and applying that pivot to a bunch of tasks. Each pivot is computed by one of the processes, and its availability is signaled to others by setting a flag. Once this flag is set, all other processes can read and apply this pivot to their tasks.
6.6 A typical use of the provided interface, when many smaller pieces are created and each piece is associated with a token. In our example, we assume token 0 is associated with particles 0..numParticlesPerToken-1, token 1 is associated with particles n..2*numParticlesPerToken-1 and so on. .......................................................... 79

7.1 Average message latency as a function of multiprogramming and message frequency. Each of the graphs shows averages for a different number of competing processes on the message recipient side, changing from 0 to 3. The 4 groups in each graph shows results with different number of processes competing with the sender. The average amount of computation is varied from 0.5ms to 17ms, and each of the marks in each group shows the result for that. For each computation amount, the same experiment is repeated 8 times. The bars indicate the smallest, average and largest (i.e., the spread) of the average message latencies of each of the runs. Note that the scale of each graph is different. .......................... 86

7.2 In a priority based scheduler, compute-intensive processes of equal priority are scheduled in a round-robin fashion. The percentage of messages answered immediately between two processes is directly proportional to the time the two processes run simultaneously. In each case there are two nodes, and change of shade indicates a context-switch. Assuming processes of the same shade are communicating, the processes in “Case 1” are coscheduled, while the processes in “Case 3” are running completely out of sync. ................................. 87
7.3 Effectiveness of FCS in reducing the message latency in the presence of multiprogramming. The experiments here are a subset of those in Figure 7.1, showing only the cases with number of processes competing with the sender and receiver being equal. Similar results were obtained for all other cases, and were eliminated due to redundancy. The graph shows the cumulative distribution of the fraction of the total number of messages with a given latency, for 99% of the messages. Understandably, in the case of no competitor, all messages are responded to almost immediately. However, even in the case of 3 competitors on each side, about 70% of the messages are responded to immediately, with 99% of all the messages being responded to within 650μs. Some messages of the remaining 1% took in the order of milliseconds, due to FCS refusing to schedule the recipient process, as it would have been unfair to do so.

7.4 Average barrier time on 32 processors, with different number of processes running on the same processors as the processes of the parallel application. The 4 groups indicate the average barrier time for cases with the number of processes competing with each process of the parallel application varied from 0 to 3, with the leftmost group being 0 and the rightmost being 3. The x-axis ticks show the average amount of computation between each barrier.

7.5 Graph of the experiment shown in Figure 7.4 with FCS support being turned on. FCS, reduced the average barrier time considerably, in addition to making it predictable (i.e. independent of the amount of multiprogramming.)
7.6 Overhead percentage when running applications on 32 processes, with 16 of the processes having a competing process running alongside them. Ideally, an application taking "t" seconds on a dedicated cluster should take "2t" seconds when 16 processes have a competing process running alongside them. Any amount of time above "2t" is assumed to be overhead, and that amount is divided by "2t" to find the overhead percentage.

7.7 Execution times of applications under different scenarios. The x-axis labels indicate the type of load, as explained in Table 7.2. The 6 bars in each group show the time it takes to execute the application under a different setup. The first bar of each group is when nothing is done. The second bar shows the case when FCS support is turned on. The third bar shows the effect of task stealing for intra-node load balancing, along with FCS. The fourth bar is LADLE-IND (task stealing and load balancing, independent of synchronization) without FCS. The fifth bar is LADLE-BAR with FCS, task stealing, and load balancing turned on. The sixth bar shows the case of LADLE-IND with all the features turned on, including FCS. Finally, the rightmost bar shows the ideally achievable execution time, and is given as a reference for comparison. The ideally achievable time is calculated as a factor of the completion time on a dedicated machine without any load balancing or task stealing. For instance, in the case of LOAD-4, since 4 of the processors are time-shared, the parallel application nets 28 dedicated and 4 half-time processors, totaling 30 processors. Ideally, the execution time of a parallel application running in such a configuration can be at best (32/30) of its execution time on a dedicated cluster.
7.8 Execution times of parallel applications in the presence of a memory intensive competing application. All the experiments were conducted using 32 processors, with 16 of them running the memory intensive application along with the timed benchmark application. The 16 multiprogrammed processors are on 4 of the SMP nodes, leaving the remaining 4 SMP nodes dedicated. Therefore, there is no load imbalance within nodes. The labels in the legend are the same as those shown in Figure 7.7. .................................................. 101

7.9 Evaluation of FCS, LADLE-BAR, and LADLE-IND in a highly dynamic environment, with many I/O and memory intensive short-lived processes. In order to get a reasonable reading of the performance, each application is made to conduct its computation multiple times to take sometime between 1500 and 1700 seconds. ........................................ 103

7.10 Normalized execution times of parallel applications when they are competing with another parallel application. Experiments are conducted using six 4-way SMP nodes totaling 24 processors with three nodes (12 processors) being multiprogrammed by running another parallel application continuously. The leftmost bar is the base case where the timed application has all the cluster to itself, and it is always at 100. The middle bar shows the normalized execution time in the presence of a competing application and neither LADLE nor FCS is switched on. The third bar shows the result when all the features are turned on. ... 105
7.11 Evaluation of LADLE-IRR. The x-axis indicates the number and geometry of competing processes as described in Table 7.2. The leftmost bar of each group shows the execution time when no runtime load balancing or scheduling support is present. The second bar from the left is when FCS is turned on without load balancing, and the third bar is LADLE-IRR including FCS. The fourth bar is the ideally achievable execution time, as described in Figure 7.7.
1 Introduction

A cluster of workstations (COW) is built by connecting a number of individual workstation nodes with a high speed local area network (LAN) or system area network (SAN). Such environments offer impressive processing capability at a particularly low price, since they are built completely from relatively inexpensive off-the-shelf components and require little or no extra engineering effort. The total cost of ownership of such clusters is further reduced by factoring in their low operating cost.

Compared to large scale Massively Parallel Processors (MPP) or Symmetric Multiprocessors (SMP), COWs have the advantage of using current technology, whereas MPP systems might use components that are one or more generations out of date. The reason for such delay in MPPs is the long time it takes for them to be designed and deployed. Furthermore, unlike MPPs, COWs can be partially and continuously upgraded, making them particularly suitable building blocks for an expandable system [6]. Nothing in their architecture necessitates the use of homogeneous hardware, thus allowing owners to maintain several generations of hardware and operate them seamlessly, from a system administration standpoint.

Another advantage of COWs is in the area of fault containment. It is desirable to have the effects of a faulty node confined to the node itself. Unfortunately, in hardware shared memory multiprocessors, providing fault containment is not a trivial is-
sue. A faulty node in such multiprocessor machines might incorrectly modify and corrupt other nodes’ memory (issuing wild writes), causing a system-wide crash [14]. When running parallel applications on COWs, the “native” programming model is message passing, with each node doing its own local memory management. Updates to shared data structures, and synchronization between collaborating processes, is handled through the exchange of messages over the interconnect. Because messages, by default, are not trusted, wild writes are virtually impossible. Furthermore, COWs do not have the single point of failure introduced in systems with central control. Thus, fault containment on COWs is trivial. Even in cases of parallel applications running in such environments, an application is affected only if it has a thread running on the failed node.

1.1 Challenges of Using Clusters of Workstations

Despite their many advantages, COWs present a number of new challenges. Although they can potentially provide a large number of cycles, using the cycles efficiently and effectively is a challenge on its own. In particular, lack of access control allows arbitrary job submission and may result in (potentially uneven) multiprogramming of these COW nodes. Incremental upgrades implies potentially unequal resources (processing, memory, I/O, communication bandwidth) among nodes. The program itself may have poor parallelism, and may be hard to distribute evenly across nodes.

Writing parallel programs on COWs requires the exchange of messages in order to synchronize and communicate updates to the data shared across nodes. Often, a message exchange requires an action on the part of the party receiving the message. Any delay in the execution of the required action will cause delay in other processes as well. Response to a message will be delayed if the process that is expected to act is not running when the message arrives. In the presence of multiprogramming, and absence of coordination between communicating processes, it is inevitable that processes will
be forced to wait for their communication partner to be scheduled. In the absence of job admission control to prevent multiprogramming, a possible solution to this problem is to schedule all the processes of the same application simultaneously — a method known as coscheduling, or gang scheduling.

Synchronization, on the other hand, requires all synchronizing processes to be at a certain point in their execution. Late arrival of one of the synchronizing processes to a synchronization point forces other synchronizing processes to wait. Such late arrivals may arise either due to uncoordinated scheduling as described above, or due to load imbalance. Load imbalance occurs when work is assigned disproportionately to available resources. Disproportional work assignment could be a result of three, potentially interacting, factors:

1. differing levels of multiprogramming on equally powerful nodes
2. equal amounts of work assigned to heterogeneous nodes that have different capabilities in terms of either CPU, memory, or network bandwidth, etc.
3. unequal work assignment to processes of the same application running on identical nodes

The potential solution is to assign load proportional to available resources, through a process called load balancing. Both load balancing and coscheduling techniques have been studied extensively.

1.2 Summary of Prior Work

Both load imbalance and scheduling discrepancies are well-known and have been studied extensively, though always in isolation and not in unison. Since in a COW environment the problems are likely to arise jointly, it is particularly important to find holistic solutions. Unfortunately, coscheduling and load balancing alone not only do not offer such solutions, but also sometimes fail to work together.
Efficient coscheduling requires the number of processes on each of the processors to be equal [28, 64]. As a result, different levels of multiprogramming is not only a cause of load imbalance, but also a reason for ineffective coscheduling. Furthermore, implementing coscheduling in a cluster of workstations is hard, due to the absence of any central control needed for simultaneous context switch by all the cluster nodes. Imprecise approximations of coscheduling with implicitly available information have been implemented [22]. However, similar to explicit coscheduling, these techniques are sensitive to different levels of multiprogramming.

Load balancing targets making load assignment proportional to available resources. Determining the “load” and the amount of available “resources” is central to any load balancing scheme. Centralized task-queue load balancing schemes put all the work in a central repository, from which processes compete to get work. All processes access that central repository within a critical section, to get as much work as they have resources. However, such an approach is not suitable for cluster environments, due to the cost of maintaining such a central task-queue. Furthermore, such approaches disregard the locality of the data, which is too expensive to be disregarded. In the interest of preserving locality, load balancing methods with per-processor task-queues were devised and implemented. However, such methods are generally designed to work on hardware shared memory environments, and are not suitable for clusters of workstations, either. Process migration schemes for load balancing, in general, take each process to be a unit of load, and attempt to make the number of processes on each node be proportional to the amount of resources that node has. Unfortunately, processes are required to conform to a long list of requirements (e.g. not having pinned memory, not having shared memory, etc.) in order to be eligible for migration. Also, in general, process migration is too coarse grain to be useful for load balancing in a parallel computing environment. Besides, using the load index as a metric is not suitable since such a metric either assumes all nodes to be homogeneous, or uses a scaling factor to determine the relative resources available on one node relative to another. Such a scaling factor is not realistic,
since nodes might have more of one resource (e.g. memory) but less of another (e.g. floating point processing).

Since load imbalance due to multiprogramming includes scheduling issues, addressing load imbalance and scheduling skew simultaneously is important. However, a simple combination of (perfect) coscheduling with (even a perfect) load balancing system may not yield the desirable result. Load balancing on a set of identical workstations with processes of a parallel application assigned equal amount of work, would cause load imbalance only when the number of processes on each node differs. However, coscheduling would be most effective when the number of processes on each of the nodes is equal. Therefore, a simple combination of the two methods would not handle all the scenarios simultaneously.

1.3 Proposed Solution

**Thesis Statement:** Efficient use of resources in a Cluster of Workstations, especially when heterogeneity and multiprogramming exist, requires addressing issues of load balancing and coordinated scheduling simultaneously. It is possible to balance load and coordinate scheduling while retaining scheduling fairness. Our proposed solution combines fair cooperative scheduling with locality aware dynamic load balancing, using progress as the base metric, and provides an effective means for parallel applications to use cluster resources efficiently.

Our proposed solution is composed of three components, and it targets inefficient use of resources, regardless of what caused them to arise in the first place.

We do provide scheduling support, not in the form of coscheduling, but in what we call “coordinated scheduling”, a cooperative scheduling scheme, in which “well-behaved” processes are allowed to get the processor on demand, so that they can provide services to their peers. The scheme is called Fair Cooperative Scheduling (FCS), and stresses fairness as an important part of the policy.
Our runtime system, Locality Aware Dynamic Load-balancing Environment (LADLE), balances load and manages locality, and interacts with FCS for the purposes of getting scheduling support to reduce the scheduling induced wait. For the purposes of load balancing, LADLE uses progress as the basic metric. The fundamental idea is to move work from a process progressing slower than its peers, to a process progressing faster. Progress is represented with relative processing power — the amount of work that can be finished by a process within a unit of time. A metric based on progress is independent of the underlying cause of load imbalance. Any cause would be equally represented, as faster progress by one process indicates that process has access to more of the needed resources, and therefore can finish more of the assigned work in a given unit of time.

The third component is static analysis of the parallel application, with the aim of extracting information to be used by LADLE. LADLE balances load by moving work from one process to another. Moving work invariably means moving data, and the ability to do so requires knowledge of the data structure. The static analysis component provides the necessary information to the runtime system, so that such data movement decisions could be made in a locality-preserving manner so as to reduce communication.

Our implementation of the proposed techniques has shown that it is possible to tackle load imbalance and scheduling issues simultaneously. The proposed coordinated scheduling technique has reduced the per-message round-trip latency from a high average of above 30\(ms\), to below 100\(\mu s\). In a 32-processor case, the average per-barrier time was also reduced from a high of above 40\(ms\), to below 1\(ms\), when the number of processes competing with each of the parallel processes was 3. In the absence of multiprogramming or any other source of load imbalance, the overhead added by the runtime system was negligible. In the presence of multiprogramming, however, the execution time of parallel applications was reduced by as much as 40\%, compared to having no runtime support.
1.4 Dissertation Organization

The remainder of this dissertation is organized as follows: Chapter 2 delves into background, describing previously proposed solutions, as well as their successes and shortcomings. Chapter 3 explains our overall approach to the problem. Chapters 4, 5, and 6 explain each of the components of our solution in more detail. Chapter 4 explains design goals and implementation of our scheduling policy, Fair Cooperative Scheduling (FCS). Chapter 5 describes our static program analysis and the code transformations we use for communicating statically extracted information to our runtime system. Chapter 6 describes the details of our runtime system, Locality Aware Dynamic Load-balancing Environment (LADLE). Chapter 7 evaluates our system for a range of applications under different workload environments. Finally, Chapter 8 presents conclusions from our work, and discusses promising future research directions in light of the results we present.
2 Background and Related Work

Clusters of workstations (COWs) offer impressive compute power at a relatively low price. Advantages of owning such clusters include their ability to be built incrementally (expandability) and their natural fault containment due to their decentralized structure. However, expandability could bring heterogeneity, as the newly added nodes are likely to be different from the original cluster. The decentralized structure might bring uncontrolled multiprogramming. When parallel applications are concerned, each of these reasons causes load imbalance, with some of the processes of the parallel applications having work disproportional to the resources available to them relative to their peers. Load imbalance could also arise due to inherent application behavior, such as lack of parallelism in the application, or programmer errors. Multiprogramming also causes scheduling issues by having some of the processes of the parallel application descheduled when some of their peers need to communicate or synchronize with them.

In this chapter, we will discuss earlier work that addresses these issues. Issues related to load imbalance and lack of coordinated scheduling have been looked into extensively, although in isolation. Load balancing techniques will be discussed first, followed by coordinated scheduling methods. The effects of load balancing on locality will be discussed as well to establish its importance, particularly for modern architectures.
2.1 Load Imbalance

In general, the goal of load balancing is to make the proportion of work to resources equal among all units. In this context, a "unit" could be a node, a processor, or a process that is intended to have an equal amount of work relative to its counterparts. Similarly, "work" could be the number of threads, or processes, or any other entity that needs to be balanced across the selected units.

Parallel applications synchronize in order to impose an order among accesses to shared resources. Each synchronization point can be viewed as a rendezvous, where all synchronizing parties have to meet. In the presence of load imbalance, some of the synchronizing processes may finish their work after their partners and arrive late at the rendezvous point. In the context of parallel applications, the goal of load balancing is to have all processes of an application reach their rendezvous points simultaneously. Thus, in the case of parallel applications, the "units" involved in the load balancing are processes of the parallel job\textsuperscript{1}. Balancing load for parallel jobs invariably involves migration of work and transfer of data between processes, which in turn increases the amount of communication necessary. In current architectures, fetching data from processor memory is much slower than fetching from cache. The gap between the memory and processor is likely to grow wider, making it even more costly to go to memory [41]. Transfer of data not only destroys cache locality, but also adds the large overhead of transmitting data over relatively slow networks. The fastest network tends to be several orders of magnitude slower in terms of latency and bandwidth than the same generation of memory access. Therefore, balancing load without regard to locality of data is likely to hurt performance rather than helping. The issue of load imbalance, as well as techniques to balance load and manage locality, have been investigated in the literature extensively, especially in the context of loop scheduling [26, 46, 48, 49, 66, 78, 87].

\textsuperscript{1}In this dissertation, the terms "process" and "thread" will be used interchangeably. Without loss of generality in our argument, the parallel application could very well be written using multiple threads of control running in the same address space.
In the following subsections, the effectiveness and limitations of prior load balancing approaches will be discussed briefly. In order to establish a base case, initially static partitioning of work among processes will be also discussed.

2.1.1 Static Partitioning of Work

Perhaps the simplest way to write a parallel application is to statically split the work among the number of processes, without regard to what might change in the course of execution. However, its simplicity comes at a price; such an approach is vulnerable to any dynamic changes in the environment. In the case of homogeneous clusters, no multiprogramming is assumed, and all nodes are assigned equal amounts of work. Although it is fairly easy to work with such an assumption, in reality it necessitates much stricter access control, perhaps causing a loss of performance, and, more importantly, creating a single possible point of failure. Furthermore, expandability, a major advantage of COWs, has to be limited for such an approach to work.

Of the static approaches, Block Scheduling divides the work into blocks of loops of fixed sizes and assigns each of these blocks to a process. Cyclic Scheduling, on the other hand, assigns loop iterations to processes in a cyclic order, so that process $p$ will execute iterations $p, p + P, p + 2P, \ldots$ and so on, where $P$ is the total number of processes in the system. This sort of scheduling tends to perform better if the amount of work in each iteration increases (or decreases) monotonically. The main motivation is to prevent the iterations with the smallest amount of work and those with the largest amount of work from grouping together. However, since none of the iterations are scheduled with their neighbors anymore, this type of distribution might perform very poorly if nearest-neighbor type of communication is exhibited by the application. Block cyclic scheduling lies somewhere in between block scheduling and cyclic scheduling. A block of size one means cyclic scheduling whereas a block size of $N/P$ means block scheduling. The grouping into blocks is intended to reduce the number of boundaries,
while still guarding against iterations with monotonically increasing (or decreasing) work [86].

There has also been work that statically addresses heterogeneity in a COW. The fundamental motivation of such an approach is the rapid increase in processor speed and the potential for partial upgrades. When all the nodes of the cluster are assumed to be dedicated to the parallel job, and the configuration of nodes is not changing, statically allocating work in a manner proportional to available resources tends to help most, and without any runtime overhead [36]. Furthermore, since there is no runtime data movement for the purposes of load balancing, there is no additional communication, either. The issue of "heterogeneous loops", in which each loop iteration might have different amounts of work, has also been addressed. In such cases, the amount of work assigned to each of the processors is computed not in terms of number of iterations, but instead by combining multiple iterations to form groups with almost equal amounts of work [15]. However, it is clear that such methods are not intended for (and do not work well in) dynamic environments, such as multiprogrammed nodes. Furthermore, even though heterogeneous loops are addressed, the approach would work only if the work per iteration can be determined statically and is not a function of the content of the data being processed.

2.1.2 Centralized Task-Queue Based Load Balancing

Of the dynamic load balancing techniques, centralized task queue-based schemes are perhaps the ones explored most. Their ease of implementation is the main attraction. The general idea is to have all the iterations of a loop reside in a central repository, and have all the processes of the parallel application compete in order to grab some iterations. Each time a process accesses the queue, it gets a piece of the remaining work, called a task. This way, load imbalance can be at most the longest time it takes to complete the last task grabbed by any of the processes. Furthermore, faster processes
would access the queue more frequently and do more work than their slower counterparts. Since a centralized task queue is employed, access to the queue has to be in a critical section, which requires usage of a lock. The smaller the task, the less the potential load imbalance upon depletion of the task queue. However, the larger the task, the smaller the number of lock acquisitions needed. Both reducing the number of lock acquisitions and potential load imbalance is crucial. Therefore most variations of this general idea deal with finding a way to reduce the number of lock acquisitions, while preserving the potential to balance load.

The pure self-scheduling approach assigns one iteration per processor until all the iterations are consumed. This approach would clearly have the least load imbalance, while having the highest loop scheduling cost due to synchronization. In a compromise to reduce the synchronization overhead, while keeping the load reasonably balanced, fixed-size chunking was introduced. This scheme makes each of the processors take K iterations at a time rather than one, reducing the number of chunks from \( N \) to \( N/K \), with \( N \) being the number of iterations [50]. The assumption made by fixed-size chunking was to take iterations as identical independent random variables. Based on this assumption, a formula was derived to calculate an optimal chunk size (\( \kappa \)) in terms of mean and variance of iteration execution times, and the number of processors. However, since the number of iterations (\( N \)) and number of processors (\( P \)) can change, determining the right \( \kappa \) is hard, and is left to the programmer.

Since load imbalance is a factor of the size of the last chunk grabbed by each process, it was observed that it is better to make the chunk size smaller as the computation progresses. This way, it would be possible to reduce the overall number of chunks, while addressing the issue of load imbalance effectively. Several methods have been proposed that change the chunk size dynamically, without any user input. Guided self-scheduling [66] changes \( \kappa \) so that a process gets \( 1/P \) of the remaining iterations, where \( P \) is the number of processors. This implies the first process takes \( N/P \) iterations, the second process takes \( (N-(N/P))/P \) and so on. Trapezoid self-scheduling [81], on the
other hand, sets the largest and smallest chunk sizes and computes a fixed decrement value $D$. Each new task has $D$ iterations fewer than the earlier one.

*Factoring* [43, 44] is similar in spirit to the *self-scheduling* techniques described above. However, instead of computing the size of a single chunk at a time, the size of $P$ chunks are computed, where $P$ is the number of processes. The total size of each newly created set of chunks is expected to be a pre-specified fraction of the total work. This fraction is decreased for each new set, and once a new set is created, all the chunks are placed at the head of the task queue. The fundamental intuition is that when all iterations are of constant length, all the processes have an equal amount of resources, and all processes are started simultaneously, each of the processes will be executing $N/P$ iterations. This is not a feature easily achievable by previously mentioned techniques.

None of the loop scheduling techniques discussed so far has dealt with locality of the data. If the same parallel loop is to be executed multiple times, the same data might end up being processed by different processors. Given the cost of transferring data from one processor's cache to another, it is crucial to have the owner of the data do the computation, if possible. Studies as early as 1990 have shown that disregarding locality of data by employing a centralized task queue could cause a performance penalty of up to 99% on shared memory multiprocessor machines, with 69% of it being due to cache reload [75, 76]. This figure can only be expected to increase, due to the increasing speed gap between processors and memory. It has also been found that when load imbalance exists, even though both load imbalance and locality management should both be considered, locality management should be given higher priority [59]. In a COW environment, transfer of data between two processors is even more expensive, since the interconnecting network has higher latency and lower bandwidth. This calls for attributing even higher priority to locality management.
2.1.3 Multiple Task Queues and Emphasis on Locality

Affinity scheduling [58] makes reducing of synchronization time and preserving locality the central issue. In this algorithm, per-processor task queues exist rather than a centralized task queue. Each processor is initially assigned a chunk of size \(\lceil N/P \rceil\), where \(N\) is the number of iterations in the loop, and \(P\) is the number of processors. Processors get \(1/k\) of the available iterations in their task queue until they deplete it, where \(k\) is an externally provided parameter of the algorithm. If, by the time the local queue is depleted other processors still have some iterations to work on, the processor with the largest number of iterations is found and \(1/P\) of its iterations are grabbed to be executed. The significance of this approach is to preserve locality of data as much as possible. Furthermore, even though accesses to each of the task queues are still protected by a lock, existence of per processor queues reduces the contention. Two processes would compete for access to a queue only if the owner of the queue is lagging behind, and there is another process that has already finished its assignment.

Locality based dynamic scheduling (LDS) [52] is very similar to affinity scheduling. It groups iterations in small chunks using any of the algorithms specified by the approaches mentioned above. During computation, a subtask size \(S\) is determined first (e.g., \(\lceil n/(2P) \rceil\), where \(n\) is the number of total remaining iterations). If the processor has \(r\) iterations in its local queue, then a subtask of size \(\min(r, S)\) is executed, otherwise the processor with the largest number of iterations, \(r_{\text{max}}\), is found, and a subtask of size \(\min(r_{\text{max}}, S)\) is executed.

2.1.4 Non-Task Queue Data Migration Techniques

Even though task queue-based load balancing techniques are effective in hardware shared memory environments, none of these techniques are suitable for a COW environment. Access to shared task queues requires some form of synchronization. Synchronization over a network, in most cases, is prohibitively expensive. Therefore, finding
methods for balancing load without the excess need for communication and synchro-
nization is of great value.

Adapt [45, 55] combines static and runtime analysis to relocate data in a COW in an
attempt to reduce communication and thus execution time. The number of page faults
incurred provides hints for data relocation. Adapt relies on the existence of a software
distributed shared memory (SDSM), which implicitly converts page faults to page fetch
requests over the network. Therefore, measured network traffic is directly proportional
to the number of page faults. If the amount of communication is high, it is perceived
as an indication of the need to redistribute data. Also, the time it takes to complete
one iteration of computation is compared between each of the nodes. High variations
in these times is taken as an indication of load imbalance, and again perceived as a
need to redistribute data. Collected results are interpreted at global synchronization
points (barriers) and new data distributions are made. Data is binned as either blocks
of iterations or cyclically grouped iterations. Cyclic distribution is chosen if it was
observed that the time it takes to complete execution of a parallel loop monotonically
increases or decreases for consecutive adjacent loops.

CRAUL is also a system that combines static analysis and dynamic load balancing.
It uses progress to determine the required load reassignment, and is the predecessor of
the work in this dissertation [47]. The fundamental idea is to move work from processes
that take longer to finish their assigned work. The granularity of monitoring is a phase
of computation between two global barriers. Using progress as a metric has the advant-
age of being independent of the cause of load imbalance. It could be due to heteroge-
neous hardware, multiprogramming, or application behavior. To avoid being fooled by
short time spikes, the application is monitored for a certain minimum time. Once that
certain time is exceeded and a global synchronization point is reached, the amount of
work completed by each of the processes is compared. Based on this comparison, the
relative processing power of each process is computed, with the process spending the
least amount of time per iteration having the highest value. New work assignments are
made such that each process receives iterations proportional to the computed relative processing power. Similar to Adapt, CRAUL works only with applications that employ barriers. However, unlike Adapt, the type of distribution is always block. Therefore, reassignment means shifting the boundaries of the assignment to each process. Unfortunately, that causes unnecessary data movement, since it might change the boundaries of processes whose relative processing power remained the same.

Both Adapt and CRAUL reassign work only at a barrier. Reassigning work only at a barrier has two major disadvantages: 1) applications that do not use barriers can not balance load, even if they are running under extreme load imbalance. 2) Even applications that use barriers may have to run unbalanced for long periods of time, if the time elapsed between two barriers is long.

2.1.5 Process Level Load Balancing Across Clusters

Multiprogramming is also known to cause load imbalance. For a parallel application running on P processors, it is not unusual to have some subset of these P processors being used by some other processes, particularly in a COW environment where access control to the machines is lax, at best. Besides the methods mentioned above, there have been some studies to balance load especially in cases of multiprogramming. Most of these ideas are centered around migrating processes from overloaded processes to underutilized ones. Process migration is a difficult technical problem with many unresolved issues [12, 19, 63, 88]. However, even if all the underlying migration related technical issues are solved, load balancing adds many other complications: it is not trivial to decide what process to migrate, when to migrate, and where the migrating process should be relocated. Many different, sometimes contradictory, conclusions have been drawn by different researchers, with respect to the effectiveness of employing process migration for the purposes of load balancing. Eager et. al. contended that the performance benefits of migrating active processes is limited [25]. However, their study was
criticized, claiming the model used was unrealistic [20]. Another study concludes it is actually beneficial to migrate active processes, given that process arrival and lifetime is well understood and correctly modeled [39]. Yet another study concludes that the effectiveness of process migration for load balancing was heavily related to the assumed model of process lifetime [62]. In this study, it was concluded that only those CPU-bound workloads that were generated using an unrealistic Poisson distribution for execution times show improvements with dynamic load balancing using process migration. However, when more realistic workloads and short lived tasks were added, process migration algorithms failed to be effective as a tool of load balancing [62].

Condor [53] is a system intended to harness cycles from idle workstations. At a high level, it is a batch system intended to be used for compute-intensive jobs. Condor allows users to group processes of a job, prioritize these jobs, and specify how each of these jobs must be run. A combination of centralized and distributed load balancing is used, whereby each workstation is responsible for scheduling its own jobs while the job assignment to each node is performed at a centralized controller. Thus, the central controller periodically polls each workstation and collects state information consisting of a node’s current load as well as the number of jobs in the node’s queue. This information is used to compute a workload index at the central controller. The metric being used is simply the number of runnable processes averaged over a period of time. Load balancing is achieved by using those workstations with the lowest load-index to start a new job. Additionally, Condor supports checkpointing, whereby a disk image of an active process is stored on the workstation in order to facilitate both fault tolerance and process migration. Process migration is also used as a tool for load balancing where a process could be migrated from a heavily loaded node, to a lightly loaded one. For a process to be fit for migration, it has to be linked with a special library and has to refrain from using some types of interprocess communication mechanisms, such as shared-memory [79]. The process checkpointing and migration subsystem of Condor is available separately and is used as a building block by many systems, including IBM’s
Load Leveler and Load Sharing Facility [54].

IBM’s Load Leveler [16], on the other hand, controls the job submission process in an attempt to balance load. Load balancing is achieved by controlling the number of processes on each of the nodes, and matching the demand of the process with what is available on a particular node, or set of nodes. This approach, for instance, would try to dispatch processes with high memory demand on nodes with high memory supply. All the machines and their resources need to be registered in order to become members of the managed cluster. Once a job is started, its processes could be migrated. Similar to Condor, migration is achieved by checkpointing and restarting. Load Leveler uses Condor’s mechanisms for checkpointing and migration, and requires user involvement during the decision process.

Platform Computing Corporation’s Load Sharing Facility (LSF) [17, 90] and its predecessor UTOPIA [91], have been successfully deployed for the purposes of coarse-grain load balancing of sequential or distributed applications. LSF also attempts to match the application requirements with the node or set of nodes that the application is dispatched to. For the purposes of load balancing, LSF uses load-index scaled to the relative speed of the processor, with the speed of the fastest processor being the base. The ability to scale this metric requires statically registering the ratios of the speed of the processors. LSF also uses Condor’s checkpointing and migration subsystem in order to provide fault tolerance and the ability to relocate processes in order to balance load.

Global Layer Unix (GLUNIX) [32] runs on COWs and offers load balancing through “intelligent” initial job placement. Processes of newly arriving jobs are distributed to nodes in a manner to make the number of runnable processes on each node equal. A unique characteristic of GLUNIX is that it restricts itself to user space and does not attempt to modify any kernel structure. Even though dynamic process migration for the purposes of load balancing was initially envisaged, it was later eliminated due to its complexity and portability issues.
MOSIX [7, 8] is very similar to GLUNIX in terms of running environment and goal. However, unlike GLUNIX, which restricts itself to user space, MOSIX requires major modifications to an OS kernel. The initial implementation of MOSIX was with the BSD operating system [60], and later was ported to the Linux operating system. The system employs process migration transparently as a tool for load balancing. However, the process migration approach of MOSIX is fundamentally different than that of Condor. In MOSIX each process has a unique home node (UHN). During migration, MOSIX splits the migrating process into two contexts: the user context and the system context. The user context, which contains the program code, user stack, data, memory-maps and registers of the process, will be moved to a remote location depending on the MOSIX resource sharing algorithms. The system context, which contains the resources attached to the kernel stack, will remain in the UHN of the migrating process. After migration, interaction between the user context and the system context will be intercepted by a link layer, which sets up a special communication channel for interaction.

There is no process migration system that is capable of migrating all types of processes. Condor based process migration systems require any migrating process to conform to a long list of conditions, such as being linked with special libraries and not using pinned memory. Furthermore, it requires a parallel job to have the same number of processes and process geometry (the processes that are common within a node) at its restart point as when the job was checkpointed. Even though MOSIX is less restrictive in terms of migration, it still has limitations. For instance, any process conducting interprocess communication through shared memory, or having pinned memory (which many user-level networks require) can not be migrated. Furthermore, process migration under MOSIX incurs large overheads due to leaving some portion of the process at its home node. Such a split requires all system calls to be forwarded to the UHN, and the results be sent back to the process making the call.

However, aside from the technical issues surrounding process migration, using this technique as a tool for load balancing, in most cases, is not even suitable for paral-
lel applications. To begin with, using "load-index", which is generally the number of runnable processes averaged over a period of time, as a metric for load balancing assumes homogeneous hardware. In order to alleviate this problem, LSF scales "load-index" to capture the power of the underlying hardware. However, it is not easy to scale such behavior, since the performance of a particular hardware changes dramatically based on the application. A quick look at the SPEC benchmarks shows that it is possible for one system to perform better than another on any given benchmark, while performing much worse on a different one [74]. Under these circumstances, finding the correct scaling factor is very hard at best, and in many cases practically impossible.

2.2 Coordinated Scheduling of Parallel Applications in the Presence of Multiprogramming

Effective scheduling of parallel applications has been a topic of interest for decades. Early studies have dealt with effective use of SMPs or MPPs, while more recent studies have dealt more with COWs. The existence of multiple jobs on a parallel machine has traditionally been dealt with in two steps: spatial scheduling and temporal scheduling [27]. The first step, spatial scheduling, is the act of assigning processors to a job. The second step is time multiplexing of multiple threads of control on the corresponding processor. Scheduling on a uniprocessor, on the other hand, deals with only the second step, and therefore is much easier to handle. Both of these dimensions has been dealt with extensively in the literature [1, 2, 28, 29, 35, 56, 84].

Effective space sharing requires complete control of the job submission process [89]. It also requires either an accurate estimation of job completion times [30] or the ability to dynamically change the number of processes of each job [80]. Even though complete control on job submission is technically achievable, accurate estimation of task completion times is not always achievable [33]. Dynamically changing partitions requires
applications to be written in a particular way, which requires fundamental changes to the way a parallel application is implemented. When that is achievable, space sharing has been shown to be superior to time sharing, even in the presence of coordinated scheduling [18].

The existence of a job occupying only a small subset of the processors might lead to delay of larger jobs requiring more than the available processors. Since the future can not be known, there is no solution to this dilemma. In the absence of future knowledge, processing power is lost because of fragmentation caused by either intentionally leaving some processors idle, in anticipation of arrival of large jobs, or due to forcing larger jobs to wait, while many, albeit less than needed, processors lie idle. It is possible to reduce the wasted computing power when job submission is made in batch mode, by reordering the submission queue [51]. However, such solutions easily deteriorate into bin packing problems, and even optimal solutions to a set of jobs may still leave unused processors.

The other dimension of scheduling parallel jobs, time sharing, is concerned with multiplexing the same processor among multiple processes of different applications. Uncoordinated scheduling of parallel processes has been identified as a major source of delay in synchronizing applications, due to forcing some of the processes to wait for their peers if they are not scheduled simultaneously [64]. Early work on shared memory multiprocessors has shown coscheduling (simultaneous scheduling of all processes of the same job, sometimes referred to as gang-scheduling [28]) to be effective in reducing the overall wasted processing power. However, as in the earlier studies, a complete control over the hardware and the ability to force simultaneous context-switch across all the processors were assumed. The earliest work assumed a global scheduler on a shared memory multiprocessor machine [37, 64]. MACH supports a mechanism to provide hints between cooperating processes as a means of reducing wait time [10]. Similarly, however, for that mechanism to work, a single OS must have total control of the multiprocessor machine. Even later work on distributed systems assumed the exis-
tence of a global synchronous heart-beat in order to provide a global signal to switch from one task to another [65]. Coscheduling suffers from another type of fragmentation: since each job may occupy a different subset of the available processors, some processors might have more processes than others. Making all the processes on all the processors equal and still keeping each job runnable as a group, again deteriorates into a bin-packing problem. Therefore, it is possible to have some quanta on some of the processors go unused, causing a waste of computing power.

All coordinated scheduling solutions mentioned so far require fairly complex algorithms and very tight control over the whole system. Attempts to simplify the solution traditionally have focused on either limiting the domain of applicability, or giving up on the precision of the solution. In our environment of interest, each node of a COW generally lives as an autonomous entity, and no global scheduling event exists. In the absence of a central decision making authority for simultaneous context switching, it is not possible to implement true coscheduling on a COW, without changing the characteristics of that COW. Furthermore, when running parallel applications on a COW, it is almost always the case that updates to shared variables are made by exchanging messages. This in turn makes every update to shared memory a rendezvous point, with potential for delay for each of the communicating processes. In many ways, in the presence of multiprogramming on a COW, applications experience a bigger problem (more rendezvous points), with less ability (lack of central decision making) to cope with these problems. Luckily, the problem has been very widely explored and numerous solutions addressing different domains have been proposed and implemented.

IBM’s “Load Leveler” [16] uses a precomputed scheduling matrix, disseminated to all the involved nodes, and synchronized clocks in order to achieve an approximation of coscheduling. Each column of the matrix corresponds to a processor and each row corresponds to a scheduling slice. Row boundaries are when context switches happen. All processes of an application are scheduled in the same time slice (scheduling row), even though it is possible to have more than one application in a row. The clocks are
kept synchronized with the help of the network clock. The mere existence of such a central authority defies the definition of autonomy, since it enforces a scheduling policy externally.

*GLUNIX* [32] provides a coarse approximation of coscheduling from the user space. A master process forms a matrix of all jobs, and sends messages to each of the daemons running on each of the nodes. The message specifies the job to be run. Upon receiving this message, each daemon sends a stop signal (SIGSTOP) to the process of the currently running job, and a continue signal (SIGCONT) to the process belonging to the job to be run. This scheme would work only if all the jobs were submitted through *GLUNIX*. However, there is no way of enforcing such a job submission policy. The time between sending a message and the time that message is picked up could be arbitrarily large, since the daemon needs to be scheduled before the message is picked up and acted upon. Signal delivery could also take an arbitrarily long time on many platforms. Therefore, the time-slice of parallel scheduling is changed essentially by having only one runnable user process per processor. Since only one process is runnable, the same process is scheduled back-to-back for many quanta until a new parallel time-slice is signaled by the master process.

A user-level distributed approximation of coscheduling, implicit coscheduling, has been shown to be effective [22]. The general mechanism of implicit coscheduling is to give up the processor if the process being waited for is known to be in preempted state. However, since the process being synchronized with is running on another node, it is not possible to know for sure whether the process is actually preempted or simply busy and not able to respond in time. Hence, some assumptions are made implicitly about the remote process' behavior, and therefore, the scheme is called implicit coscheduling. Deciding when to block for the processor to be used by another process is fundamental to the idea of implicit coscheduling. Experimentally it has been shown that spinning for a small period of time prior to blocking is much more effective than immediately blocking. However, deciding and adjusting the time to spin prior to blocking is not
trivial [21, 22].

Dynamic coscheduling (DCS) [72, 73] uses more explicit information to approximate coscheduling. The fundamental observation is that synchronization and communication events generate messages on a COW. Therefore, the goal is to schedule the intended recipient immediately upon receiving a message, if it is not running already. Unfortunately, DCS favors communicating processes over others by giving them the processor any time a message arrives. This in turn might starve non-communicating or coarse grain communicating processes, in favor of frequently communicating ones.

In this dissertation, we propose Fair Cooperative Scheduling (FCS) [67, 68], which places special emphasis on fairness. FCS demonstrates that it is possible to achieve coordinated scheduling without compromising fairness. FCS guarantees fairness, not only in the long term, but also in the short term, at a granularity of a single scheduling quantum: at no time does a cooperating process have more than its fair, proportional share. In order to communicate the need for coordination between cooperating processes, FCS uses messaging and synchronization events, similar to DCS. FCS also borrows mechanisms from implicit coscheduling regarding making decisions as to when to yield the processor. Applications are linked with a runtime library that yields the processor when it is not possible to make effective use of it due to waiting on some event. Rather than blocking, FCS yields the processor, therefore giving the processor up only if there is another process to take it. The approach is similar to the "Spin-Yield" mechanism [61], however, it allows the application to have more control over its willingness to yield. Design and implementation details of FCS are explained in Chapter 4.

2.3 Summary

Clusters of workstations offer many advantages as a platform for parallel applications. COWs are much cheaper in terms of hardware and software to obtain and maintain. However, economics is not the only reason for their attractiveness. COWs
are naturally expandable, since their structure allows addition of new nodes as the need arises. They offer natural fault containment since failure of a single node does not affect other nodes. Furthermore, lack of a single point of failure provides a basis for a fault tolerant system built on top of a COW. Despite these advantages, expandability and lack of central authority could cause load imbalance within parallel applications running on these clusters. Also, in the presence of multiprogramming, lack of coordinated scheduling could force a process to wait for its peers to be scheduled when there is a need to communicate or synchronize.

Both load imbalance and scheduling issues have been studied extensively, though always in isolation. Scheduling-centric solutions mostly concentrated on coscheduling, — simultaneous scheduling of all processes of an application — ignoring load balancing completely. Similarly, load balancing techniques ignored the scheduling issues all together. A simple combination of these two techniques is unlikely to work, either. Furthermore, most load balancing techniques ignored the locality of data, favoring load balancing at all cost. We argue that for a solution to be viable it has to address both load imbalance and scheduling issues together, while preserving locality of data as much as possible. Recognizing these dimensions of the problem, this dissertation introduces a technique where load balancing and coordinated scheduling of interacting processes are addressed simultaneously. Furthermore, the solution presented takes locality of data into account while making load balancing decisions, minimizing data movement when possible.
3 Design Overview

Our target environment is an autonomous cluster of workstations (COW). We use autonomy to mean that resource management decisions for each node are made locally at that node, and no central authority necessarily exists to decide job admissions.

In an autonomous COW, each of the nodes is freely accessible by any of the users, and each user is free to make use of available resources as he/she wishes. Users can start short or long living, parallel or sequential jobs. For a user, it is desirable not to need to know the details of the available hardware or resources in order to improve the performance of a parallel application. It is also desirable not to need to be aware of other users or their competing applications. Our design's adaptive nature aims to take care of all these factors, without any user involvement.

In Chapter 2, we discussed the fact that load imbalance and uncoordinated scheduling almost always arise together in a COW environment. Therefore our design goal is to address both of these issues together. Furthermore, it is also our design goal to preserve locality of data, preventing costly data movement between nodes of the COW.

The fundamental goal of our design is to devise an adaptive system that can handle dynamically changing multiprogramming, heterogeneous hardware, and changing application behavior, and adapt accordingly for the purposes of reducing the execution time of a parallel program and increasing the overall throughput. Also it is our de-
sign's goal not to favor any process or user over another (i.e., guarantee fairness) while achieving these performance goals.

The ability to reduce wait-time due to load imbalance and scheduling discrepancies requires both load balancing and scheduling support simultaneously.

Our overall system is composed of a runtime subsystem — Locality Aware Dynamic Load-balancing Environment (LADLE) — and Fair Cooperative Scheduling (FCS). FCS is our operating system scheduler that interacts with applications and allows them to have a say in how they use their allotted time. LADLE is responsible for initial distribution and subsequent runtime redistribution of load assignments among the processes of the parallel application.

**Fair Cooperative Scheduling (FCS)** addresses the issue of uncoordinated scheduling by letting applications have a say in the way they use their time as long as they do not exceed what they would have received if they were not using FCS. First, we observe that ideal coscheduling can not be achieved when the number of processes on each node is not exactly the same. Second, we observe that a process requires a peer to be scheduled only when it needs to communicate or synchronize with it. Third, for a process, having a peer scheduled at the time of a synchronization request does not guarantee immediate synchronization, since the peer is required to be at a certain point of its execution, for it to be able to respond.

All existing coscheduling techniques assume an equal number of processes on each node, or at least are willing to pay the price of unused scheduling slots [28, 64]. Dynamic coscheduling (DCS) [72] makes the second observation as well, and relaxes the requirement of simultaneous scheduling of all interacting processes by requiring that a process gets scheduled on demand only when one of its peers need to interact with it. However, the approach relies on homogeneous hardware, equal load on all nodes, and ignores the fact that a process may not be in a position to respond, even if it was scheduled on demand.

In our solution, we do not make any assumptions about the level of multiprogram-
ming of each node. Furthermore, based on our second observation above, we do not require that all cooperating processes be scheduled simultaneously, but instead, we provide a system where a process has some credit that can be used to schedule the process on demand. Based on our third observation, to each process, we assign work proportional to available resources with the goal of ensuring that all cooperating processes arrive at synchronization points simultaneously. Therefore, our ultimate goal is not only to provide coscheduling where all processes are scheduled simultaneously, but to provide all the necessary mechanisms where cooperating processes can interact for both load adjustments and wait reduction by guiding scheduling decisions.

A scheduler should not have any loophole in which some applications can exploit some advantage in terms of resource usage. Preserving expected system behavior from the user's point of view is also an important factor that should be considered. Another aspect of (FCS) is that it provides scheduling on demand while preserving the proportional share of resources. FCS gives special emphasis to fairness not only because of its importance to the expected system behavior but also since it is designed to work with our load balancing system which uses progress as the basis for load balancing. Since a progress-based load balancing scheme relies on correct determination of completed work in a certain amount of time, it is important for the underlying scheduler to consistently allocate the same proportion of processor time, even in short periods of time.

The Static Program Analysis component of our system analyzes access and data structure information of applications to provide information to the runtime system. Static program analysis automatically extracts the needed information and presents it to the runtime system in a concise fashion. Static program analysis is also responsible for generating the code in a way that allows the run-time system readjust load assignments. It also inserts the runtime hooks needed for dynamic monitoring and load balancing.

Locality Aware Dynamic Load-balancing Environment (LADLE) is the runtime system that interacts with FCS and uses progress information of each process to shuffle
load from one process to another for the purposes of balancing load.

As it was made clear in Chapter 2, locality of data is too important to ignore. Therefore, our design gives special consideration to ensure that data is not moved unless necessary when balancing load. For the purposes of load balancing, we ruled out using a centralized task queue based approach not only because it was too expensive to synchronize access to such a central data structure, but also because such an approach would ignore locality of data, and potentially create excessive traffic. It is a well known fact that even cache to cache communication within the same multiprocessor node is prohibitively expensive. Considering the fact that communication in a COW environment is much more expensive, such an approach would be unacceptable. Instead, we use a per-process task queue and periodically move tasks when balancing is needed based on the RelativePower metric. The information extracted by the static program analysis is used to move as little data as possible. With our design, in most cases, only the subset of processes that have a changed load assignment need to communicate, leaving the rest of the processes undisturbed.

LADLE rules out process migration as well, since such an approach requires correct proportional representation of resources of all nodes, which is not always possible. The scaling factor used between two nodes is very application dependent and finding the right number of processes for such proportional assignment is hardly ever possible. Furthermore, even when technical complications are solved and the hardware is homogeneous, unless the number of processes is an integral multiple of the number of nodes and all the processes are equally demanding, it is not possible to make such a scheme work effectively. Our system performs load balancing based on progress.

The fundamental idea of LADLE's load balancing mechanism is to move work from processes lagging behind their peers to those that finish their assignment faster. LADLE uses progress as the metric underlying load balancing decisions. Lack of progress for a process compared to its peers, independent of the underlying cause, indicates lack of a necessary resources for that process. Hence the load needs to be redistributed to
processes that have more of the needed resources and hence progressing at a faster pace. Load runtime redistribution is done by re-assigning the work of each of the processes of the parallel application. The ability to distribute and redistribute data requires knowledge of the shapes and sizes of data structures. The information provided by the static program analysis provides the required knowledge.

LADLE is also responsible for using the interface exported by FCS to influence the underlying scheduling decisions. For that LADLE uses the exported interface to interact with FCS, and employs policies as to when to yield the processor if proceeding is not possible.

The following chapters describe each of our subsystems in detail, and the way they interact with each other. Chapter 4 describes the fair cooperative scheduler and its interface. Chapter 5 describes the static analysis used to extract access pattern information. Chapter 6 describes the runtime system, LADLE, and how it uses the statically extracted information, dynamically adjusts the load among processes, and interacts with FCS to make better scheduling decisions. In the same chapter, special considerations given to locality management are also discussed.
4 FCS: Fair Cooperative Scheduling

Similar to coscheduling schemes, fair cooperative scheduling (FCS) is based on the observation that a communicating or synchronizing process would have to wait unless its partner is running and able to communicate or synchronize right away. Further, we observe that such communication and synchronization requests are services provided to another party, and therefore a process needs to be scheduled only for the duration required to provide these services. Therefore, FCS schedules a process for a fraction of a quantum when such scheduling occurs due to a demand by another process.

Most importantly, FCS emphasizes fairness and preservation of expected system behavior. Even though there is no unique definition of fairness [85], the fairness FCS offers is intended to maintain the proportional share and responsiveness of the underlying scheduler, as if FCS services were not being used. FCS provides a fair share of the processor time, not only in the long run, but also at a granularity of one quantum: 1) No process using FCS to cooperate will ever receive more time than it would have received with the unmodified scheduler in the long term. 2) No process using FCS to cooperate will ever hold the processor for more than two consecutive quanta in the presence of a competing process of the same base priority. The first of these two items guarantees fairness in the long run. The second guarantees fairness and responsiveness of all competing processes in the short term, which is particularly important for interactive applications; such applications are very likely to be running on an autonomous
workstation.

4.1 Overview of the Unmodified Scheduler

Our FCS implementation is on top of the priority-based scheduler of the Tru64 operating system, using Compaq's Alpha 21164 processor. Tru64 is a UNIX-like system built on top of a MACH [3] kernel. Under normal operation, the OS has two entry points. 1) A periodic tick: A programmable counter decreasing at every clock cycle traps into the OS once it reaches zero. 2) Regular system calls made by the executing applications. Expiration of the counter register is called a hardclock tick. Scheduling in Tru64 relies on these ticks, which have a frequency of 1200Hz. This roughly corresponds to a period of 833μs.

Under normal circumstances, every process is given one quantum of execution when it is scheduled. A quantum corresponds to 12 hardclock ticks, equaling 10ms. At each hardclock tick the leftover time of a quantum is decreased by one, a usage worth 833μs is charged to the process, and the following conditions are checked:

- If there is a higher priority process waiting to be run, even if the current process has not finished its assigned quantum, switch to the higher priority process.

- If the leftover ticks of the currently running process has reached zero, and there is an equal priority runnable process, preempt the currently running process, and schedule the other runnable one.

- If the currently running process was running during the earlier quantum as well, and we have another runnable process of equal priority, preempt the currently running process and schedule the other runnable one.

Priorities of processes are recalculated each time they are preempted and are based both on static priorities and on the proportion of execution times. In addition, priorities
are calculated at the turn of every second, so that those processes that are not scheduled for a long time can be given a boost due to aging of their resource usage. At the turn of every second, resource usage of any runnable process that has not been scheduled during the last second is multiplied roughly by $5/8$ for every second elapsed since it was last run. Internally, the Tru64 scheduler has 64 run-queues, with 32 of them — 32 through 63 — being for user processes. Run-queue 0 represents the highest priority, and run-queue 63 represents the lowest. Under normal circumstances, a process would move at most 1 run-queue higher (lower priority) every second. The run-queue of a process is calculated by having its CPU usage, divided by a constant, added to its base priority.

Since charging a process is made at a hardclock tick granularity, it is possible for some processes not to be charged for their usage, while others are charged for more than they actually use. If a process blocks for I/O right before a hardclock tick, and another process is scheduled instead, the blocking process will go uncharged for the fraction of the time it used, while the newly scheduled process is charged at the next hardclock tick for the full duration of 833\(\mu\)s.

### 4.2 FCS Design and Policy

In order to implement a responsive scheduler we devised a fair way of honoring scheduling demands by the OS. Our design of the scheduler targets fairness and differentiates between scheduling a process due to an external request and scheduling a process because it was at the head of the priority queue, without any external request. If a process is scheduled on demand, it is perceived as being scheduled to provide a service, and therefore it is scheduled only for enough time to provide that service. If on the other hand, the process is scheduled because it had the highest priority it is given a full quantum.

FCS is designed to ensure that under no circumstances would one process be fa-
vored over another. Being subjected to FCS policies and accounting is not automatic, and requires a process’ request. Once a process requests to be scheduled on demand, a repository containing an “emergency-fund” is maintained for that process. The repository is filled with one quantum of scheduling time, and the owner process is charged that much usage time. The repository is used only if that process receives a scheduling demand from its peers, so as to provide room for emergency scheduling requests.

The balance of the “emergency-fund” repository is replenished each time a process yields the processor voluntarily. Any leftover from the quantum of a yielding process is added to the balance of the repository so long as the balance in the repository does not exceed one quantum. Any amount of time added to the repository is charged to the process’ usage. Since there is no OS handled event to block on, yielding is achieved by depressing the priority of the caller process to be the lowest of all runnable processes. Before making such a priority change, all usage and priority values are stored, so that they can be restored whenever necessary.

The “Yield” system call is implemented as part of FCS. It takes one argument to indicate the willingness to yield. The argument specifies how low priority a process can be picked to yield the processor to, with 0 meaning the caller is willing to yield to a process of the same or higher priority. The return value, on the other hand, specifies the success of the call. A return of zero indicates no runnable process within the given priority interval was found to be scheduled. A non-zero value, however, indicates a process within that interval was found and scheduled.

Upon arrival of a scheduling demand, the intended process’ priority is restored to its original value, if it was depressed. If that was enough to put the process at the head of the run-queue and guarantee immediate scheduling nothing else is done — the process will be scheduled immediately. However, if the process was not depressed or restoring the depressed priority was not enough, and it has some funds in its “emergency-fund” repository, the process priority is boosted to the highest priority, and it is marked as scheduling-by-demand. This guarantees immediate scheduling of the process, though
for only a fraction of a quantum. When a process is put to run using funds from its “emergency-fund”, it is not charged for the time it is scheduled, since it was charged at the time these funds were added to the repository. If there are no funds in the repository, the demand is simply ignored.

Charging a process is done only at hardclock ticks under the unmodified scheduler, which favors frequently blocking processes, since they are never charged if they were to be blocked between two hardclock ticks. Since voluntary yields are likely to be more common under this design, frequently yielding processes might get favorable treatment due to such coarse grain accounting. In order to fix this problem, we use the hardware cycle counter as the basis of our accounting. Under this scheme, every process is charged the exact amount it used, if it were to yield voluntarily. Furthermore, the subsequently scheduled process is charged for only the fraction of the hardclock-tick that it used, and not for all of it. The overhead of using such fine-grain accounting is negligible.

4.3 Communicating Scheduling Demands

In order for a process to be scheduled on demand, the existence of such demand has to be communicated to the scheduler. Dynamic coscheduling [72] used network events to raise such demands. Upon receiving a message, the network driver checks whether the intended process is already scheduled. If so, the message is delivered as usual, and nothing else is done. However, if the recipient process is not scheduled, a scheduling event is raised and the recipient process’ priority is set to the highest in order to be scheduled immediately.

When there is some driver involvement or programmable hardware on the recipient end at the time of message delivery, it is possible for invocation of some scheduling policy that will raise events in order to guide demand based scheduling. However, not all networks provide driver involvement on the recipient end, or programmable hardware.
The network connecting the nodes of our COW, Compaq’s Memory Channel [34], uses user-level communication and does not have a notion of a message. Communication is done not by messages, but by remotely modifying another node’s memory, as if writing into local memory. Arrival of bytes at their destination does not raise any event. The network hardware at the recipient node does nothing but streaming of incoming bytes to their corresponding pinned (unpageable) memory. The semantics that the Memory Channel network provide is very similar to that of shared memory.

If we were to have either programmable hardware, or some driver involvement upon receiving a message, we would be able to use those mechanisms to raise the necessary events, and employ our FCS policy. Memory Channel allows processes to explicitly send signals over the network as a means of interprocess communication. In the absence of a better alternative, we use that signaling mechanism to communicate arrival of a message to a remote node. A process is free to choose any “catchable” signal to be registered as its “WAKEUP-SIGNAL”. Upon receiving that signal, FCS raises the priority of the process if it has a balance in its “emergency-fund”. Once the signal is acted upon, it is not delivered to the application, and simply discarded.

4.4 FCS Interface And Initialization

Usage of FCS policies is voluntary and a process must explicitly choose to be subjected to it. If a process decides to cooperate with its peers using FCS policies, it conveys its desire to the scheduler through a system call. Upon making the system call, FCS creates the aforementioned “emergency-fund” repository, and makes an initial deposit of one quantum into it.

The same system call provides a “WAKEUP-SIGNAL”, which upon reception will be interpreted as an incoming demand for the registering process to be scheduled. Since the signal is never delivered to the application, it is important for that signal to be permitted to be caught (e.g. SIGKILL can not be registered as the “WAKEUP-SIGNAL”).
<table>
<thead>
<tr>
<th>FCS Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean RegisterMemAndSig(void *p, int sig)</td>
<td>Tells the scheduler to register “Is_Signaled” and “Is_Running” flags at the given address “p”. Also, each time the signal “sig” is received, interpret it as some peer is requesting the caller process to be scheduled and schedule that process if policies permit. Returns TRUE if the given address (pointer + size of the flags) is in the address space of the calling process and the given signal can be used for that purpose.</td>
</tr>
<tr>
<td>Boolean Yield(int n)</td>
<td>Tells the scheduler that the caller process is willing to yield the processor to a process as many as “n” priorities below itself. If such a process is found it is scheduled immediately, and Yield(n) returns TRUE the next time the calling process is scheduled; otherwise, it returns FALSE immediately.</td>
</tr>
<tr>
<td>Boolean WakeupPeer(int peerNo)</td>
<td>Sends a “WAKEUP-SIGNAL” to the given peer, if neither of that peer’s “Is_Running” or “Is_Signaled” flags are set (i.e., the target process is not running and no other process has signaled it). Also, set the “Is_Signaled” flag if the process was signaled. Returns TRUE if a signal was actually sent.</td>
</tr>
</tbody>
</table>

Table 4.1: The exported FCS interface. FCS calls are safe since they are untrusted by default, and they only provide hints about an application’s expectations rather than dictating policies.

Even though FCS uses signals to convey demand for scheduling, the policies are completely independent, and the choice of signals is a mere convenience in the absence of any better alternative. However, signals in general are expensive to deal with, and cutting down the number of signals would be of great use. We use the network feature of remote writes to communicate the status of a process to its peers, so that a process is signaled only when necessary.

In order to reduce the number of signals, every registering process provides a pointer to two variables, which are shared with its peers: “Is_Running” and “Is_Signaled”. Both of these variables are generally mapped to the network for broadcast (if the cooperation is across nodes; otherwise, non network-mapped memory can be used to achieve FCS within an SMP node, for instance). Therefore, modification of these vari-
ables is propagated to all nodes that have mappings of the same addresses. The variable "Is_RUNNING" is always modified by the scheduler: It is set when the process is scheduled, and reset when it is preempted. The "Is_SIGNALED" variable of process "A" is set by its peers when any of its peers demands (i.e. sends it a signal) process "A" to be scheduled, and reset each time the process is scheduled or preempted. Prior to sending a signal, the values of these two variables are checked. There is no need to send a signal if "Is_RUNNING" is set (the process is already running), or if "Is_SIGNALED" is set (the process has already been signaled and this avoids signaling twice). Therefore, a signal is sent only if the process is not running, and no other process has already signaled it.

The use of the "Is_RUNNING" and "Is_SIGNALED" flags has a few known potential races. The overall operation includes four steps: 1) checking whether the "Is_RUNNING" flag is set, 2) checking whether the "Is_SIGNALED" flag is set, 3) sending the signal 4) setting the "Is_SIGNALED" flag. There is a potential race between each of these steps since either a read flag could change or the reading process get preempted. Luckily, none of these races pose any serious problem. The worst that could happen is to have a process spin a while before it yields the processor, or for the intended signal recipient to receive multiple signals. For instance, it is possible for multiple processes to simultaneously read the flags of the process they want to signal and both send "WAKEUP-SIGNALS" afterwards. This is harmless even though multiple signals are delivered to the intended process. When these signals arrive at their destination they will either all be ignored (the balance in the "emergency-funds" account is zero), or the first one received causes the scheduling event and the rest are simply ignored. Another case of a race is when the process changes state after its flags were checked by a peer that might potentially send a signal. Normally, the sender, after sending a message and before going into spin-waiting for a response, checks to see whether the "Is_RUNNING" flag of the recipient is set. If neither "Is_RUNNING" or "Is_SIGNALED" is set, the intended recipient is signaled. If the recipient is preempted after the flags are checked the sender process goes into the spin loop having the wrong information about its peer's running status. If
not addressed, the sender might spin unnecessarily waiting for the response. The issue is addressed by having the the sender process check not only for the arrival of the response while spin-waiting, but also for a status check of the intended recipient process. If the (“Is.Running” flag of the recipient is set to FALSE) before the expected response arrives, the sender yields the processor since it is not likely to get a response until the recipient peer is rescheduled. Another race case has to do with the “Is.Signalaled” flag and could potentially have slightly worse side effects. Normally if the recipient process is not running and is not already signaled, it is signaled and the “Is.Signalaled” flag is set to TRUE. If the sender process is preempted right after it sends the signal, but before setting the “Is.Signalaled” flag, the first thing it will do after it is rescheduled will be to set the flag. If the recipient process was actually scheduled and preempted in the mean time, such incorrect setting of the “Is.Signalaled” flag would prevent other processes from legitimately signaling that process, and potentially force them to yield the processor, after spinning for a period of time. Luckily, the chances of something like this happening are very slim, since the window between sending the signal and setting the variable is very small. Nevertheless, the negative effects of all the race cases are fixed after a process is scheduled or preempted since the scheduler clears the flags at each such event.

4.5 Emphasis on Fairness in FCS

FCS stresses fairness in its policies for several reasons. To begin with, it is desirable to preserve the proportional share of resources expected by users. After all, since the platform is a COW, it is natural to have interactive users, and interactive users expect the system to be responsive. FCS guarantees that no cooperating process will hold the processor for more than two consecutive quanta at any given time. Furthermore, it guarantees that the cooperating process will never exceed what would have been its allocated time under the unmodified scheduler. Therefore, FCS preserves the expected
system behavior for any unsuspecting user process, and presents a system view that is
the same as the system running an unmodified scheduler.

However, the importance of fairness is not limited to meeting user expectations.
Even though, as discussed in Chapter 2, progress based metric addresses all causes
of load imbalance, for such a metric to work, progress of the concerned application
should be measured correctly. A scheduler that does not observe fairness might favor
a process for a period of time thereby having it progressing faster, and perhaps later
starving it. Such variation would not only cause bad behavior for interactive programs,
but also would distort the measurement of progress, at least in the short term. In that
sense, fairness at all times is central to our approach of load balancing, by providing a
fundamental feature required to make a portable metric work.

4.6 Safety of FCS

FCS provides several ways of interaction for processes that ask to be subjected to its
policies. The first interaction is the “WAKEUP-SIGNAL”, coming from the application
to indicate a scheduling request. The other ways are through the “Is.Running”, and
“Is.Sigaled” variables, as well as the argument of the “Yield” system call.

Every “WAKEUP-SIGNAL” goes through all the safety checks any signal goes
through, and is discarded unless the sender is authorized to signal the destination pro-
cess. Even when such a signal is presumed allowable, it is merely a request for scheduling,
which could be discarded. Potentially, the values of “Is.Running” and “Is.Signedal”
can be changed by the application. However, neither of these values are ever read by
the scheduler; they are only written. The argument to the “Yield” system call is always
checked to see whether it is in the allowable range, and an error is returned otherwise.

Potentially, it is possible for the application to provide an invalid location for (or
deallocate memory of) “Is.Running”, and “Is.Signedal”. If that was the case, the
scheduler would attempt to modify an address that is not in the address space of the
registering process. This would not cause a problem, since each writing attempt to any process provided address is checked for validity before any writing is done, as it is the case with any system call. If such a check yields that the address is not valid, it is marked as being invalid, and no further attempt to write to this address is made. If that happens, the process would need to re-register for the use of FCS.

4.6.1 Effects of Having a Different Network

FCS uses the remote write capability of our network, Compaq's Memory Channel, in order to expose the scheduling status of registered processes to their peers. Even though this is a feature not found in many networks, the circumstances that make usage of this feature necessary are not likely to be found in many other networks either. In terms of policy, FCS does not rely on remote write, or the ability to send signals. It relies on voluntary yields and the ability to generate an event that can be interpreted as a scheduling request.

Many networks that provide user-level messaging (e.g., Myrinet [11]) trigger an event upon message arrival. Since no event is triggered upon message arrivals on our platform, signals were used to raise scheduling requests. For the purposes of FCS, if the network triggers any event when a message arrives, it is possible to use that event to communicate the needed scheduling request. In the absence of such an event, it is possible (though not very efficient) to have every incoming message followed by a signal, in order to request scheduling of the recipient. In either case, the scheduling request that FCS relies on can be communicated.

The use of the remote write capability of our network is merely an optimization intended to reduce the number of rather expensive scheduling-request signals. Our implementation, rather than indicating a dependence on such a capability, shows that it is possible to implement FCS on most of the current networks, or even in a hardware shared memory environment.
4.7 Summary and Discussion

The general design philosophy of FCS is to schedule the processes of the same application on demand, while maintaining the intended proportional share of resource usage. Unlike coscheduling schemes, the goal is not to schedule all the processes of an application simultaneously, since such a goal assumes equal levels of multiprogramming on all nodes. Instead, FCS allows a process and its peers to have a limited say on the timing of the use of resources. A processes might give up a processor in order to build up equity, for the purposes of potentially getting the processor in the future, to provide services to its peers. Providing these services immediately is important since peer processes might have to wait otherwise.

FCS provides an interface that allows processes to conditionally yield the processor — a process may choose what priority range it is willing to yield to. The provided call returns whether the yield attempt has been successful, and gives the process a tool to determine the amount of contention on the processor. These observations then can be used to adjust the frequency of yield attempts.

For the purposes of communicating scheduling demands, the signaling feature of our underlying network is used. For the purposes of reducing the number of delivered signals, however, the remote write capability of the network is used to communicate the scheduling status of the destination process.

FCS can be easily ported to other platforms with different interconnects or even to clusters with hardware-based coherent shared memory systems. In case of hardware shared memory systems, FCS could be implemented simply by using the shared memory to map "Is_Signaled" and "Is_Running" flags and using the same signaling mechanism as the current implementation. The use of signals in our current implementation is due to the fact that the interconnect on the platform we implemented FCS uses interrupt-free user level messaging, providing semantic very similar to that of shared memory. This forced us to use user-level signals to communicate scheduling requests.
In an environment with a network that raises interrupts upon message delivery, it would have been possible to use these interrupts as scheduling request and eliminate the use of user-level signals all together.
5 Static Program Analysis

Load balancing can be defined as the act of making the assigned work on all computational nodes proportional to the available resources on the nodes. Load balancing parallel jobs requires adjusting the amount of work each process is assigned to be proportional to the available resources, so that processes reach their synchronization points at the same time. The ability to change such an assignment requires knowledge of computation as well as the processed data structures and their sizes in order to balance load in a locality preserving manner. For performance reasons, it is also valuable to know the boundaries and types of sharing. While it is possible for the user to supply this information, automation is desirable in order to retain ease of use. In this chapter, we describe the static analysis required as well as the interface used to supply the necessary information to the run-time system.

Gathering statically available information has been shown to be a valuable complement to runtime systems in reducing the overall execution time of parallel applications, especially when applications have a regular structure that permits such analysis [23]. In their study, Dwarkadas et. al. have observed that pure run-time systems, despite their support for a wide class of applications, consume considerable amount of time detecting shared memory accesses and sending messages accordingly to maintain consistency. The same study has shown that exploiting statically available information could reduce the runtime overhead of software distributed shared memory consider-
ably, leading to performance similar to that achievable by applications relying only on compile-time analysis. SUIF-Adapt [42] combines compile-time program analysis with runtime information to guide global data distribution with the purpose of balancing load and reducing communication.

The extracted information is packed and presented to LADLE in a concise manner similar to Regular Section Descriptors (RSD) [40]. Our interface is designed to provide as much information as possible, with little time and space overhead. Information can be easily extracted and hand-coded into an already parallel program to be fed to LADLE. However, the same interface can be automatically inserted by a compiler, if the input program is sequential. The compiler not only analyzes the application and inserts the necessary LADLE hooks in order to provide the statically extracted information, but it also parallelizes the sequential application in a format needed by LADLE to permit load balancing. The compile-time analysis is carried out by a modified SUIF compiler [4, 38].

Our runtime system hooks are also inserted during this phase. The hooks are library calls to LADLE in order for LADLE to be able to observe the progress of the application and redistribute the load if necessary. LADLE hooks are also needed so that cooperation with the OS scheduler (FCS) can be achieved.

5.1 Identifying Data Structures of a Program

L00: int shArrayId;
L01:
L02: shArrayId = GetNewSharedArrayId();
L03: SetSharedArrayName(shArrayId, 'sh_arr');
L04: SetSharedArrayAddr(shArrayId, sh_arr);
L05: SetSharedArraySize(shArrayId, 0, N);
L06: SetSharedArraySize(shArrayId, 1, M);
L07: SetSharedArrayType(shArrayId, sizeof(int));

Figure 5.1: Inserted code for identifying the shared data structure sh_arr to the runtime system LADLE. Initially, we get a unique id for the array, and register its starting address and sizes in each dimension. As the type of the array, we are only interested in the size of each element.
For the purposes of correctness of the application, each of the shared data structures are identified and information about their types, number of dimensions, and size in each of the dimensions are extracted and presented to LADLE. Identifying and registering data structures have performance implications as well. Knowing the number of elements, their layouts, and sizes also allows the runtime system to make locality-aware decisions in partitioning work.

Even though potentially all data could be shared, we discriminate between process-private and shared data structures syntactically. In order to separate private data from shared data, shared data structures are identified by a `sh_` prefix in their names. Once we identify each of the shared data structures, code is inserted for each of the data structures to be identified by LADLE. Registration is done only once, right after allocation, since a data structure’s shape or size does not change after allocation. A typical registration of a data structure would be as shown in Figure 5.1. The data structure in this example is a two dimensional array of integers, with M rows and N elements in each row. As the type of the data, we take the size of each element.

```
L00: function ModifyMatrix(start,end)
L01: {
L02:   int i,j;
L03:   for (i = 0; i < M; i++){
L04:     for (j = start; j < end; j++){
L05:       sh_arr[i][j] = ...
L06:     }
L07:   }
L08: }
L09: }
```

Figure 5.2: A simple sequential loop, operating on array “sh_arr”. The loop assigns some values to every element of each row between column “start” and column “end” of the array. We assume this loop is parallelizable. Even though the outer-loop boundary always goes from 0 to M, and is known statically, the inner-loop boundaries are received as arguments to the function, and are known only during execution.

Aside from identifying all the data structures upon their allocation, information about accessed data structures in each of the parallel regions of the program is also extracted and fed to LADLE. For example, the loop in Figure 5.2 iterates over every element of dimension 1 of “sh_arr”, between elements “start” and “end” of dimension
L00: if (RegionNotInitialized(cur_region)) {
L01:     SetRegionInitialized(cur_region);
L02:     array_num = 0;
L03:     shArrayId = GetSharedArrayId("sh_arr");
L04:     AccessedDataInfo[cur_region][array_num].Id = shArrayId;
L05:     AccessedDataInfo[cur_region][array_num].access_type = WRITE;
L06:     AccessedDataInfo[cur_region][array_num].along_dim = 1;
L07:     dimension = 1;
L08:     AccessedDataInfo[cur_region][array_num].lower_bnd[dimension] = 0;
L10:     AccessedDataInfo[cur_region][array_num].a[dimension] = 1;
L11:     AccessedDataInfo[cur_region][array_num].b[dimension] = 0;
L12:     dimension = 0;
L13:     AccessedDataInfo[cur_region][array_num].a[dimension] = 1;
L14:     AccessedDataInfo[cur_region][array_num].b[dimension] = 0;
L15:     
L16:     SetTotalArrays(cur_region, 1);
L17:     SetSharingType(cur_region, INDEPENDENT);
L18:     SetLoadType(cur_region, FIXED);
L19:     }
L20:     array_num = 0;
L21:     AccessedDataInfo[cur_region][array_num].lower_bnd[0] = start;
L22:     AccessedDataInfo[cur_region][array_num].upper_bnd[0] = end;

Figure 5.3: The access information of the loop in Figure 5.2. "Sh_arr" is the only accessed shared data structure. This data structure is assumed to have already been registered as shown in Figure 5.1. The data structure is modified. Hence the access_type is set to WRITE. Dimension 1 of the array is always accessed from element 0 through element M with subscript "i" ("1 * i + 0 = i"). Dimension 0 of the data structure is always accessed between elements "start" and "end", which are determined during the execution. Therefore, these values are set at each instantiation of the loop, while other values are set only upon the first entry.

0. The array "sh_arr" was registered, as shown in Figure 5.1. "sh_arr" is modified, indicating the access type to this array is "WRITE". The way the accessed data structures of the loop are introduced to LADLE is shown in Figure 5.3.

In Figure 5.3, "array_num" is the number of arrays within a parallel region, starting from 0. "access_type" can be either "READ" to indicate read-only data, or "WRITE" to indicate write-only or read/write data. In this example, since "sh_arr" is modified, its "access_type" is set to "WRITE". "along_dim" specifies which dimension the parallelization is being made along. In this example, rows are to be distributed among processors, and therefore dimension 1 is the one that is parallelized. Lower and upper bounds are the lowest and highest elements accessed in each dimension. Further, in order to understand the access distance of elements, we specify the array subscript formula for each dimension, in terms of "a" and "b". "a" and "b" specifies the coef-
ficients of a polynomial, in the form of \( a \times i + b \), where \( i \) is the loop-counter, i.e. we assume that the array indices are affine functions of the loop indices.

One thing to notice in Figure 5.3 is that the lower and upper bounds of dimension 0 are set at each instantiation of the function, while all other values are set only upon the first execution. The reason for doing so is that while all other values remain constant, \textit{start} and \textit{end} are passed as arguments to the function, and therefore, are not known until the function is called. Because it is possible for the same function to be called multiple times, it is possible for these "\textit{start}" and "\textit{end}" values to be different for each call.

5.2 Static Hint Generation About Loop Structures and Data Accesses

While writing parallel applications, it is important to distribute the computational loops among processes in an efficient manner. As outlined in Section 2.1.1, most common distributions are \textit{block}, \textit{cyclic}, and \textit{block cyclic}. For the appropriate distribution to be determined, it is necessary for the decision maker to know the type of sharing between processes.

```plaintext
L00: for (ctr = 0; ctr < 100; ctr ++){
L01:    for (i = 1; i < M - 1; i++){
L02:        for (j = 1; j < N - 1; j++) {
L03:            sh_b[i][j] = (sh_a[i-1][j] + sh_a[i+1][j] +
L04:                sh_a[i][j-1] + sh_a[i][j+1])/ 4;
L05:        }
L06:    }
L07:    for (i = 1; i < M - 1; i++){
L08:        for (j = 1; j < N - 1; j++) {
L09:            sh_a[i][j] = sh_b[i][j];
L10:        }
L11:    }
L12: }
``` 

Figure 5.4: A simple nearest neighbor averaging loop. For each element of \textit{sh.a}, four neighbors of that element are averaged and the result is put in \textit{sh.b}, then \textit{sh.b} copied back to \textit{sh.a}. This operation is repeated for 100 times.

Take, for instance, the loop in Figure 5.4. This loop works on a two-dimensional
array of floating point numbers. The loop averages four neighboring elements of element (i,j) of \(sh.a\), and puts the result in element (i,j) of \(sh.b\). Once the averaging pass is complete, all elements of \(sh.b\) are copied back to \(sh.a\). This is repeated 100 times.

Such loops are common when solving partial differential equations.

```c
L00: // P is number of processes and "id"s are 0 through "P - 1"
L01: for (ctr = 0; ctr < 100; ctr++) {
L02:     for (i = my_pid + 1; i < M - 1; i += P){
L03:         for (j = 1; j < N - 1; j++) {
L04:             // We need to get row i-1 and i+1 from other processes,
L05:             // since they were modified during the last iteration
L06:             sh_b[i][j] = (sh_a[i-1][j] + sh_a[i+1][j]) +
L07:             sh_a[i][j-1] + sh_a[i][j+1])/ 4;
L08:         }
L09:     }
L10: } // Do not proceed until all others arrive this point
L11: barrier();
L12: for (i = my_pid + 1; i < M - 1; i += P){
L13:     for (j = 1; j < N - 1; j++) {
L14:         sh_a[i][j] = sh_b[i][j];
L15:     }
L16: } // Do not proceed until sh_a is updated by all processes
L17: barrier();
L18: }
```

Figure 5.5: Parallelization of the loop in Figure 5.4 using cyclic distribution. Each process iterates over rows \(i, i+P, i+2P\), ..., with \(i\) starting at \(pid+1\) of the respective process. Since each row is modified at each execution of the loop, rows \(i-1\) and \(i+1\) needs to be brought up-to-date before the loop can proceed. This would require communication to receive the two rows modified in the earlier iteration for each row a process works on.

Figure 5.5 shows parallelization of the loop in Figure 5.4 using cyclic distribution of data rows among processes. Process \(p\) writes rows \(p, p+P, p+2P\) ... of \(sh.b\), averaging the four corresponding values of \(sh.a\). This requires the loop to read rows \(i - 1\), \(i\), and \(i + 1\) of \(sh.a\), in order to calculate row \(i\) of \(sh.b\). Since rows \(i - 1\) and \(i + 1\) are modified by other processes, such distribution would incur extra costly communication, making cyclic distribution a poor choice for this type of sharing [86].

Figure 5.6, on the other hand, shows parallelization of the loop in Figure 5.4 using a block distribution of the data rows among processes. Each process in this case takes one large chunk with two boundaries only, and therefore exchanges at most two rows of data with other processes. Such distribution cuts the amount of communication considerably.
L00: // P is number of processes and "id"s are 0 through "P - 1"
L01: start = my_pid * N / P;
L02: end = (my_pid + 1) * N / P;
L03:
L04: // Take care of the boundary conditions
L05: if (my_pid == 0) start ++;
L06: if (my_pid == P - 1) end --;
L07:
L08: for (ctr = 0; ctr < 100; ctr ++){
L09:   // Fetch modified rows "start-1" and "end"
L10:   for (i = start; i < end; i++) {
L11:     for (j = 0; j < N - 1; j++) {
L12:       sh_b[i][j] = (sh_a[i-1][j] + sh_a[i+1][j] +
L13:                  sh_a[i][j-1] + sh_a[i][j+1])/ 4;
L14:     }
L15:   }
L16:   // Do not proceed until all others arrive at this point
L17:   barrier();
L18:   for (i = start; i < end; i++) {
L19:     for (j = 0; j < N - 1; j++) {
L20:       sh_a[i][j] = sh_b[i][j];
L21:     }
L22:   }
L23:   // Do not proceed until sh_a is updated by all processes
L24:   barrier();
L25: }

Figure 5.6: Parallelization of the loop in Figure 5.4, using block distribution. Each process iterates over rows start to end and therefore needs to exchange at most two rows with other processes.

In order to signal nearest neighbor type sharing in a loop, the loop is labeled “STENCIL” (identified by multiple accesses to the same data structure where each of the accesses have a constant displacement relative to the index the loop is parallelized by). If we know such sharing does not exist, the loop is labeled “INDEPENDENT”. In cases where it is not possible to tell whether such sharing exists, the loop is labeled as “UNKNOWN” and the decision about the parallelization strategy is decided by LADLE during execution.

The type of sharing is not the only parameter of interest when it comes to picking the right workload distribution for parallel applications. For the purposes of load balancing, it could be beneficial to distribute the loops cyclically, when the amount of work increases or decreases monotonically (the number of iterations across executions of the same parallel region, or the number of processed data elements between iterations of the parallel region changes monotonically). For instance, take the loop in Figure 5.7. It shows a case where at each execution of the loop (the loop that is parallelized), a row is
L00: void example_variable_work()
L01: {
L02:     for (i = 0; i < M; i++) {
L03:         for (j = i; j < M; j++) {
L04:             for (k = i; k < M; k++) {
L05:                 sh_a[j][k] = ...
L06:             }
L07:         }
L08:     }
L09: }

Figure 5.7: A simple loop operating on a two dimensional, MxM array. Since the loop at line L03 starts from “i” rather than a constant number, the number of rows operated on is decreased by 1 at each iteration of the outermost loop. Similarly, since the loop at line L04 also starts from “i”, the number of columns being processed decreased by 1 at each iteration of the outermost loop.

L00: // P is number of processes and "id"s are from 0 through "P - 1"
L01: start = my_pid * M / P;
L02: end = (my_pid + 1) * M / P;
L03: for (i = 0; i < M; i++) {
L04:     start = max(start, i);
L05:     for (j = start; j < end; j++) {
L06:         for (k = i; k < M; k++) {
L07:             sh_a[j][k] = ...
L08:         }
L09:     }
L10: }
L11: }

Figure 5.8: Parallelization of the loop in Figure 5.7 using block distribution. Each process starts working on all rows between “start” and “end”. However, once the index of the outer loop (“i”) reaches the value of “start”, the process loses one row at each iteration. Once the index reaches the value of “end”, the process would not have any data to work on. This will create load imbalance among processes by having some processes spin idly, while others have to work on higher order rows.

left out of the computation. This loop is very similar to the main loop of the Gaussian elimination method for solving linear equations. Since at each execution of the parallel loop one row and one column is left out of the computation, distributing work in blocks, as shown in Figure 5.8, is clearly not a viable option. The process assigned the first block (the lowest “start” and “end” values) will be left without work at early stages of the computation, while the process receiving the block with the highest “start” and “end” values will have to work continuously. One option would be to set the values of “start” and “end” at each instantiation of the parallel loop. However, that would require extra communication between processes.

Distributing rows in a cyclic fashion, as shown in Figure 5.9 will assign roughly the
L00: // P is the number of processes. Id's are from 0 through 'P - 1'
L01: start = my_pid;
L02: for (i = 0; i < M; i++){
L03:     if (start < i) start += P;
L04:     for (j = start; j < M; j += P){
L05:         for (k = i; k < M; k++){
L06:             sh_a[j][k] = ...;
L07:         }
L08:     }
L09: }

Figure 5.9: Parallelization of the loop in Figure 5.7 using a cyclic distribution. Each process starts working at row "start" and skips P rows at each iteration. Each process loses one row of its assignment every P iterations of the outermost loop, making all processes finish execution of the loop at almost the same time.

same amount of work to all the processes while minimizing communication. In this case each of the processes will lose one of their rows periodically and eventually all processes will finish at most doing one row more (or less) than their peers.

If the number of elements being accessed in a loop is known to monotonically increase or decrease across iterations or across invocations of a parallel region, the amount of work per loop is set to "VARIABLE". If, the number of accessed elements is known to be fixed across iterations and executions of a parallel region, the load is set to "FIXED". Finally, if the load can not be determined statically, it is marked as "UNKNOWN".

5.3 Parallel Code Generation

The identification of data structures and generation of hints about data accesses and loop structures, along with the parallelization of sequential code can be done automatically. 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) [4, 38] compiler. The SUIF system is organized as a set of compiler passes built on top of a kernel that defines the intermediate format. For the version of the compiler we use, each of these passes is implemented as a separate program that reads its input from a file and writes its output to another. The architecture of the com-
L00: void example_variable_work_exec()
L01: {
L02:     for (i = 0; i < M; i++) {
L03:         while ((task = LADLE_getNewTask()) != NULL) {
L04:             start = task->start;
L05:             end = task->end;
L06:             step = task->step;
L07:             for (j = start; j < end; j += step) {
L08:                 for (k = i; k < M; k++) {
L09:                     sh_a[j][k] = ...
L10:                 }
L11:             }
L12:         }
L13:     }
L14: }
L15: }

Figure 5.10: The LADLE formatted parallel version of the loop in Figure 5.7. The “while-loop” at line L03 is added for the loop to be executed in smaller tasks, making it possible to change assignments during execution. The entry point into LADLE is the call to LADLE_getNewTask() at line L03.

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.

Two passes were added to the SUIF system for our purposes. The first pass works on the sequential code after it is converted into SUIF readable form. This pass searches the abstract syntax tree (AST) for symbols representing shared data structures and adds LADLE hooks that define their shapes and sizes. As described in Section 5.1, only information about data structures whose names start with “sh_” is analyzed. SUIF provides a test of parallelism for each given block in the AST. If a block is found to be parallelizable, a chunk of code is prepended to that block setting the loop’s characteristics such as its lower and upper bounds, increment, accessed data structures and access boundaries, types of access for each of the data structures (read-only or read/write), the type of sharing with respect to each of the data structures (whether nearest neighbor or independent), and whether the amount of work in the loop remains fixed or varies from one iteration to another. The AST has all the information to complete the needed analysis. Our pass takes this information and retains it in a format presentable to LADLE.
Figure 5.11: The LADLE formatted data definition for the loop in Figure 5.7. Every parallelized region is given a unique identifier shown here by “cur_region”. The section between L04 and L25 is executed only once, at the first call of this function. The amount of data accessed by inner loops is variable across consecutive executions of the same parallel region and therefore it has to be set at each invocation. Therefore lines L26 and L27 set these bounds at each instantiation of the parallel region. The parallelized loop of the original function is shown in Figure 5.10 and called at line L29. At this point LADLE has all the information to divide the work, and execute the parallelized loop correctly.

at runtime. After these hooks are inserted, a standard SUIF pass is used to translate the resulting AST into a parallel C program.

The second pass converts the parallel regions of the newly generated parallel program into a format that allows changing load assignments dynamically. Each loop-block of the AST is subjected to the parallelism test, and if a loop-block is found to have been parallelized an extra outer “while-loop” is added to the block that makes a library call to set the boundaries of the currently processed loop-block. The resulting parallel program has a section that produces the access information, and a section that
actually does the parallel computing. An example translation of the loop in Figure 5.7 is shown in Figure 5.10 and Figure 5.11. The section shown in Figure 5.11 is the initialization section and it inserts information about the loop structure and accessed data. The code between lines L04 and L25 is executed only the first time the function is called. That initialization assumes the data structure sh.a has already been introduced to LADLE immediately after it was allocated, as shown in Figure 5.1.

The initialization section of the loop tells LADLE the parallel section’s sharing type is “INDEPENDENT”, and the amount of work is “VARIABLE” between two iterations. This will guide LADLE’s creation of tasks.

The section that does the actual parallel execution of the function is shown in Figure 5.10. The structure of the loop is to keep asking for tasks until there are no more tasks to work on. Since LADLE is responsible for forming and distributing these tasks, each task acquisition requires a call to LADLE.getNewTask(). The extra “while-loop” at line L03 of Figure 5.10 is intended to allow assignments that can dynamically change. A call to LADLE.getNewTask() might trigger a load balancing operation, which might result in bringing more work from other processes without the user application being aware of it.

5.4 Summary

In this chapter we have described the static program analysis we perform which automates the process of extracting access information available at compile time so that dynamic load balancing decisions can be made and run-time overheads are reduced. Static program analysis helps identify all accessed shared data structures of a program, along with its boundaries, access types, etc. It also identifies parallelizable loops of a program along with its access boundaries and indices. The extracted information is inserted in a format that is understandable by the runtime system. The runtime system
uses the extracted information for the purposes of load partitioning as well as repartitioning and reassignment.

Static analysis requires a well defined interface to pass the extracted information to the runtime system. Our developed interface is powerful enough to express access information for a large class of applications. Furthermore, the interface is well structured enough to be inserted automatically, with the help of a compiler, and simple enough to be hand coded if necessary. Along with instrumentation of the code, for many applications, transformation of the code from sequential to parallel can be done automatically.
6 LADLE: Locality Aware Dynamic Load Balancing Environment

Our runtime system, Locality Aware Dynamic Load balancing Environment (LADLE), is responsible for handling initial load assignment as well as balancing load as needed. LADLE takes locality of data into account and uses the interface provided by FCS to reduce the communication and synchronization delay due to scheduling discrepancies. Locality-aware load assignment and balancing is accomplished through the identification of the data associated with the computation.

6.1 Initial Task Creation and Assignment

At the first execution of a parallel region, two decisions have to be made by the runtime system: 1) How to create tasks 2) How much to assign to each process. The answer to the first question is guided by whether the sharing type is “STENCIL” or “INDEPENDENT”, and whether the load is “FIXED” or “VARIABLE”. The answer to the second question is guided by our progress based metric, RelativePower.

RelativePower is represented by a single floating point number and represents the ability of a process to complete a task relative to its peers. If a process is capable of finishing twice as much work as one of its peers within a given amount of time, its RelativePower will be twice as much as that for each of its peers.
We will describe the estimation of `RelativePower` in Section 6.4. The sum of `RelativePowers` of all the processes of a parallel application is always 1, which makes the `RelativePower` of a process representative of the percentage of the work that should be assigned to it.

The simplest case is when a parallel region is labeled as having a sharing type of "STENCIL". As explained in Section 5, "STENCIL" signals nearest neighbor sharing between consecutive loops. In such cases, each boundary in an assignment indicates a need to communicate with the process that has the neighboring boundary row. Therefore, it is important to keep the number of boundaries as low as possible to cut the steady-state communication during execution. With these facts in mind, LADLE is designed to assign a single task to each process, with each of the tasks sized proportionally to the `RelativePower` of that process. This guarantees that each process will have at most two boundaries. Load balancing is achieved by resizing the tasks through shifting the bounds.

When a parallel region's sharing type is labeled as "INDEPENDENT", more tasks than the number of processes are created. Tasks are created in such a way that each of the tasks has an equal amount of accessed data elements, and tasks are fixed throughout the execution. The shape of these tasks depends on whether the load is determined to be "FIXED" or "VARIABLE". When the load is "FIXED", tasks are created as blocks of iterations, packing N consecutive iterations as a single chunk, where N is the size of the task. When the load is "VARIABLE", however, tasks are created in a cyclic or block-cyclic manner to capture the fact that the load across iterations is not fixed.

While creating a large number of tasks allows finer grain load balancing, it might have higher cost of maintenance at the time of balancing. The driving principle behind our initial task creation strategy is to create a reasonable number of tasks to allow load balancing, while keeping the overhead small. We use the size and alignment of the data structure being accessed in a parallel region, as well as the number of processes, as our guides while creating initial task assignments.
The initial attempt is to create as many tasks as twice the number of processes, for each process. In other words, if the number of processes is \( P \), the attempt is to create \( 2 \times P^2 \) tasks. This is also set to be the highest allowed number of tasks to be created for each parallel region. The reason for picking such a number is to allow the tasks to be distributed equally among other processes, if one of the processes happened to have only half the resources other processes have. However, that number might have to be reduced, if such distribution is found to cause extra communication due to false sharing — the concurrent, but otherwise independent, access by two or more processes to a single coherence unit. LADLE has a notion of coherence unit size and attempts to adjust the tasks to start and end at coherence unit boundaries. The initial attempt is to create the intended maximum number of tasks. However, if that is found to cause false sharing, a larger amount of work is packed together, if doing so makes it possible to find a task size without causing false sharing. Increasing the task size reduces the number of created tasks. This is allowed as long as the number of tasks per process is no less than 8. That number is obtained empirically, as a compromise between reducing the effects of false sharing and still leaving room for load balancing.

Finding the right size for a task is done differently for block tasks and cyclic tasks. In the case of block tasks the least common multiple of the size of the coherence unit and the size of data accessed per parallel loop iteration is found, and the corresponding number of loops are packed to form a single task. However, in the case of cyclic tasks, loop iterations "T" iterations apart are packed together, where "T" is the total number of created tasks. In this case false sharing happens at each element boundary, and increasing the number of elements per task would not help. Therefore, we resort to creating block-cyclic tasks, where multiple consecutive data elements are packed together to form a block without false sharing, if possible. The minimum number of 8 tasks is applicable regardless of what type of tasks are created.
6.2 Task-Queues and Task Management in LADLE

LADLE maintains per-process task queues, where tasks of each parallel region are kept. With the exception of the case where there is nearest neighbor sharing, there are always more tasks than the number of processes and such tasks are allowed to float among processes for the purposes of load balancing. The task queue abstraction allows data to be partitioned into smaller pieces, which can be moved around for the purposes of load balancing. Such an abstraction allows movement of tasks to occur only between processes relinquishing and acquiring work. In the case of parallel regions with nearest neighbor sharing, a task queue abstraction is still used, except that instead of having multiple tasks per process and letting tasks float for the purposes of load balancing, there is only one task per process, which is resized at the time of balancing load.

Several fundamentally different design choices were made in order to make a task queue approach viable in a COW environment. To begin with, rather than having a centralized task queue accessed by all the processes to grab a task, LADLE maintains a separate per-process task queue for each parallel region. Since access to a shared task queue has to be within a critical region protected by a lock, existence of per-process task queues reduces contention for such locks dramatically. Furthermore, since each process has its own set of tasks, it will work on the same set of data, as long as its assignment does not change. Even in cases where assignments change, if the sharing type is “INDEPENDENT”, a process with decreasing RelativePower would lose some of its tasks, however all the tasks it will be left with will be those that were owned before. Processes with increasing RelativePower, on the other hand, keep all their formerly assigned tasks and simply add more tasks to their assignments. This guarantees minimal task reassignment (and thereby data movement) for each new RelativePower computation.
6.3 Load Re-assignment Strategy

As Figure 5.11 shows, tasks are picked up one at a time and worked on until completed. This makes it possible for load assignments to be changed dynamically during execution.

Processes periodically exchange their execution progress information and have their new RelativePowers computed. If the newly computed RelativePowers differ from the earlier value by more than a predetermined threshold\(^1\), load is redistributed among processes. The information exchanged is fairly small, and includes only the "id" of each parallel region, the amount of work completed in that parallel region, and the time it took to do so. The work is stated either in terms of completed tasks, for regions marked as "INDEPENDENT", or in terms of iterations, for regions marked as "STENCIL".

When the sharing type is known to be "STENCIL", load is balanced among processes by increasing or decreasing the task size assigned to each process. Such a change in boundaries is intended to keep the number of boundaries limited to two. However, the drawback of such a load balancing strategy is that even processes whose RelativePower remain the same might incur some extra communication due to the shift in their assignment's starting and ending positions. The example in Figure 6.1 shows a 4-process case that starts with RelativePowers of (0.25; 0.25; 0.25; 0.25). The newly calculated RelativePowers for P0 to P3 respectively are (0.40; 0.25; 0.25; 0.10), which shows a decrease in the RelativePower of P3 and an increase in that of P0. Instead of moving some work from P3 to P0, boundaries of all processes are shifted causing extra communication for processes P1 and P2. However, when the existence of nearest neighbor sharing is taken into account, and the likelihood of iterating

\(^1\)The predetermined threshold is set so that at least one of the processes' RelativePower differ by more than 10% of its former value. The 10% is determined empirically, and was motivated by the need to strike a balance between avoiding overhead due to small RelativePower changes and not to ignore actual load imbalances.
Figure 6.1: An example task reassignment in a “STENCIL” parallel region. Initially all processes are of equal RelativePowers with (0.25; 0.25; 0.25; 0.25) as shown in (a). When RelativePowers change to (0.40; 0.25; 0.25; 0.10), indicating P0 is progressing faster, and P3 is progressing slower, the size of the chunk assigned to P0 is increased and the size of the one assigned to P3 is decreased, as shown in (b). Even though the sizes of chunks assigned to P1 and P2 do not change, their assignments change and they incur some communication overhead. However, considering the same data is likely to be iterated over multiple times, incurring one time overhead in the interest of reducing overall communication overhead due to steady state nearest neighbor communication could be well worth it.

over the same parallel region more than once is considered, incurring one-time unnecessary communication is worthwhile in the interest of avoiding future high steady-state communication.

When the parallel region’s sharing type is tagged as “INDEPENDENT”, communication among processes is kept at the minimum. When a new RelativePower is computed, only processes whose RelativePowers change incur communication. Tasks are transferred from processes with decreased RelativePower to processes with increased RelativePower. Figure 6.2 shows an example of such distribution. Initially all processes have equal RelativePowers. When RelativePowers change to (0.40; 0.25; 0.25; 0.10), three tasks of P3 are transferred to P0, without disturbing P1 and P2.
Figure 6.2: An example reassignment in a region marked as having an "INDEPENDENT" sharing type and "FIXED" load. The distributed data structure is a two-dimensional array, and two consecutive rows of this array are grouped together to form a task. Initial RelativePowers are (0.25; 0.25; 0.25; 0.25) for P0-P3 respectively, and each process is assigned 5 tasks (10 rows). When RelativePowers change to be (0.40; 0.25; 0.25; 0.10), 3 tasks of P3 are assigned to P0. Processes P1 and P2 are not affected by this move since their RelativePowers did not change and the sharing type is marked as "INDEPENDENT". Such reassignment would cause communication only between processes whose RelativePowers change.

6.4 Relative Processing Power

The fundamental metric of our load balancing system is progress. The intuition is to assign more work to processes that are capable of finishing assignments faster than their peers, until all the processes are able to finish their work, ideally, at the same time. While determining that ability, we use earlier performance as an indication of future performance, through a metric called RelativePower.

The ability of a process to finish twice as much work as its peer is represented by a RelativePower that is twice as large as its peer’s. RelativePower is independent of the cause of lack of resources to a process. It equally represents lack of processing power, memory, or any type of hardware resource, or existence of multiprogramming. The universally applicable nature of this metric makes it particularly attractive to us, since we target load imbalances without regard to their cause.

At the highest level RelativePower of a process is equal to the amount of com-
pleted work in a given unit of time.

L00: float RelativePower[NumOfPros];
L01: // Initialized at program start
L02: // to 1/NumOfPros
L03: float TaskTime[NumOfPros];
L04: // Execution time of parallel region
L05: float SumOfPowers=0;
L06:
L07: // Calculate new RelativePower
L08: for all Processors i
L09: RelativePower[i] /= TaskTime[i];
L10: SumOfPowers += RelativePower[i];
L11: endfor
L12:
L13: // Normalize RelativePowers to ensure
L14: // that sum of powers is 1.0
L15: for all Processors i
L16: RelativePower[i] /= SumOfPowers;
L17: endfor

Figure 6.3: A pseudocode algorithm to determine relative processing power for applications that use barriers for synchronization. Since work assignment to each process is proportional to its previously known relative processing power, and arrival at a barrier signals completion of that assignment, new relative processing powers could simply be calculated by dividing the earlier relative processing power of each process by the time it took for that process to finish its assignment. TaskTime has the time it took each of the processes to finish their earlier assigned work. RelativePower values are normalized to ensure their sum is 1.

Our initial implementation of LADLE [67], LADLE-BAR, was intended to support applications that use barriers for synchronization. That implementation computed new RelativePowers and balanced load only at barriers. This approach simplified load balancing considerably, since arrival at a barrier indicates completion of all assigned work. If a process is known to have been assigned a certain percentage of the work (the RelativePower of a process indicates the percentage of the work a process should be assigned), and the time it took to complete that work, it is fairly straightforward to compute the new set of RelativePowers for all the processes. Figure 6.3 shows a pseudocode algorithm to compute the RelativePower upon arrival to a barrier.

Waiting for a barrier for the purposes of load balancing has limitations in several ways. To begin with, not every application uses barriers to synchronize. Furthermore, even for applications that use barriers to synchronize, the time between two barriers could be so long that, running unbalanced for such a period of time could be a waste
of resources in itself. To solve this problem LADLE-IND was designed and implemented [68]. LADLE-IND provides load balancing ability without regard to the type of synchronization used, and works even if no synchronization is used at all.

The *Relative Power* was defined as the amount of completed work per unit time. However, if processes are not at a global synchronization point at the time of an attempt to balance load, determining the amount of completed work could be much harder. In the presence of barriers (global synchronization point), where load reassignment is done only at such points, processes depart barriers with a predetermined assignment, and do not arrive at another barrier until they are done with their assignment. Therefore, at the time of arrival at the barrier, both the amount of completed work, and the time it took to do so is known. However, in the absence of barriers, it is possible for a process to span multiple parallel regions, completing a different number of tasks in each of these regions. In this case simply taking the total number of completed tasks as the completed work may not be suitable, since it is very likely that a task in one region would have a different amount of work than a task in another region. It is even possible for some processes to complete more tasks in one region than one of its peers, while completing less tasks in another parallel region. In order to design a universally applicable metric, a way to compute the completed work is needed.

For the purposes of determining the amount of completed work, we associate a work value with each task of a parallel region. Once that is determined it is multiplied with the number of tasks completed in that parallel region, and the result is added to the completed work. The pseudocode of the algorithm to determine the *Relative Powers* of processes independent of the used synchronization method is shown in Figure 6.4.

As an illustration, assume a case where two processes, process A and process B, are performing computations spanning two parallel regions, as shown in Table 6.1. In this example, process A has completed 5 and 7 tasks, and process B has completed 8 and 5 tasks from region 1 and region 2 respectively. The average time spent by each process to complete one task is computed and the minimum of these two averages is taken to be the
L00: float RelativePower[NUMOfProcs] // Relative processing powers
L02: float NumTasks[NUMParRegs][NUMOfProcs] // Executed tasks per region
L03: float WorkPerProc[NUMOfProcs]
L04: float PerProcExecTime[NUMOfProcs], Power, SumOfPowers = 0
L05:
L06: // Find the amount of work done by each process and the time it took
L07: for all Parallel Regions i
L08:    float AvgTime, WorkDone, MinTaskTime = LARGENUMBER
L09:
L10:    for all Processes j
L12:    AvgTime = TaskTime[i][j]/NumTasks[i][j]
L13:    if (AvgTime < MinTaskTime) MinTaskTime = AvgTime
L14:    for all Processes j
L15:       WorkDone = NumTasks[i][j] * MinTaskTime
L16:    WorkPerProc[j] += WorkDone
L17:
L18:    // Calculate RelativePower
L19:    for all Processes i
L20:       Power = WorkPerProc[i]/PerProcExecTime[i]
L21:    RelativePower[i] = Power
L22:    SumOfPowers += Power
L23:
L24:    // Normalize The RelativePower
L25:    for all Processes i
L26:       RelativePowers[i] /= SumOfPowers

Figure 6.4: Computing "RelativePowers" without relying on any type of synchronization being used. The challenge is to determine the amount of completed work by each of the processes. The section between lines L07 and L16 does just that. The minimum average time spent in a parallel region working on one task is assumed to be the time tasks of that region can be completed at. Once that value is found, it is multiplied by the number of completed tasks by each of the processes. The same set of operations is repeated for all of the parallel regions, finding the amount of work completed by all the processes. Since it is known how much time is spent by each of the processes, computing RelativePowers is straightforward. Once RelativePowers are computed, they are normalized to ensure their sum is 1.

... work-value of a task in a region. Once this is determined, the total load on each process is calculated and divided by the total execution time to complete that work to obtain raw RelativePowers. These values are normalized to find RelativePowers to be used as the basis for load balancing.

In both LADLE-BAR and LADLE-IND, calculating RelativePowers, and redistributing load is done in a distributed fashion, without interacting with other processes. Interaction is needed only to exchange execution statistics such as execution time, and in the case of LADLE-IND, the number of tasks executed in each of the parallel regions. Similarly, after new RelativePowers are calculated, each process decides the new load assignment locally without exchange of any information with other
<table>
<thead>
<tr>
<th>Procs</th>
<th>Num of Tasks</th>
<th>Exec Time</th>
<th>Avg Time Per Task</th>
<th>Min Avg Per Task</th>
<th>Load</th>
<th>Total Load</th>
<th>Rel. Power</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Reg1: 5</td>
<td>3s</td>
<td>1s</td>
<td>1</td>
<td>5</td>
<td>10.6</td>
<td>10.6/11.3</td>
<td>0.938</td>
</tr>
<tr>
<td></td>
<td>Reg2: 7</td>
<td>6.3s</td>
<td>0.9s</td>
<td>0.8</td>
<td>5.6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Reg1: 8</td>
<td>10s</td>
<td>1.25s</td>
<td>1</td>
<td>8</td>
<td>12</td>
<td>12/14</td>
<td>0.857</td>
</tr>
<tr>
<td></td>
<td>Reg2: 5</td>
<td>4s</td>
<td>0.8s</td>
<td>0.8</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: An illustration of calculating RelativePower in LADLE-IND. The parallel application has two processes and each of the processes span two parallel regions — Region 1 and Region 2. “Min Avg Per Task” chooses the minimum average time spent by each of the processes working on a task. This is used as the actual work content of a task, and the number of completed tasks by each process is multiplied by this value to compute the amount of completed work in each parallel region. The work of all parallel regions is summed to find the “Total Work”, which is then divided by the total “Exec Time” to find raw RelativePowers. The raw RelativePowers are normalized to obtain RelativePowers.

Processes. Reassignment of tasks is nothing but moving task identifier pointers from one processor’s task queue to another. Since all the processes use the same algorithm and the same data input, they all reach the same outcome.

### 6.5 Execution of Newly Moved Tasks

Once new RelativePowers are computed and a new load assignment is made, tasks need to be moved from their former hosts to their newly assigned hosts. Since our implementation of LADLE is running on top of a software distributed shared memory system, CASHMERE-2L [77], the data associated with each task is moved implicitly as needed during execution. Therefore, LADLE sidesteps the need to accurately identify the data to be communicated, thereby relaxing requirements from the compiler or user. The issue of dealing with the consistency model of the underlying shared memory system is handled by LADLE.

The underlying shared memory system, CASHMERE-2L, implements a multiple writer, release consistent [31] protocol\(^2\). Under release consistency, memory accesses are classified as either ordinary memory accesses, or acquire or release. Even though

\(^2\)CASHMERE-2L implements a “moderately” lazy release consistency protocol, which lies somewhere between the eager implementation of that protocol as it is implemented by Munin [9] and its lazy implementation as implemented by TreadMarks [5].
ordinary memory accesses are not ordered, *acquires* and *releases* are strictly ordered. A *release* indicates completion of an operation which other processors may depend on, and therefore, all writes made by the processor performing that *release* should be visible by all processors performing a subsequent *acquire*. Therefore, in order for the correct data to be transferred by the underlying shared memory system, any process giving up a task has to perform a *release* operation to make its changes to that data visible. That *release* has to be followed by an *acquire* operation performed by the process receiving the task.

```c
L00:  int getNextPivot(double *pivot)
L01:  {
L02:   static int curPivot = 0;
L03:   int retValue;
L04:   if (curPivot > lastPivot) return -1; // We are done
L05:   if (whoCalculatesCurrentPivot(curPivot) == my_id){
L06:     calculatePivot(curPivot);
L07:     setFlag(curPivot);
L08:   }
L09:   waitForFlag(curPivot);
L10:   retValue = curPivot;
L11:   copyPivot(pivot, curPivot);
L12:   curPivot ++;
L13:   return 1;
L14: }
L15: void mainLoop()
L16:  {
L17:    int curPivot;
L18:    double pivot[PIVOTSIZE];
L19:    while((curPivot = getNextPivot(pivot)) > 0){
L20:      applyThisPivotToMyTasks(pivot);
L21:    }
```

Figure 6.5: A simple program section that shows an application getting a pivot row, and applying that pivot to a bunch of tasks. Each pivot is computed by one of the processes, and its availability is signaled to others by setting a flag. Once this flag is set, all other processes can read and apply this pivot to their tasks.

In some cases, reassignment of a task does not necessarily mean the task can immediately be moved to its new host. Consider the program section shown in Figure 6.5. In this program a pivot is calculated by one of the processes, and its availability is signaled to others. Once it becomes available, every process (including the one that computed the pivot) applies this pivot to their own tasks. In this case, it is possible to have one
process far behind the others, since processes do not have to wait for anything other than for the pivot to be computed. In a case where some tasks are reassigned from a process lagging far behind to a process running several steps ahead, moving the tasks immediately would cause incorrect results, since some pivot applications are likely to be skipped. In order to deal with such cases, either the process running ahead has to preserve its past state in order to apply it to the newly arriving tasks, or has to wait for the former owner of the tasks to bring the moving tasks to the stage of computation it is in. Even though the former seems to be more advantageous in terms of performance (since the extra work is done by the potentially faster process), it could be prohibitively expensive to preserve state throughout execution in anticipation of getting tasks from earlier stages of the computation. Therefore, LADLE guarantees no task moves to its reassigned location unless the new location can work on it.

In case of LADLE-BAR, this does not need any effort at all. All load balancing operations are carried out within a barrier operation. In a barrier, all processes are at the same stage of their computation, and therefore any assignment can immediately take effect. Furthermore, all processes perform a release operation at the entrance of a barrier, and an acquire operation before departing it. Hence all modifications to all tasks is properly communicated, and any data access in a newly reassigned task will bring up-to-date data.

The case of LADLE-IND, on the other hand, is more complicated. Since it is possible to balance load at any point in the application, it is LADLE’s responsibility to ensure correctness of any data movement. In the case where some tasks are being moved from a process lagging behind to a process that is ahead (or vice-versa, even though less likely), the process lagging behind has to work on its earlier assignment until it brings its tasks to a state that can be moved to the new process. However, once it reaches that state it needs to stop working on any task that is to be moved. The new host process on the other hand has to wait for these tasks to be released before it can start accessing and executing any of them. For the purposes of achieving these goals, LADLE-IND
exchanges more information than just execution timing statistics.

LADLE-IND associates a time-stamp with each task, which is incremented by 1 each time the task is pulled out of the task queue for the purposes of execution. At each entry to a parallel region, LADLE-IND knows the time-stamp of the tasks to be executed, and only those tasks that have the expected time-stamp are executed. If the task pulled out of the task queue has a smaller time-stamp, it is an indication that it is a task moving from a process lagging behind, and it is not executed until that time-stamp is incremented to the expected value. If the task has a higher than expected time-stamp, however, it indicates a movement from a process that is further ahead, i.e., that has already executed this iteration of the task (less likely, but possible), and it is simply skipped without being executed. These time-stamps are the basis for correctness in LADLE-IND.

After LADLE-IND calculates new RelativePowers and finds out that load balancing is necessary, each process forms a message with the time-stamps of all the parallel regions it has executed. This indicates the lowest time-stamp of tasks for each parallel region, that can be accepted from other processes, should such movement be necessary. When a task is taken from a certain parallel region of process A and re-assigned to process B, processes A and B exchange their time-stamps for that region. The reassigned task is handed over only when its time-stamp reaches (or already has reached or exceeded) the time stamp of process B, for that particular parallel region. If, however, process A has a higher time-stamp already, it drops any tasks moving out and continues working on the remaining set of tasks. Process B, on the other hand, does not work on these tasks until its local time stamp for that parallel region is equal to the time stamp of these tasks.

Another issue LADLE-IND has to deal with is the coexistence of multiple assignments. When a slow process’ tasks are reassigned to a faster process with higher time-stamps, the slow process has to work on its old set of tasks until it brings their time-stamps to a state acceptable by the faster process. However, the new assignment of the
slow process does not even have the reassigned tasks listed, and hence starting to work immediately on the new assignment would leave all the reassigned tasks in an unmovable state, due to their time-stamps. Therefore, the slow process has to work on its old assignment, until all its reassigned tasks are movable, and then switch to its new assignment. In order to achieve that, LADLE-IND maintains multiple task queues. Newly created assignments are not used as long as any reassigned task in the older assignment needs to be further worked on.

6.6 Task Stealing for Intra-Node Load Balancing

LADLE balances load across nodes only after recalculating RelativePowers and finding they differ by the predetermined threshold. However, communication between processors of a symmetric multiprocessor (SMP) node is much cheaper compared to the cost of communication across the interconnecting network. Therefore, LADLE adopts a different policy with regard to load balancing within an SMP node: processes are allowed to steal tasks of other processes on the same node. Task stealing requires accessing the task queue within a critical section to avoid data races. Furthermore, since centralized task queues are known to cause bad cache behavior because subsequent executions of the same task might be carried out on a different processor, LADLE maintains the per-process task queue semantic, similar to the approach of affinity scheduling [58].

LADLE guarantees a process would always finish the work in its own task queue before it steals tasks from another process. Furthermore, a process pulls tasks out of its own task queue from one end, and from another process’ task queue from the other. Once a task is stolen, it is appended to the stealing process’ task queue for any possible future access. This guarantees that, as long as the progress rate is the same, stolen tasks will remain in their newly adopted task queues. Another benefit of per-process task queues is the reduction in the likelihood of contention for the lock of a task queue.
Normally, acquiring a lock in the absence of any contention is in the order of a few nanoseconds, which is negligible for all practical purposes, considering the fact that it is an infrequent operation. Contention for a lock is likely only if there is an intra-node load imbalance and multiple processes simultaneously attempted to access the same task queue.

6.7 Deciding to Balance Load

Balancing load should be frequent enough that a cause of load imbalance could be caught early on. However, sufficient time between two balancing operations needs to pass so that every process would have some work-report that is representative of its progress in order to avoid wrong measurements. Also, regardless of how lightweight it is, every load balancing operation incurs a slight overhead while calculating the new RelativePowers and assignments of all the processes.

The minimum time between two load balancing checks is set to 4 seconds. The choice of 4 seconds empirically meets representative-progress requirements, and is still frequent enough to capture even short term load imbalances. The frequency of load balancing is also adjusted during the execution. Each time a load balancing attempt yields RelativePowers that do not differ enough to warrant a load reassignment, the period is incremented by 4 seconds, up to 16 seconds. Conversely, each time a load balancing attempt produces different RelativePowers, the period of load balancing is reduced by 4 seconds, with a minimum of 4 seconds. This adjustment of load balancing period intends to reduce the number of load balancing attempts in fairly stable environments, while being more aggressive in highly dynamic environments.

In the case of LADLE-BAR, it is known that all processes exchange their execution statistics at each barrier. If any of the processes seemed to exceed the minimum time required to balance load, new RelativePowers are computed by each of the processes independently. If RelativePowers were found to differ from earlier val-
ues, load runtime redistribution is done, the period of load balancing is readjusted, and
computation is resumed. If they were found to be the same as before, however, only the
period of load balancing is adjusted and computation resumes.

In the case of LADLE-IND, however, there is not necessarily a barrier where all
the processes could agree to recalculate their RelativePowers. This calls for a
distributed scheme where any of the processes could request such action. Each process executes normally, as long as it can do so. When a process reaches the end of its own
task-queue, the first attempt is to steal tasks from processes running on the same SMP-
node, if any. If nothing was found there either, and sufficient time has elapsed since
the last attempt to balance, the process broadcasts a load balancing request to its peers,
by sending its own execution statistics. Once the message is received, all processes exchange their statistics, and new RelativePowers are calculated. It is possible to
have multiple processes send a balancing request simultaneously. However, this is not
an issue, since there isn’t any assumed hierarchy, and all decision making is distributed.

6.8 Locality Management in LADLE

LADLE gives special consideration to locality of data during reassignment. It guar-
antees that all individual tasks (and correspondingly the data they access) will be as-
signed to the same process (hence remain on the same processor) as long as the load is
balanced. Furthermore, if the sharing type is “INDEPENDENT”, even when the load is
unbalanced and needs to be balanced, the absolute minimum number of tasks moves
for any given new assignment. Processes with increased RelativePowers receive
additional tasks while retaining the tasks they had before. For processes with decreas-
ing RelativePowers, the tasks in the new assignment is a subset of the earlier one,
thereby retaining locality.

When there is nearest neighbor sharing (i.e., sharing type is “STENCIL”), how-
ever, new assignments might change the distribution of load to processes, even if their
RelativePowers remained the same. Even though this seems to be disregarding locality of data, not doing so would create many boundaries and therefore increase the total amount of communication, hurting the overall performance. This is the only case where locality of data is not taken into account in the interest of future reduction of communication.

LADLE's attempt to preserve data locality and reduce communication is not limited to reassignment of tasks due to load balancing. LADLE monitors the amount of accessed data and the time it takes for that data to be processed in each parallel region. In some cases, the accessed data is small relative to the coherence unit, located on a single node, and the amount of work is also small. In such cases, distributing the data among many nodes and collecting the results back would be more costly than doing the computation locally. LADLE detects such cases, and takes the appropriate action: rather than distributing the work among all the processes, it localizes the computation to a single node avoiding all the necessary inter-node communication. Detecting such cases provides a very powerful method to react dynamically at runtime to the coherence unit of the execution platform.

6.9 Interacting with FCS

Chapter 4 described the interface and policies of the Fair Cooperative Scheduling mechanism. LADLE handles the cooperation with FCS and peer processes transparently. When a message is sent and a response is expected, or a synchronization point is reached and arrival of another process is expected, LADLE checks the "Is_Running" and "Is_Signaled" status of the process expected to respond. If the process is found not to be running and it is not signaled, a "WAKEUP-SIGNAL" is sent. Once such a signal is sent, "Is_Signaled" is set to prevent other processes from sending more signals. If the process, however, was found to be either running or was signaled by some other peer, nothing is done. Either way, LADLE uses spinning to wait for the expected message
arrival.

If, after spinning for some time (as explained in Section 6.10), no response is received, LADLE yields the processor. However, yielding the processor unconditionally might have some adverse effects. Assume a case where the processor is given away with no conditions, and only a subset of the processors are multiprogrammed. Further, assume that competing processes are not cooperative, and are compute intensive (i.e., always willing to take the processor). Each time LADLE gives away the processor, competing processes will take it, and the parallel application would need to wait to be rescheduled by regular means, or a signal to be scheduled on demand. Since even scheduling on demand takes time for the signal to be polled at a hardclock tick, the time allocated to the cooperating process will be less than its fair share. This in turn would reduce the RelativePower of that process and shift some of its work to its peers. This cycle of negative feedback could continue until all the work is pulled from the process running on the multiprogrammed processor. To avoid this scenario, the processor is released conditionally: i.e., if there is a competing process within 1 run-queue (as described in Chapter 4.1) distance of the yielding LADLE process. In all cases, it is enough to have 1 run-queue distance and a full "emergency-fund" reservoir as a safety net for scheduling on demand.

6.10 Adjusting Spin Time

Perhaps one of the hardest aspects of having coordinated scheduling in a COW, without total control, is locally deciding what to do when no useful work can be done. "Implicit Coscheduling" [22] has shown spin-block — spinning for a while before blocking — saves time in many cases, since if it was to block immediately, an expected response or a message might arrive right after blocking. However, the "spin-block" mechanism blocks, waiting for a message arrival even if there is no other process to take over the processor. It is suggested, though not implemented, that reducing priority
to allow some other process to run, if such a process exists, could be more beneficial, since the blocking and rescheduling cost can be avoided if no such process is found [72]. This scheme was later implemented by other researchers, called “spin-yield”, and found to offer better performance compared to using spin-block [61].

Even though these studies suggest that spinning before giving up the processor is more beneficial, deciding the spin time in many cases is very platform and application dependent. For instance, it was found that, in an application with little or medium amount of communication (on average 5 to 500 ms between messages), and in the presence of load imbalance, yielding the processor immediately rather than spinning performed much better if there was a competing process [21].

The underlying FCS mechanism is to yield the processor only if there is an eligible process, rather than blocking, which is very similar in spirit to “spin-yield”. However, FCS gives more control to the yielding process in determining its willingness to yield. Furthermore, using the FCS provided return value about the yield attempt, the yielding process could implicitly predict whether a competing process actually exists, and use such information to guide its spinning time. Another important parameter while deciding whether to spin or yield is the time it takes to do a context-switch. Unfortunately, finding this value is hard at best, and therefore LADLE does not use it while adjusting the time to spin.

In order to set the spin time, LADLE records the minimum time $T_{min}$ it takes to exchange a message with each of the processes. The base spin time, $T_{sbase}$, is calculated as $2 * T_{min}$. Multiplying $T_{min}$ by 2 is intended to alleviate any fluctuation in message processing time. The choice of 2 was empirically determined to work well. Also determined is the likelihood of having a competing process. Each time a yield attempt returns failure to find a process to yield the processor to, a variable, $L$ is incremented by 1, but never beyond 4. Conversely, each time a yield attempt succeeds (gives away the processor), $L$ is decremented by 1, to a minimum of 1. The time to spin is then set to $T_{spin} = L * T_{sbase}$. Determining the spin time as a parameter of the messaging
time is platform independent and does not need user input, as well as does not assume existence of the same underlying network between all the peer processes.

6.11 Handling Irregular Applications

In some cases however, it is very hard, or even impossible, to find a regular structure that characterizes the shape of the data, or the way the application accesses it (in other words, it cannot be described by a regular section descriptor (RSD) [40]). In order to provide load balancing support for that sort of application, LADLE-IRR was implemented.

Similar to the LADLE-BAR and LADLE-IND implementations, progress is used as the metric of load balancing, and RelativePower is used to represent progress. However, unlike LADLE-BAR or LADLE-IND, LADLE-IRR does not decide how the work (and correspondingly the data) is distributed since LADLE-IRR is not expected to know the size or the shape of the data, or the boundaries of the parallel loop being executed. The general design is to provide the programmer with an interface that can be used to guide load assignment and reassignment.

The goal is to provide an interface that allows the programmer to parallelize the application assuming an underlying system that guides the proportional distribution of work. LADLE-IRR can manage a token pool divided between the processes proportional to their perceived RelativePower. Token pool management is, in many ways, similar to task-queues. However, tokens need to be associated with computation and data by the programmer. In a particle simulation application, for instance, each token could represent one particle or a group of “n” particles. Even though in many particle simulation programs the structure of the program can not be represented using an RSD, it is fairly easy for a programmer to associate a subset of particles with a token and write the program accordingly. LADLE-IRR also provides calls to retrieve the computed RelativePower of any process, should the programmer decide to use that
metric for a different purpose. The following is a summary of the exported interface, with a brief explanation of each provided call.

- **StartProfiling(void)**: This library call starts program monitoring in order to provide feedback to the application. It is conceivable that some programs have some initialization phase that should not be profiled, as profiling them would return misleading timing results.

- **StopTimingRegion(void)**: This call informs the runtime system that the profiling should be suspended temporarily until a request to restart is received. Programmers typically would use this call to prevent a sequentially executed segment of the application from being profiled, since the result might once again be misleading.

- **StartTimingRegion(void)**: Restarts profiling after it was suspended.

- **GetRelativePower(pid)**: Returns the **RelativePower** of the process with the provided id.

- **GetCumulativeRelativePower(pid)**: Returns the sum of **RelativePowers** of all processes with an id smaller than “pid”.

- **CreateTokensForRegion(region_id,num)**: Create **num** tokens for the parallel region with id **region_id**, if it has not already been done. Once the tokens are created, they are distributed among processes proportional to the observed **RelativePower**, and redistributed each time the **RelativePower** is recalculated. Calling this function multiple times has no effect.

- **GetNextToken(region_id)**: Return the next token from my process’ token queue, if available, if not -1 is returned.

- **ResetRegionTokens(region_id)**: Mark all tokens as not taken for the given parallel region.
- **MayNotBalanceRegion(region_id):** Inform the runtime system not to change load assignment until further notice. This call should be used when a parallel region depends on local state (non-shared data) based on the current work assignment, and hence changing the assignment would yield incorrect results. Even though load balancing can not be done, monitoring progress continues.

- **MayBalanceRegion(region_id):** Inform the runtime system that it may change load assignment if necessary.

```c
// A sample particle simulation program. Even though the tokens to be
// associated with each process are decided by LADLE-IRR, the group of
// "numParticlesPerToken" particles to be associated with each token
// is decided by the programmer.
int numParticlesPerToken, curToken, begin, end, i;
CreateTokensForRegion(region_id, num_tokens);
while((curToken = getNextToken(region_id)) != NO_TOKEN){
  begin = curToken * numParticlesPerToken;
  end = begin + numParticlesPerToken;
  for (i = begin; i < end; i++){
    ...
  }
}
ResetRegionTokens(region_id);
```

Figure 6.6: A typical use of the provided interface, when many smaller pieces are created and each piece is associated with a token. In our example, we assume token 0 is associated with particles 0..numParticlesPerToken-1, token 1 is associated with particles n..2*numParticlesPerToken-1 and so on.

A typical program written using LADLE-IRR looks similar to what is seen in Figure 6.6. The program resembles a typical particle simulation application. In this example the programmer creates a number of tokens, with each of the tokens representing of a group of particles. The group being represented is determined by the programmer and is transparent to LADLE-IRR.

Locality management by LADLE-IRR is not possible since LADLE-IRR does not know anything about the data. However, as an attempt to preserve locality, whenever load is balanced, the minimum number of tokens are moved around in the hope of preserving the locality of the associated data.
6.12 Summary and Discussion

Our implemented Locality Aware Dynamic Load-balancing Environment, LADLE, provides the ability to balance load arising due to one or more of inherent application behavior, multiprogramming, or heterogeneous hardware. LADLE uses the progress of a process to determine its capability to finish a given assignment. That capability is represented by a metric called RelativePower, which represents the amount of work a process is capable of finishing in a unit of time. Computing RelativePower requires knowing a process’ completed work, and the elapsed time for finishing that work. Both of these variables are independent of the cause of load imbalance, and can be easily computed on any platform.

Aside from balancing load, LADLE pays careful attention to locality of data. As long as load is balanced, assignment of load to processes are preserved throughout execution. Even when load imbalance arises, for a large class of applications, only assignments of processes that are lagging behind and those racing ahead change, leaving other processes undisturbed. LADLE also uses the exported interface of FCS in order to communicate scheduling on demand requests. Also, LADLE uses the exported “yield” call of FCS to yield the processor in the absence of any useful work.

Our implementation and evaluation of LADLE has been done on a particular platform. In the following sections, we will discuss the potential effects of porting LADLE to a different platform, particularly not having a shared memory system. Also, we will discuss the effects of having a heterogeneous system, since even though we argued LADLE was capable of handling heterogeneity induced issues, all our experiments were conducted on a homogeneous system.

6.12.1 Effects of Heterogeneity

Heterogeneous nodes of a cluster offer different computational capabilities. A load balancing system should either know or deduce that capability properly for load bal-
ancing to be effective. As mentioned in Chapter 2 most load balancing systems simply assumed homogeneous hardware and emphasized load imbalance due to inherent application behavior [15]. Systems such as LSF [17, 90] require the user to provide some constants representative of each node’s capability.

Load imbalance in a parallel application, regardless of the underlying cause, results in some processes progressing faster than their peers. Whether one node has faster processor, more memory, or faster I/O than its peers will be captured by that node’s ability to progress faster than its peers.

LADLE bases its load balancing decisions on the progress of an application’s processes. Using progress as the metric of load balancing makes LADLE capable of dealing with heterogeneous hardware as well as any other cause of load imbalance.

### 6.12.2 Effects of Not Having Shared Memory

For our implementation of LADLE, we assumed the presence of a shared memory system, which is not the default programming paradigm for a cluster of workstations.

LADLE, in terms of load balancing, fundamentally makes two decisions: 1) The proportion of work assignment 2) The time a new load assignment becomes effective for each of the processes. The existence of shared memory does not affect either of these two decisions. The metric we use, RelativePower, is independent of the programming paradigm. Similarly, the time a new assignment takes place is independent of the programming paradigm. Nevertheless, the presence of a shared memory system drastically simplifies movement of data associated with each reassigned task. Once LADLE decides it is allowable to have a task move to its newly assigned node, the actual data movement is handled implicitly by the underlying shared memory system when the task is accessed at its newly assigned node.\(^3\)

\(^3\)Our underlying shared memory system’s coherence protocol is release consistency. Therefore, for these changes to take effect, a *release* operation on the former node has to be followed by an *acquire*
If a shared memory system did not exist, the two fundamental tasks of LADLE would still be carried out. However, the data to be moved would need to be identified precisely by other means. It is possible to identify the moving data with the help of either a compiler or a runtime system, or a combination of both.
7 Evaluation

In this chapter, we first describe the platform used for evaluating our system. Our first set of experiments uses some microbenchmarks to evaluate the effects of multiprogramming on communication and synchronization operations, and show how effective FCS is in countering these effects. Afterwards, FCS and LADLE are tested under different scenarios to evaluate their effectiveness in tackling complications arising due to multiprogramming or inherent application-behavior induced load imbalance.

7.1 Evaluation Platform

Our experimental environment is a cluster of eight Compaq AlphaServer 4100 workstations. Each workstation (node) is equipped with four 21164A processors operating at 600 MHz, totaling 32 processors. The 21164A has two levels of on-chip cache. The first level consists of 8 KB each of split direct-mapped instruction and (write-through) data cache, with a 32-byte line size. The second level is a combined 3-way set associative 96 KB cache, with a 64-byte line size. Also there is an 8MB board level direct mapped cache, with a 64-byte line size. The OS running on each AlphaServer is Tru64 4.0F, with TruCluster v. 1.6 extensions. Each node is equipped with 2 GB of shared memory, and a Memory Channel network interface. The Memory Channel [34] is a PCI-based crossbar network, with a peak point-to-point bandwidth
of approximately 83 MBytes/sec (664 Mbits/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 32-bit write operation is 2.7 $\mu$s, and for a 64-bit remote-write operation is 3.3 $\mu$s.

On our platform, a hardclock ticks every 500K processor cycles, corresponding to a frequency of 1200Hz. At every tick, it is checked whether any event is waiting, and the scheduler is entered to check whether the current quantum has expired, or a higher priority runnable process is waiting to be executed. A scheduling quantum is set to be 12 hardclock ticks, approximately 10ms. A process runs until the quantum expires, unless there is a higher priority process. A null system call takes approximately 0.5 $\mu$s and a context switch takes approximately 6 $\mu$s.

Cashmere-2L [77] is used as the underlying software distributed shared memory system (SDSM). Cashmere-2L uses the virtual memory (VM) system to detect coherence operations, and therefore has a coherence unit of 1 VM page — 8KB. The round-trip latency for a null message in Cashmere-2L is 15 $\mu$s. This time includes the transfer of the message header and the invocation of a null handler function. A page fetch operation takes approximately 220 $\mu$s on an unloaded system. Since Cashmere-2L allows multiple writers to modify the same coherence unit, it might be necessary to create an identical copy (twin) of a page prior to modifying it, so that updates of multiple writers could be combined. Upon completion of an update, the difference between the twin and the modified copy is generated (diff), and that difference is used to update the master copy of the page. Twin operations take 68 $\mu$s, and a diff operation ranges from 100 to 245 $\mu$s, depending on the size.

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

7.2.1 Effects of Multiprogramming: Microbenchmarks

A set of experiments were conducted to measure the effects of multiprogramming on communication and synchronization operations, which are common for parallel applications. Multiprogramming is known to cause uncoordinated scheduling among the processes of a parallel application if nothing is done about it. Multiprogramming might also cause load imbalance by having some nodes more loaded than others. The effects of load imbalance caused by multiprogramming will be demonstrated with actual benchmarks. This section will focus on the effects of uncoordinated scheduling. Finally, we will establish how effective FCS is in its ability to coordinate interacting processes.

The first set of experiments is conducted to measure the effects of multiprogramming on message latency. For that purpose, a pair of identical processes exchange 1-word messages. In the absence of any competing process, the one-way latency for a 32-bit remote-write operation is 2.7 $\mu$secs. Sending a message is done by writing to a pointer in the network address space, which is then automatically written to the local memory of the remote process. The network does not generate any events on either end while sending messages, and normal use of the interface requires periodic polling to check for message arrivals. However, it does provide a means to send signals explicitly to peers running on other nodes. Each process accounts for the time it takes from the moment it sends the message until a response is received. Processes do some tight loop computation, and continuously poll for message arrivals between sending two messages. The amount of computation is a randomly generated, uniformly distributed series that would have taken between zero and $2\times \bar{x}$ ms on a dedicated machine, where $\bar{x}$ indicates the average of the series. The average dedicated computation between two messages ($\bar{x}$) is set to 12 different values between 0.5 ms and 17 ms, and for each of the values, the experiment is repeated 8 times, with each run exchanging 10K messages.
The number of processes competing with each of the sender and the receiver are also varied from none to 3, creating 16 different combinations. The competing processes are all compute-intensive, with very small memory footprints. All processes are bound to the associated processor to ensure they compete with the intended process. We call the process on which timing measurements are made the sender, and its counterpart the receiver.

Figure 7.1: Average message latency as a function of multiprogramming and message frequency. Each of the graphs shows averages for a different number of competing processes on the message recipient side, changing from 0 to 3. The 4 groups in each graph shows results with different number of processes competing with the sender. The average amount of computation is varied from 0.5ms to 17ms, and each of the marks in each group shows the result for that. For each computation amount, the same experiment is repeated 8 times. The bars indicate the smallest, average and largest (i.e., the spread) of the average message latencies of each of the runs. Note that the scale of each graph is different.

The results of this experiment are shown in Figure 7.1. The four graphs in the
figure are grouped by the number processes competing with the receiver. The 4 groups in each of the graphs is the number of processes competing with the sender. Each point in a group shows the message latency with a different amount of average computation between two messages. The amount is varied from $0.5ms$ to $17ms$.

![Diagram showing Time and Cases 1, 2, 3](image)

Figure 7.2: In a priority based scheduler, compute-intensive processes of equal priority are scheduled in a round-robin fashion. The percentage of messages answered immediately between two processes is directly proportional to the time the two processes run simultaneously. In each case there are two nodes, and change of shade indicates a context-switch. Assuming processes of the same shade are communicating, the processes in “Case 1” are coscheduled, while the processes in “Case 3” are running completely out of sync.

The existence of higher variation in some sets of experiments than others is clearly noticeable. That variation is higher when the total number of processes running on the sender node and the receiver node (including the sender and receiver processes) are integral multiples of each other. In the case of compute intensive processes, competing processes are scheduled in a round-robin fashion. Therefore, when the number of processes on one node is an integral multiple of the other, depending on how much skew there is at the beginning of the computation, the same skew will be preserved
throughout the computation. Figure 7.2 shows a case with two processes. Rows in each case represent a node, and each shade represents a process, with two processes of the same shade on different nodes assumed to communicate with each other. In "Case 1", communicating processes are coscheduled, and messages will be responded to immediately. "Case 2" shows some skew, which repeats itself throughout execution. "Case 3", on the other hand, shows the case where every exchanged message is responded to after exactly one quantum. The variation in average message times when the number of processes on two nodes are integral multiples of each other, therefore, depends on the skew at the time the computation started, and that is what we see in Figure 7.1. When the number of processes on one of the two nodes is not an integral multiple of the other, however, the overlapping time during each execution is almost always constant, regardless of the initial scheduling skew, and hence there is very little variation.

Another thing to notice in Figure 7.1 is the small effect multiprogramming on the sender node has on the observed latency. Even when the number of processes competing with the sender is 3, while the number of competing processes is 0 on the receiver, the observed latency averages less than 8\,\mu s. However, when the number of competing processes is high on the recipient side, even if there is no competitor on the sender side, the observed latency is in the order of milliseconds. Also, as the message frequency increases, the observed message latency goes down. The reason for such behavior can also be seen in Figure 7.2. When message inter-arrival time is smaller, the number of messages that will be sent and replied to immediately during the simultaneous execution period will be higher, therefore bringing the average latency per message lower.

Figure 7.3 shows the observed message latency when FCS is used to coordinate the communicating processes. Only cases with equal number of processes competing with the sender and receiver are shown, since almost all the cases exhibit the same behavior. The cases shown in the figure are a subset of the sets of experiments in Figure 7.1. The results shown in this figure show FCS to be effective not only in reducing the message latency, but also in making the results predictable regardless of how multiprogramming
Figure 7.3: Effectiveness of FCS in reducing the message latency in the presence of multiprogramming. The experiments here are a subset of those in Figure 7.1, showing only the cases with number of processes competing with the sender and receiver being equal. Similar results were obtained for all other cases, and were eliminated due to redundancy. The graph shows the cumulative distribution of the fraction of the total number of messages with a given latency, for 99% of the messages. Understandably, in the case of no competitor, all messages are responded to almost immediately. However, even in the case of 3 competitors on each side, about 70% of the messages are responded to immediately, with 99% of all the messages being responded to within 650\(\mu\)s. Some messages of the remaining 1% took in the order of milliseconds, due to FCS refusing to schedule the recipient process, as it would have been unfair to do so.

In order to determine the effects of multiprogramming on barrier synchronization, similar experiments were conducted on 32 processors. Similar to the messaging latency experiments, a uniformly distributed random sequence was generated to determine the amount of work between two consecutive barriers. For each experiment, the value of \(x\) is chosen to be between 0.5\(ms\) and 17\(ms\). Since all the processes use the same seed, they generate the same sequence, and therefore they all do the same amount of computation between two barriers. Also, since the same number of processes compete with all the processes of the parallel application, no load imbalance across processors
Figure 7.4: Average barrier time on 32 processors, with different number of processes running on the same processors as the processes of the parallel application. The 4 groups indicate the average barrier time for cases with the number of processes competing with each process of the parallel application varied from 0 to 3, with the leftmost group being 0 and the rightmost being 3. The x-axis ticks show the average amount of computation between each barrier.

exists. Each experiment executes 10K barriers and is repeated 4 times.

The results of the experiment are shown in Figure 7.4. For the experiment on a dedicated cluster of 32 processors, average time per barrier changes between 48μs and 383μs, depending on the amount of computation between two barriers. With 3 competing processes running along with each of the processes executing the barriers, the average time per barrier goes beyond 40ms, an almost 3 orders of magnitude increase. The increase is a function of the number of competing processes and the scheduling quantum. Each time the number of competing processes is increased by one, the average time per barrier increases slightly more than one quantum.

One interesting behavior is the steady increase in barrier time as the amount of computation between two barriers increases. Further experimentation showed that some daemons were being scheduled, with each daemon taking up-to 2.5ms. With longer
periods between barriers, it is conceivable that one of the 32 processes is more likely to have a daemon running, which will cause that process to reach the barrier later than its peers, hence increasing the time it takes to execute a barrier.

Figure 7.5: Graph of the experiment shown in Figure 7.4 with FCS support being turned on. FCS, reduced the average barrier time considerably, in addition to making it predictable (i.e. independent of the amount of multiprogramming).

Figure 7.5 shows the execution of the same experiment, with FCS support turned on. Regardless of the number of competing processes, FCS reduces the average barrier time to less than 1 ms. Even though this is still much higher than the minimum achievable time when the processor is dedicated, reducing the time per barrier from more than 40 ms to below 1 ms is a considerable reduction. We still see the steady increase in the time per barrier, as the amount of computation between two barriers is increased, due to daemon activity. Daemons generally run at a priority higher than that of user processes, and FCS does not boost the priority of a process to beyond what is allowable for a user process.
7.2.2 Benchmark Application Experiments

In order to evaluate our system, we used several benchmarks with different characteristics. Following are descriptions of these applications and results of experimental evaluation under different scenarios.

**Gaussian Elimination with Barriers:** A parallel Gaussian elimination algorithm. The solution is computed by using partial pivoting and back substitution. Upon applying the current pivot, all processes reach a barrier, where they wait until the next pivot is computed. The application peels one row of the matrix at each iteration, and therefore, the total work decreases as the computation progresses. This behavior calls for cyclic distribution of iterations among processes.

**Gaussian Elimination with Flags:** The same parallel Gaussian elimination algorithm as above, written using flags rather than barriers. 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 work with different pivots at any given time. Furthermore, flags are known to be less affected by multiprogramming than barriers [57]. However, LADLE-BAR is not able to handle applications without barriers. Therefore this application is not evaluated using LADLE-BAR.

**Jacobi:** An iterative method for solving partial differential equations with nearest neighbor averaging as the main computation. Two arrays are employed, with one being the 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 tasks rather than changing the number of fixed tasks assigned to each processor.

**Matrix Multiply:** A matrix multiplication algorithm parallelized by forming tasks with groups of rows and distributing these tasks among processes. The dataset consists
of three 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 not sensitive 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, 13 2-dimensional arrays and 11 parallel regions are spanned, with some of the regions taking only a few milliseconds, and involving a single row of the matrix.

**SOR:** A red-black successive-over-relaxation is a nearest neighbor averaging algorithm from the TreadMarks [5] distribution. The application is used to solve partial differential equations. This application exhibits nearest neighbor sharing, and a single task per process is created with one block of data sized proportional to the perceived RelativePower of that process.

**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. The number of vertices in the graph is 4K, leading to a 4Kx4K connectivity matrix of integers.
Water-NSquared: A molecular dynamics simulation from the SPLASH-1 [70] benchmark suite. It is run for 3 time 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 to 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.

<table>
<thead>
<tr>
<th>App</th>
<th>Input Size</th>
<th>Seq. Time</th>
<th># of Barriers</th>
<th># of Locks</th>
<th># of Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gauss-Bar</td>
<td>4Kx4K (134M)</td>
<td>2176</td>
<td>16384</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Gauss-Flag</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>131040</td>
<td>0</td>
</tr>
<tr>
<td>Jacobi</td>
<td>8Kx8K (536M)</td>
<td>995</td>
<td>401</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Matrix Mul.</td>
<td>2Kx2K (100M)</td>
<td>5038</td>
<td>8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MGS</td>
<td>4Kx4K (134M)</td>
<td>3735</td>
<td>4096</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Shallow</td>
<td>2Kx2K (438M)</td>
<td>1153</td>
<td>1399</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SOR</td>
<td>8Kx8K (536M)</td>
<td>1499</td>
<td>3997</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Tran. Closure</td>
<td>4Kx4K (134M)</td>
<td>1371</td>
<td>4089</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Water-Nsq</td>
<td>32K (22M)</td>
<td>3537</td>
<td>70</td>
<td>1114432</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.1: Applications used for evaluation of our system. Their input size, the time it takes for each of the applications to execute on a uniprocessor (in seconds), and the number of different synchronization operations performed by each of them when they are run in parallel on 32 processors, are shown.

<table>
<thead>
<tr>
<th>Label</th>
<th>Node1</th>
<th>Node2</th>
<th>Node3</th>
<th>Node4</th>
<th>Node5</th>
<th>Node6</th>
<th>Node7</th>
<th>Node8</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-Load</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Load-4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Load-8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Load-16</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 7.2: Names and meanings of different load scenarios used for experiments. "Node" indicates a 4-way SMP node of our experimental platform. The number underneath each node show how many processes, other than those of the parallel application, run on that node. The last column indicates the total number of these processes, on the whole cluster. In all cases, the parallel application is running on 32 processors.

The applications, their data size, their sequential execution time (without SUIF transformations), and the number of synchronization operations they perform when running in parallel on 32 processors, are shown in Table 7.1. In order to evaluate the
effectiveness of LADLE, several cases are devised with different numbers of contending processes. Contending processes, in this experiment, are all compute intensive sequential processes, which run in a tight loop and monitor whether they are being treated fairly. Each process continuously monitors its progress. Every second, it checks whether its progress is less than half of what it would have been, if it was running on a dedicated processor. If so, it prints a complaint. Table 7.2 shows the labels of each scenario and its explanation.

**Effectiveness of FCS**

![Graph showing multiprogramming overhead on 32 processors with 16 multiprogrammed.](image)

Figure 7.6: Overhead percentage when running applications on 32 processes, with 16 of the processes having a competing process running alongside them. Ideally, an application taking “t” seconds on a dedicated cluster should take “2t” seconds when 16 processes have a competing process running alongside them. Any amount of time above “2t” is assumed to be overhead, and that amount is divided by “2t” to find the overhead percentage.

The first result we show is the effectiveness of FCS with our application suite, when applications are running on 32 processors with 16 of these processors running one more process along with the process of the parallel application. In the absence of load bal-
ancing, the execution time is expected to be twice that of the time when the application is running on a dedicated machine. Any amount of time above that is assumed to be overhead (either due to lack of coordination, or context switch time, or cache pollution, even though competing programs are too small to cause considerable cache pollution). In all the cases FCS alone seems to be capable of reducing the multiprogramming incurred overhead to below 22%, in some cases (Gaussian elimination with barriers) from more than 90%.

Effectiveness of LADLE-BAR and LADLE-IND

Experimental results of LADLE-IND and LADLE-BAR are shown in Figure 7.7. Each of the 4 groups in every graph shows the execution times under different load conditions as explained in Table 7.2.

Throughout the execution, none of the competing processes complained about being treated unfairly, indicating they received their fair portion of the processor for every second of their execution (actually, FCS might favor them a little while the parallel application is building up equity and yielding the processor aggressively).

The “No-Load” case shows different executions of an application with different features being enabled, and is intended to show the incurred overhead in the absence of any competing load. In all cases the overhead incurred due to FCS or LADLE is fairly small. Two applications, however, see benefits by having some LADLE features turned on even in the absence of any competing process. Transitive closure consistently benefits from having intra-node task stealing, even in the absence of any competing process. The graph being traversed by this application is filled with random values, and certain work needs to be done only if the graph has an edge between any two picked vertices. It is likely to have some of the processes receive more edges than others, at least in the short term. Task stealing alleviates this case by moving some of the work from a process which happened to have more edges than its peers. Having task stealing
for intra-node load balancing proves to be helpful in dealing with the short-term load imbalance.

Shallow is the other application that benefits from having LADLE-IND or LADLE-BAR features turned on. This application has some parallel regions that operate on a very small amount of data, with very little computation. By default, this data is distributed among all the processes, and the results are gathered back. By monitoring the computation and communication, LADLE decides to localize the computation and avoid data distribution, which in turn reduces the execution time of the application.

Since LADLE-BAR can not handle applications without barriers, Gaussian elimination with flags can not be load balanced using LADLE-BAR (accounting for the missing bar in the graph). However, another thing to notice is the execution time difference between Gaussian elimination using barriers and flags. Even on a dedicated machine, the implementation of Gaussian elimination using barriers takes about 40% more time to execute than the one using flags, for the same data set. Similar to results obtained by earlier work, applications using barriers, especially those frequently synchronizing tend to be more adversely affected by the presence of multiprogramming [57].

In the case of “Load-4”, since all the competing processes are running on a single node, load imbalance between nodes is severe. However, there is no intra-node load imbalance. Therefore, task stealing, which is intended for intra node load imbalance, is not expected to help much. Just as expected, the execution time is reduced minimally by having the task-stealing feature turned on. FCS helps a little by coordinating the processes to reduce the scheduling skew induced wait time. LADLE-BAR and LADLE-IND, with their ability to do inter-node load balancing seem to help the most.

The “Load-8” group of the graph shows a case with intra-node load imbalance. One of the processors in each node is running two processes, with the rest being dedicated to the parallel application. However, the number of processes running on each node is the same without any inter-node load imbalance. Task stealing, which is intended for intra-node load balancing, is expected to help most. With the exception of Jacobi
and Modified Gramm Schmidt (MGS), that turns out to be the case. Neither Jacobi nor MGS benefit from task stealing since both of these applications exhibit nearest neighbor sharing. When nearest neighbor sharing exists, only one task is assigned to each process, not providing any chance for task stealing. Having inter-node load balancing is not expected to help any application other than Jacobi and MGS. Jacobi and MGS benefit from LADLE-BAR and LADLE-IND, by having their tasks resized properly. The remaining applications, on the other hand, incur some overhead without any benefit from inter-node load balancing attempts. However, the incurred overhead (due to extra computation and communication associated with recalculating a new assignment) is fairly small in all the cases.

“Load-16” shows a case in which 4 nodes are overloaded, with 4 processes competing with the parallel application on each, while the remaining 4 nodes are dedicated to the parallel application. With the high number of competing processes, coordination of the processes becomes an important issue. Since all the processes within a node are running the same number of processes, there is no intra-node load imbalance. However, inter-node load imbalance is severe. With the exception of matrix multiply, all the processes benefit from FCS considerably. With that high number of competing processes, lack of coordination among processes of the parallel application becomes a major cause for inefficient use of resources. FCS, by providing coordination, reduces the execution time in all the cases. Matrix multiply is not affected by FCS, since with its long episodes of computation and lack of synchronization and communication, it is indifferent to the presence of multiprogramming in the first place.

Task-stealing, in the case of “Load-16”, does not help since all processors of a node are load balanced already. However, having LADLE-BAR and LADLE-IND helps in all the cases, reducing the execution time dramatically.

LADLE-BAR and LADLE-IND, in many cases, perform comparably. LADLE-BAR, in general, has the advantage of having slightly less overhead compared to LADLE-IND due to not having to deal with maintenance of per task time-stamps, and not com-
municating such information at the time of load balancing. However, that change seems to be negligible in many cases. On the other hand, LADLE-IND seems to perform better even for applications that use barriers for synchronization, when the time between two consecutive barriers is long enough to cause the application to run unbalanced for an extended period of time. Matrix multiply, for instance, has an average period of about 30 seconds between two consecutive barriers, for which time LADLE-BAR will not be able to detect any existing load imbalance. LADLE-IND, however, would be able to detect and fix that case promptly.

The results show FCS to be effective for all applications, especially for those with high amounts of communication and synchronization. LADLE-IND without FCS, on the other hand, is beneficial only when the application has little communication and synchronization (e.g., matrix multiply). LADLE-IND alone fails to be very effective in the case of other finer granularity applications, especially when the number of multi-programmed processors is high. As our results show, FCS and LADLE are complementary, both being necessary to achieve the best performance over a range of experimental environments.

For the “Load-16” configuration, FCS alone reduced the execution time by as much as 44% (in the case of Gaussian elimination with barriers). LADLE-BAR and LADLE-IND, for the same case, reduced the execution time considerably as well, sometimes by as much as 40%, on top of the reduction achieved by FCS. The overall reduction in execution time having all the features on was as high as 59%, which in some cases brought the execution time to within 8% of what is ideally achievable.
Figure 7.7: Execution times of applications under different scenarios. The x-axis labels indicate the type of load, as explained in Table 7.2. The 6 bars in each group show the time it takes to execute the application under a different setup. The first bar of each group is when nothing is done. The second bar shows the case when FCS support is turned on. The third bar shows the effect of task stealing for intra-node load balancing, along with FCS. The fourth bar is LADLE-IND (task stealing and load balancing, independent of synchronization) without FCS. The fifth bar is LADLE-BAR with FCS, task stealing, and load balancing turned on. The sixth bar shows the case of LADLE-IND with all the features turned on, including FCS. Finally, the rightmost bar shows the ideally achievable execution time, and is given as a reference for comparison. The ideally achievable time is calculated as a factor of the completion time on a dedicated machine without any load balancing or task stealing. For instance, in the case of LOAD-4, since 4 of the processors are time-shared, the parallel application nets 28 dedicated and 4 half-time processors, totaling 30 processors. Ideally, the execution time of a parallel application running in such a configuration can be at best (32/30) of its execution time on a dedicated cluster.
Effects of Memory and Compute Intensive Load

Figure 7.8: Execution times of parallel applications in the presence of a memory intensive competing application. All the experiments were conducted using 32 processors, with 16 of them running the memory intensive application along with the timed benchmark application. The 16 multiprogrammed processors are on 4 of the SMP nodes, leaving the remaining 4 SMP nodes dedicated. Therefore, there is no load imbalance within nodes. The labels in the legend are the same as those shown in Figure 7.7.

In order to evaluate the performance of LADLE and FCS when competing processes are not only compute intensive, but also have a large memory footprint, we conducted experiments with competing applications having a large memory footprint. The results are shown in Figure 7.8. The large footprint is intended to stress the memory bus and pollute the cache. All the experiments were conducted using 32 processors, with 16 of these processors being multiprogrammed. The competing application is an unoptimized sequential matrix multiply application, with 3 matrices, 8MB each.

When nothing is done about balancing the load, or scheduling the processes in a coordinated manner, the execution time more than doubles. Applications with large active datasets and those frequently communicating and synchronizing suffer most.

One of the drawbacks of FCS is the increased number of context switches, since it schedules processes on demand for a fraction of a quantum. Even though that increases the total time spent on context switches, the effects of cache pollution are well amortized. In all the cases, the execution time of the application decreased by simply
having FCS turned on. In order to monitor whether the competing sequential application is being treated fairly, we recorded the execution time of that process throughout execution. The increase in execution time of the competing sequential application for all the experiments was between 108% and 122%, compared to the execution time on a dedicated processor. For comparison purposes, two copies of the sequential application were run on each processor of a node, using the unmodified scheduler, and an increase of 125% in the execution time of each process was observed. This indicates that the sequential application finished in shorter time when competing with one of the parallel applications, rather than when two identical copies of the same application were running. Despite the higher number of context switches, this could be due to: 1) Not all our parallel applications stress the memory as much as the competing application does 2) Parallel applications using FCS tend to give up the processor, rather than waiting, and perhaps give more processor time to the competing sequential application.

In all the cases, the combination of FCS and LADLE reduces the execution time considerably. Even though LADLE-IND alone is useful, since it balances load across nodes, its effect is limited particularly for frequently communicating and synchronizing applications, such as gaussian elimination with barriers. FCS alone does a fairly good job reducing the execution time, especially for frequently communicating and synchronizing application. The combination of the two performs better than either component alone in all the cases, except matrix multiply where LADLE-IND alone performs better than LADLE-BAR+FCS. The reason for such odd behavior lies within the characteristics of the matrix multiply application. Matrix multiply is indifferent to scheduling skew, and therefore FCS has limited use. Furthermore, due to its long periods of computations between two barriers, LADLE-BAR would not be able to fix load imbalances immediately, leaving the application unbalanced for an extended period of time.
Figure 7.9: Evaluation of FCS, LADLE-BAR, and LADLE-IND in a highly dynamic environment, with many I/O and memory intensive short-lived processes. In order to get a reasonable reading of the performance, each application is made to conduct its computation multiple times to take sometime between 1500 and 1700 seconds.

Coping with Dynamically Changing Load

So far all the experiments were conducted with competing applications being present throughout the execution of the parallel application. It is more likely, however, that some processes come and go while the parallel application is running. Furthermore, all the competing processes were compute intensive and did very little I/O (printing diagnostic messages). It is more realistic to conduct tests with short term competing applications, with each of the competing applications having a bigger memory footprint, and performing I/O.

For that purpose, building a large library was used. The build compiles multiple files and creates a library archive. Once the compilation is done, all the newly created files are deleted and the same operation is repeated after sleeping for a period of time. A total of 33 source files are compiled with the whole compilation process taking approximately 43 seconds on a dedicated node. After deleting the newly created object and archive files, the build script sleeps for a period of 43 seconds. Nine instances of the same build process are started on all 8 nodes, with one of the nodes having two instances. Compilation is interesting, because it has considerable amount of I/O and
frequent interaction with the OS. Furthermore, the compiler has a large memory footprint, and creates considerable amount of memory traffic. Since many processes are created in a short period of time, in the interest of being fair to all the processes, the Tru64 scheduler uses a different processor of the SMP for each of the newly created processes, one after another. Therefore, all the processes of the parallel application are disturbed for a short period of time, with none of them having a consistent load for a long period.

With many instances of the script running, and each instance sleeping between two consecutive builds, it is very unlikely to get a consistent execution time reading for a parallel application running for a short period of time. Therefore, the execution time of parallel applications is increased, to approximately 1500 seconds, by repeating the same execution multiple times back-to-back (For instance, solve multiple linear equation systems back-to-back, in the case of Gaussian elimination).

Figure 7.9 shows the result of this experiment. As explained above, LADLE-BAR can not balance applications that are not using barriers. Hence, again, implementation of Gaussian elimination using flags is not evaluated for LADLE-BAR. In all the cases, both LADLE-BAR and LADLE-IND seem to cut the execution time considerably, compared to not doing anything. In some cases the reduction is above 30%.

7.3 Running Multiple LADLE/FCS Applications Simultaneously

All the experiments so far were conducted by having a group of sequential applications competing with a parallel application. It is possible to have multiple parallel applications using LADLE and FCS coexist. In this section, we conducted some experiments with multiple parallel applications running simultaneously. Even though, from the load balancing point of view, whether the competing application is sequential or par-
allel hardly matters, effects of multiple processes interacting with FCS simultaneously might cause unexpected behavior by distorting the scheduler behavior. Experiments in this section were conducted using six 4-way SMP nodes totaling 24 processor ¹

Our first set of experiments involved running two copies of the same application on all the processors simultaneously, with all the features of LADLE-IND and FCS are turned on. The goal is to observe whether both instances of the same application receive the same amount of time from the scheduler, and how LADLE behaves under these circumstances. In all the cases the completion time of both instances were within 1 second of each other, clearly showing that LADLE-IND and FCS preserve the proportional sharing of resources and produce predictable results.

![Multiple Parallel Applications Running Simultaneously](image)

Figure 7.10: Normalized execution times of parallel applications when they are competing with another parallel application. Experiments are conducted using six 4-way SMP nodes totaling 24 processors with three nodes (12 processors) being multiprogrammed by running another parallel application continuously. The leftmost bar is the base case where the timed application has all the cluster to itself, and it is always at 100. The middle bar shows the normalized execution time in the presence of a competing application and neither LADLE nor FCS is switched on. The third bar shows the result when all the features are turned on.

¹Two nodes of our cluster were under repair at the time of the experimentation.
The next set of experiments involves a parallel application running on all the 24 processors, while another competing application (we picked transitive closure from our set of benchmarks\(^2\)) is running on 3 SMP-nodes, occupying 12 processors. The competing application runs continuously, restarting immediately after it finished. Figure 7.10 shows the normalized execution time of all the parallel applications. In the presence of a competing application, 12 processors are timeshared. The timed parallel application nets 12 dedicated and half-time of the 12 timeshared processors. Ideally, the timed parallel application should behave as if it had 18 processors to itself, which is roughly 33% more execution time than when all the processors are dedicated to a single application. Even though LADLE and FCS do not provide ideal performance they manage to cut the execution time considerably in all the cases. Moreover, this set of experiments goes to show that LADLE and FCS behavior does not get distorted in the presence of multiple applications using the same interface.

### 7.4 Evaluating LADLE-IRR on Irregular Applications

In order to evaluate LADLE-IRR, two molecular dynamics simulation applications, Moldyn and NBF, were used. For each of the applications, LADLE-IRR was asked to create 512 tokens, and manage the pool of tokens as if it was managing tasks.

Moldyn performs non-bonded force calculations and has a structure similar to that of CHARMM [13]. Non-bonded forces are calculated by considering interactions with only the set of molecules within a certain cut-off radius. The list of molecules in that radius is maintained as an indirection array that identifies interaction partners, and is updated periodically. In our experiments, we simulated 16K particles over 400 iterations, updating the interaction list every 20 iterations. During the course of execution, moldyn performs 217K lock operations, and each process goes through 800 barriers.

\(^2\)The same set of experiments was repeated having water-n\(n^2\) as the competing application with very similar results. We present results here only for transitive closure as the competing application.
Figure 7.11: Evaluation of LADLE-IRR. The x-axis indicates the number and geometry of competing processes as described in Table 7.2. The leftmost bar of each group shows the execution time when no runtime load balancing or scheduling support is present. The second bar from the left is when FCS is turned on without load balancing, and the third bar is LADLE-IRR including FCS. The fourth bar is the ideally achievable execution time, as described in Figure 7.7.

NBF (Non-Bonded Forces) is a kernel for molecular dynamics simulation. It is taken from the GROMOS benchmark [82]. It was previously used as an example to demonstrate compiler-generated irregular message-passing programs [83]. Instead of keeping a list of pairs of interacting molecules like moldyn, nbf keeps a list of interacting partners for each molecule. The lists of partners are concatenated together, with a per molecule list pointing to the end of each molecule’s partners in the partner list. For each molecule, the program goes through the list of partners, and updates the forces on both a molecule and its partner based on the distance between them. In our experiments, we simulated 1M molecules for 100 iterations. During the course of execution, each process goes through 200 barriers.

Figure 7.11 shows the result of running these two applications with the scenarios explained in Table 7.2. Even though LADLE-IRR is not able to determine the data accessed by each parallel region due to indirections and pointers, in all the cases, the execution time is considerably reduced. Moldyn, due to its high number of barriers and lock operations is very sensitive to multiprogramming, and shows the most benefit when
FCS is used. NBF benefits from both load balancing and FCS almost equally. In any case, LADLE-IRR proves to be useful if the programmer uses the provided interface correctly, and performs the right association between the data and the provided tokens.

7.5 Summary and Discussion

Our evaluation of different flavors of LADLE along with FCS has shown that neither of these techniques in isolation are sufficient to offer a solution to inefficient use of resources in a cluster of workstations. When load imbalance is caused by multiprogramming, load balancing techniques fail to provide an efficient solution, particularly to frequently synchronizing or communicating applications. Coordinated scheduling techniques, however, resolve issues related to scheduling discrepancies, with no effect on assignment of work to processes. Our experiments show that a coordinated scheduling scheme coupled with a locality-aware load balancing system is capable of addressing all three causes of load imbalance, and resolving issues arising due to lack of coordination between processes. The system successfully improves the utilization of the processor and reduces the execution time of applications even in highly dynamic environments, with no prior knowledge of process arrival or lifetime.

Our implementation and evaluation of LADLE was done on a homogeneous cluster of 8 nodes, with 4 processors each. We argue that support for hardware heterogeneity is subsumed by our support for multiprogramming (we did not have the experimental platform to verify this experimentally).

Applications running on heterogeneous hardware might suffer from load imbalance, for instance, if all the processes were assigned equal amount of work, and some of the processes happened to run on nodes that lack some of the needed resources. The resources could be memory, processor speed, network bandwidth, or any other resource the process needs. LADLE re-apportions the work among processes based on their past progress, which is used to calculate RelativePower. The RelativePower of a
process running on a multiprogrammed node would be less than its peers running on an identical dedicated node, because it has to share some of the resources it needs with its competitor. Essentially, multiprogramming creates the same effects as heterogeneous hardware, since it creates an imbalance the amount of available resources that would have been available to a process. Also, we believe heterogeneity is simpler than multiprogramming, since, unlike multiprogramming, it does not cause any scheduling issues. Furthermore, since the amount of resources offered to a process running on a node in the absence of multiprogramming does not fluctuate, it is actually simpler to calculate the RelativePower of processes running on dedicated, but heterogeneous hardware. Therefore, we believe RelativePower is a powerful enough metric to capture load imbalance induced by both heterogeneous hardware and multiprogramming.
8 Conclusions

Clusters of workstations (COW) provide a very cost-effective platform for large scale parallel computing. Such environments are built using off-the-shelf hardware components, and are run using highly available software. The ability to grow clusters as the need arises allows their owners to spread their growth over time. The availability of high speed local area or system area networks provides such clusters with the ability to compete with even massively parallel processors (MPP) in terms of performance. In some cases, since they can often be put together in a much shorter time than MPPs, COWs tend to use the most up-to-date components, and perhaps offer even higher performance.

Despite their advantages, COWs bring challenges of their own, particularly when these clusters are used to run parallel applications. The ability to add new nodes to COWs as the need grows raises the issue of heterogeneity, which is a major cause of load imbalance. Furthermore, lack of admission control on these clusters may result in some nodes being more loaded than others. Since each node runs its own operating system and makes its own resource management and scheduling decisions, cooperation between processes running on different nodes could be adversely affected due to lack of coordination. The issues of load imbalance and lack of coordination have been addressed in isolation. Load balancing techniques have mostly emphasized load imbalance either inherent to the application or due to heterogenous hardware. Scheduling
coordination, such as implicit or dynamic coscheduling solutions, has concentrated on
timing issues arising due to multiprogramming, ignoring the load imbalance effect of
multiprogramming altogether.

Load balancing alone would be effective only when the cause of inefficient use
of resources is due to disproportional load assignment, and not due to any schedul-
ing skew caused by multiprogramming. Coscheduling is intended for, and effective
mostly in, cases where all the processors are equally multiprogrammed, and it is pos-
sible to use all the scheduling slots effectively. It has also been shown that simply
running a coscheduling system along with a load balancing system is not sufficient to
achieve the goal of efficient and fair use of resources, because processes running on
non-multiprogrammed nodes would still be forced to wait if the peer they would want
to interact with happened to be de-scheduled.

We have designed, implemented, and evaluated a system of multiple components
aimed at using resources in a COW in a more efficient way. The system targets coordi-
nated fair scheduling for the purposes of reducing communication and synchronization
delays, and at the same time employs a load balancing scheme for proportional load
distribution. Locality of data is taken into account and is preserved when possible. We
summarize our contributions below.

8.1 Contributions

Universally applicable metric for load balancing: Load imbalance could arise
due to three reasons, and LADLE is capable of addressing all three reasons. The
ability is a result of the use of a progress-based metric, RelativePower. The
RelativePower of a process represents the amount of work that process is able
to finish in one unit of time, relative to its peers. The fundamental idea is to assign
more work to a process progressing faster than its peers. Since time is used to estimate
progress, the reason for the rate of progress — memory, multiprogramming, network
bandwidth, etc. — does not affect the outcome, and the metric represents a process' ability to finish a given assignment, independent of the underlying cause.

**Fair Cooperative Scheduling (FCS):** Providing coordination among interacting processes is a topic of interest for any type of inter-process interaction. FCS differs fundamentally from other systems by emphasizing and successfully implementing fairness, and by using a scheme that provides processes with some reward for cooperating with other processes. FCS differentiates between scheduling a process as a result of already having the highest priority, and due to being asked to schedule that process by one of its peers. The fundamental intuition is that scheduling on demand indicates a service request by another process, and services tend to take a shorter amount of time. Therefore, a process running due to an incoming demand is scheduled for a fraction of one quantum.

**Safe communication interface between the application and the OS:** The only input the scheduler takes from the application is the "WAKEUP-SIGNAL" it receives to request that a process be scheduled. The signal goes through all the safety checks, and is interpreted as a hint to guide the scheduling; it could be ignored without any side effect. Even though the scheduler provides information about the scheduling status of a process, the request for such service and the set of processes allowed to see this information is determined by the process whose information is being exposed.

**Concise, powerful interface to communicate statically available information to the runtime system:** The devised interface to pass statically available information is concise and powerful enough to define the structure of most applications. The interface is simple enough to be hand coded, and well-structured enough to be inserted automatically with the help of a compiler. For most applications, the interface can be inserted automatically, without any need for user intervention, generating a program capable of adapting to its changing environment.

**Ability to support a wide range of applications:** LADLE can support applications regardless of the type of synchronization they use. For a wide range of applica-
tions, the procedure of generating dynamically adaptive code is fully automated. Even in cases where it is not possible to extract any static information, or to concisely represent statically extractable information, the programmer is provided with the set of tools that make it possible to write adaptive applications.

**Successful combination of load balancing and coordinated scheduling:** A load balancing system based on progress relies on the correct estimation of progress. FCS, by not favoring any one process over another, provides the necessary tool for a portable load balancing system. Rejected scheduling requests leave a process behind its peers, which eventually provides hints to the load balancing system of the need to take the necessary steps to balance load.

**Hierarchical approach to load balancing:** LADLE differentiates between moving work within an SMP node, and moving it across nodes, over the interconnect. The ability to do so provides the flexibility to attack short term load imbalances within a node immediately and effectively.

**Special emphasis on locality of data:** Locality of data is taken into account while balancing load. Each process maintains its own task queue, and is guaranteed to work on the same assignment as long as it is able to keep up with its peers. When a new load assignment is computed as a result of a load balancing operation, the minimum number of tasks is moved, resulting in the minimum possible data movement. Another special case is when the amount of work is so small that doing the work in parallel, even though possible, is likely to be more costly due to the cost associated with distributing the data and collecting results. Since LADLE monitors communication and computation, it can detect such cases, and as a result localize the computation in an attempt to cut the communication cost.

FCS is capable of addressing scheduling skew issues arising due to multiprogramming very effectively. In the presence of equal levels of multiprogramming on each of the processors, the average time to execute a single barrier has been reduced from more than 40\(ms\) to below 1\(ms\). Similarly, message latency in the presence of multiprogram-
ming has been reduced from a high of more than 30\(ms\) to below 700\(\mu s\), with more than 70\% of all the messages having a latency below 100\(\mu s\). For our benchmark suite, FCS has been able to reduce the multiprogramming overhead from a high of 92\%, for some applications, to below 22\% for all applications.

Load balancing and locality has been shown to be very effective as well. LADLE achieved up to 40\% reduction in the execution time of some applications, on top of the reduction by FCS. The overall reduction of all the features is as high as 59\% for some applications. Reduction is achieved mostly by reducing the otherwise-wasted spin-time, hence improving the throughput of the overall cluster.

### 8.2 Future Work

**Exploration of scalability of LADLE to larger clusters:** Our current implementation scales easily to 32 processors, with negligible overhead. One issue of interest is to research scaling ability of load balancing to larger clusters, with potentially thousands of nodes. The two major overheads that LADLE exhibits are the bandwidth consumption due to exchange of progress information, and the time it takes for a new load assignment to be calculated. Even though both of these overheads consumed a negligible amount of the available bandwidth and compute time on a cluster of 32 processors, both of these overheads are likely to grow with the size of the cluster. Furthermore, our network, Compaq’s Memory Channel, has a built-in broadcast capability in the switch. Evaluating the impact of exchange of load information in the absence of such capability particularly on large clusters, would be of great value.

**Process placement guidance for better throughput:** In our work, we have assumed an environment in which process placement was not under run-time control. Considerable work has been done in the area of process placement and migration [19, 20, 25, 32, 62, 71]. Providing guidance for initial process placement, or possibly coupling a process migration subsystem with LADLE, might prove useful by
either starting new processes on idle nodes, or possibly migrating processes to idle nodes. We believe LADLE and such a process placement policy could complement each other. Earlier studies have shown initial process placement is almost always beneficial [20, 25, 32, 62], while the usefulness of process migration is largely dependent on correct modeling of process arrival time and lifetime. The challenge of exploring this field is to find a metric to guide either process migration or initial job placement. Many former systems used load-index as a guide for either process migration or initial placement [17, 32, 91]. Earlier work on homogeneous clusters has shown periodic load-index broadcast is a useful tool for placement of compute intensive long-lasting jobs [24]. However, not only is load-index not suitable for heterogeneous clusters, but also more recent studies suggest that such broadcasts to guide job placement might actually be bad for performance, particularly when arriving jobs are short lived [69]. Nevertheless, developing better job placement policies to be coupled with an application-level dynamic load balancing system such as LADLE remains a topic to be explored.

Another issue of interest is matching resources needed by an application to those available in a heterogeneous cluster. Not all processes use the same amount of all resources. While it is more valuable to have large memory for memory intensive applications, some other applications might have a huge demand for network bandwidth. Exploration of job placement policies should take into account different process demands, and match available resources to those demanded by a process.
Bibliography


