CRAUL: Compiler and Run-Time Integration for Adaptation Under Load *

Sotiris Ioannidis, Umit Rencuzogullari, Robert Stets, and Sandhya Dwarkadas

Department of Computer Science
University of Rochester
Rochester, NY 14627–0226

{si,umit,stets,sandhya}@cs.rochester.edu
Phone: 716-275-5647; Fax: 716-461-2018

Abstract

Clusters of workstations provide a cost-effective, high performance parallel computing environment. These environments, however, are often shared by multiple users, or may consist of heterogeneous machines. As a result, parallel applications executing in these environments must operate despite unequal computational resources. For maximum performance, applications should automatically adapt execution to maximize use of the available resources. Ideally, this adaptation should be transparent to the application programmer. In this paper, we present CRAUL (Compiler and Run-Time Integration for Adaptation Under Load), a system that dynamically balances computational load in a parallel application. Our target run-time is software-based distributed shared memory (SDSM). SDSM is a good target for parallelizing compilers since it reduces compile-time complexity by providing data caching and other support for dynamic load balancing. CRAUL combines compile-time support to identify data access patterns with a run-time system that uses the access information to intelligently distribute the parallel workload in loop-based programs. The distribution is chosen according to the relative power of the processors and so as to minimize SDSM overhead and maximize locality. We have evaluated the resulting load distribution in the presence of different types of load — computational, computational and memory intensive, and network load. CRAUL performs within 5—23% of ideal in the presence of load, and is able to improve on naive compiler-based work distribution that does not take locality into account even in the absence of load.

Keywords: load balancing, software distributed shared memory, compiler, run-time, integration

1 Introduction

Clusters of workstations, whether uniprocessors or symmetric multiprocessors (SMPs), when connected by a high-performance network, are an attractive platform for parallel computing due to their low cost and high availability. The default programming paradigm that is supported in hardware is message passing across the nodes, and shared memory among processes within a node. Unfortunately, the message passing paradigm requires explicit communication management by the programmer or parallelizing compiler. In applications with dynamic access patterns, this communication management can be very complex. Since the latency of communication across nodes is much larger than within a node, careful work distribution is also required in order to minimize communication overhead. , on multiprogrammed platforms or on platforms with unequal resources, the most efficient workload and communication schedule can be impossible to predict statically.

*This work was supported in part by NSF grants CDA–9401142, CCR–9702466, and CCR–9705594; and an external research grant from Compaq.

An alternative programming paradigm is software-based distributed shared memory (SDSM). An SDSM protocol (e.g., [3, 17, 26]) provides the illusion of shared memory across a distributed collection of machines, providing a uniform and perhaps a more intuitive programming paradigm. A shared memory paradigm provides ease-of-use and additionally leverages an SMP workstation's available hardware coherence to handle sharing within the SMP. SDSM has been shown to be an effective target for a parallelizing compiler [6]. Since data caching and communication is implemented by the run-time system, compile-time complexity is reduced. To improve performance, previous work [7, 21] has integrated compile-time information within the run-time system. Access patterns from the compiler are used by the run-time system to optimize communication, providing a significant improvement in performance.

Workstation clusters are typically shared by many users and are often possibly heterogeneous. Hence, balancing load to accommodate these variances is essential to obtain good performance. A parallel application should monitor the power of the computing resources and distribute the workload according to the observed power and the application's computation and communication demands. Ideally, this should be done without increasing application programming complexity.

In this paper, we present CRAUL (Compiler and Run-time Integration for Adaptation Under Load), a system that combines compile-time and run-time support to dynamically balance load in loop-based, SDSM programs. Information from the compiler on future accesses is fed to the run-time system at the points in the code that will be executed in parallel. Information on past accesses as well as estimates of the available computational and communication resources is available from the run-time. The run-time uses the past and future access pattern information along with estimates of the available computational and communication resources to make an intelligent decision on workload distribution. The decision balances the need not only to distribute work evenly, but also to avoid high communication costs in the underlying SDSM protocol.

Our techniques are applicable to any page-based software DSM [1]. In this paper, our target run-time system is Cashmere [26]. We implemented the necessary compiler extensions in the SUIF [2] compiler framework. Our experimental environment consists of DEC AlphaServer 2100 4/233 computers connected by a low-latency Memory Channel [11] remote-memory-write network.

We experiment with three different types of load — pure computational load, computational and memory-intensive load, and computational and network load. Our load balancing strategy provides a 23—81% improvement in performance compared to the execution time with a computational and memory-intensive load, and is able to adjust work distribution in order to reduce SDSM overheads. Performance is improved regardless of the type of load and is within 5—23% of ideal in the presence of load.

The rest of this paper is organized as follows. Section 2 describes the run-time system, the necessary compiler support, and the algorithm used to make dynamic load balancing decisions. Section 3 presents an evaluation of the load balancing support. Section 4 describes related work. Finally, we present our conclusions and discuss on-going work in Section 5.

2 Design and Implementation

We first provide some background on Cashmere [26], the run-time system we used in our implementation. We then describe the compiler support, followed by the run-time support necessary for load balancing.

[1] Preliminary results on a different run-time system, TreadMarks [3], are presented in [13].

2.1 The Base Software DSM Library

Our run-time system, Cashmere-2L (CSM) [26], is a page-based software DSM system that has been designed for SMP clusters connected via a low-latency remote-write network. The system implements a multiple-writer [4], "moderately" lazy release consistent protocol [15], and requires applications to adhere to the data-race-free, or properly-labeled, programming model [1]. Effectively, the application is required to use explicit synchronization to ensure that changes to shared data are visible. The moderately lazy characteristic of the consistency model is due to its implementation, which lies in between those of TreadMarks [3] and Munin [4]. Invalidations in CSM are sent during a release and take effect at the time of the next acquire, regardless of whether they are causally related to the acquired lock.

A unique point of the CSM design is that it targets low-latency remote-write networks, such as DEC's Memory Channel [11]. These networks allow processors in one node to directly modify the memory of another node safely from user space, with very low (microsecond) latency. CSM utilizes the remote-write capabilities to efficiently maintain internal protocol data structures. As an example, CSM uses the Memory Channel's remote-write, broadcast mechanism to maintain a replicated *directory* of sharing information for each page (i.e., each node maintains a complete copy of the directory). The per-page directory entries indicate who the current readers and writers of the page are.

Under CSM, every page of shared data has a single, distinguished home node that collects modifications at each release, and maintains up-to-date information on the page. Initially, shared pages are mapped only on their associated home nodes. Other nodes obtain copies of the pages through page faults, which trigger requests for an up-to-date copy of the page from the home node. Page faults due to write accesses are also used to keep track of data modified by each node, for later invalidation of other copies at the time of a release. If the home node is not actively writing the page, then the home node is *migrated* to the current writer by simply modifying the directory to point to the new home node. If there are readers or writers of a particular page on a node other than the home node, the home node downgrades its writing permissions to allow future possible migrations. As an optimization, however, we also move the page into *exclusive* mode if there are no other sharers, and avoid any consistency actions on the page. Writes on non-exclusive and non-home-node pages result in a *twin* (or pristine copy of the page) being created. The *twin* is later used to determine local modifications.

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

2.2 Compile-Time Support for Load Balancing

For the source-to-source translation from a sequential program to a parallel program using Cashmere, we use the Stanford University Intermediate Format (SUIF) [2] compiler. The SUIF system is organized as a set of compiler passes built on top of a kernel that defines the intermediate format. The passes are implemented as separate programs that typically perform a single analysis or transformation and then write the results out to a file. The files always use the same format.

The input to the compiler is a sequential version of the code. Standard SUIF can then generate a single-program, multiple-data (SPMD) program. We have added a SUIF pass that, among other things, transforms this SPMD program to run on Cashmere. Alternatively, the user can provide the SPMD program (instead of having the SUIF compiler generate it) by identifying the parallel loops in the program whose execution may be divided among the processes.

Our SUIF pass also extracts the shared data access patterns in each of the SPMD regions, and feeds this information to the run-time system. The pass is responsible for adding hooks in the parallelized code to allow the run-time library to change the load distribution in the parallel loops if necessary, given the information on the data access patterns.

2.2.1 Access pattern extraction

In order to generate access pattern summaries, our SUIF pass walks through the program looking for accesses to shared memory. A regular section [12] is then created for each such shared access. Regular section descriptors (RSDs) concisely represent the array accesses in a loop nest. The RSDs represent the accessed data as linear expressions of the loop indices along each dimension, and include stride information. This information is combined with the corresponding loop boundaries for that index, and the size of each dimension of the array, to determine the access pattern.

2.2.2 Load balancing interface and strategy

The run-time system needs a way of changing the amount of work assigned to each parallel task. This essentially means changing the number of (as well as which) loop iterations executed by each task. To accomplish this, the compiler augments the code with calls to the run-time library before the parallel loops. These calls are responsible for changing the loop bounds, and consequently, the amount of work performed by each task.

The compiler can direct the run-time to choose among partitioning strategies for distributing the parallel loops. Currently, a blocked distribution strategy is implemented. Load redistribution is effected by shifting the loop bounds of each processor, allowing us to handle blocked distributions efficiently. We change the upper and lower bounds of each parallel loop, so that tasks on lightly loaded processors will end up with more work than tasks on heavily loaded processors. Applications with nearest neighbor sharing will benefit from this scheme, since we avoid the creation of new boundaries, thereby avoiding the introduction of new communication due to increased sharing. Our goal is to expand the types of work distribution possible in the future in order to handle other initial distributions efficiently, as well as to take care of different sharing patterns.

2.3 Run-Time Load Balancing Support

As with any dynamic load balancing system, CRAUL bases its distributions fundamentally on an estimate of the available computational resources and communication overheads. Specifically, CRAUL uses a per-processor metric called Relative Processing Power (`RelativePower`) that captures the load induced by resource contention (processor, memory, network, and connecting busses) in a single value. `RelativePower` is used to directly determine a proposed distribution, however the proposal is only accepted if attendant SDSM overhead does not outweigh the distribution benefits. In the next two subsections, we describe the `RelativePower` and SDSM overhead calculations in more detail. We then provide a pseudo-code description of the entire run-time load balancing algorithm.

2.3.1 Relative Processing Power

`RelativePower` is maintained for each processor and is inversely proportional to the time the processor spent in its most recent parallel regions. We refer to the most recent parallel regions as the *region window*. Intuitively, a processor that completes its region window quickly will have a larger power relative to the other processors. Time is a good basis for `RelativePower` because it inherently captures the processor's perceived load, whether due to multiprogramming or contention for memory and/or the network. We track time over several parallel regions in order to smooth out transient spikes in performance.

Figure 1 shows the algorithm for calculating `RelativePower`. `TaskTime` holds the per-processor execution time of the last region window. This array is placed in shared memory and is updated as each processor completes a parallel region. In the above algorithm, `TaskTime` is first used to adjust the `RelativePower` and then the resulting values are normalized. This algorithm is executed by each processor. The `RelativePower` calculation is only performed at the end of a region window. This window is system-specific and determined through experimentation. It corresponds to the period of time required in order to avoid reacting to transient variations in resource availability.

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

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

// Normalize based on sum of the RelativePowers
for  all Processors i
    RelativePower[i] /= SumOfPowers;
```

Figure 1: Algorithm to determine relative processing power.

Our target class of applications are loop-based. CRAUL uses the `RelativePower` to directly partition the loop into units of work.

2.3.2 SDSM Overhead in Load Distribution

On hardware shared memory machines, the cost of re-distributing load can sometimes negate the benefits of load balancing [23]. Re-distribution on an SDSM platform will be even more expensive, leading to more overhead. Overhead on a page-based SDSM platform is also increased by *false sharing* – the concurrent, but otherwise independent, access by two or more processes to a single coherence unit. In Cashmere, the size of the coherence unit is a virtual memory page (8 KBytes on our experimental platform). CRAUL addresses both of these factors. First, CRAUL tailors its distributions to reduce false sharing. (As described in Section 3, CRAUL even provides improvements for unloaded applications by reducing false sharing.) Second, CRAUL accounts for SDSM overhead when determining the potential benefits of a redistribution.

False sharing that occurs at the boundaries of the work partitions can be eliminated in the case of regular accesses. Moreover, on an SMP-aware system, false sharing can only occur at partition boundaries that span two SMP nodes.

False sharing within a node occurs only at the cache line level and the penalties are low — hence, CRAUL does not attempt to eliminate cache line level false sharing. CRAUL uses the above knowledge of the underlying architecture to choose partition boundaries that do not introduce false sharing. If a partition as determined directly from `RelativePower` creates false sharing at the page level across SMP nodes, CRAUL will adjust the bound so that false sharing is eliminated after weighing the computational imbalance against the communication overhead incurred. As an optimization, when all work is to be performed by a single node, CRAUL assigns the work to the node that is the home for the largest chunk of the data, thereby respecting locality. Together, these steps improve performance by reducing the amount of data communicated.

In choosing a new distribution, CRAUL must ensure that re-distribution overhead does not negate the benefits of balancing the load. This is accomplished by incorporating an estimate of SDSM overhead with the impact of the balanced distribution. The SDSM overhead can be determined from the compiler-provided access patterns and information on the shared data cached at each node (available through the SDSM run-time).

CRAUL counts the number of pages that need to be transferred in the new distribution and multiplies the number by the average page transfer cost provided by the Cashmere run-time. The average is calculated over several previous transfers in order to capture current network load. This information is used to estimate the execution time for the upcoming region window, assuming the new distribution:

$$EstTaskTime_{new} = \frac{\sum_{i=0}^{i=nprocs-1} TaskTime_i}{nprocs} + SDSM \tag{1}$$

where $nprocs$ is the number of processors and $SDSM$ is the SDSM overhead calculated as described above. The first term on the right of the equation estimates the perfectly balanced execution time based on the execution time of the last region window. The second term then adds the SDSM overhead associated with re-distribution. Since the $TaskTime$ is a sum over the last several regions (the region window), the SDSM overhead is effectively amortized.

The estimated task time of the current distribution is simply taken to be the time of the slowest processor through the last region window:

$$EstTaskTime_{cur} = \max_{i=0..nprocs-1} TaskTime_i \tag{2}$$

If $EstTaskTime_{new}$ is less than $EstTaskTime_{cur}$, then CRAUL uses the new workload distribution.

2.3.3 Run-Time Load Balancing Algorithm

Figure 2 describes the run-time algorithm for executing a parallel loop. Steps 1–3 determine the load distribution and are described in the above subsections. Steps 4–7 execute and time the loop (we use the instruction cycle counter available on our platform). Step 8 controls the calculation of `RelativePower`, which drives the load distribution. The calculation is only performed at the end of a region window and only if CRAUL detects a load imbalance. Load imbalance exists if the fastest `TaskTime` is less than a certain threshold of the slowest `TaskTime`. The threshold is determined empirically (currently set to 10%).

2.3.4 Run-Time Load Balancing Summary

CRAUL bases its distribution strategy on a combination of available computational resources and expected re-distribution overhead. The availability of computational resources is modeled by a per-processor `RelativePower` metric.

```
1. Calculate loop bounds based on RelativePower.
2. Minimize SDSM communication.
2a.   Align partition boundaries to eliminate false sharing.
2b.   If work is limited to a single node then
2c.       assign computation to the data's home node

3. if there are new RelativePower values then
3b.    Verify that re-distribution costs do not negate balancing improvements.

4. Start timer.
5. Perform parallel loop computation.
6. Stop and store timer in TaskTime.
7. nRgn++          // increment region counter

8. if nRgn == Size of Region Window then
8a.    if load imbalance exists then
8b.        calculate new relative powers
8c.    nRgn=0  // Reset region counter
```

Figure 2: Pseudo-code description of a parallel loop execution.

This metric is inversely proportional to loop execution time and captures load due to contention for several different resources – processor, memory and even network. Re-distribution overhead is calculated by combining compiler-provided access patterns and dynamic information on SDSM data caching.

In choosing distributions, CRAUL also attempts to minimize SDSM communication. The system assigns work partitions to minimize false sharing and to locate computation on home nodes. A proposed distribution is based first on the `RelativePower` of processors in the system and second on the reduction of SDSM overhead.

A proposed distribution is accepted when the sum of the expected execution time and the SDSM overhead is less than the time for the slowest processor to complete the last parallel region. CRAUL also handles the special case where the expected computation time is less than SDSM overhead – all work is then performed on a single node.

2.4 Example

Consider the parallel loop in Figure 3. Our compiler pass transforms this loop into that in Figure 4.

The new code makes a **redistribute** call to the run-time library, providing it with all the necessary information to compute the access patterns (the arrays, the types of accesses, the upper and lower bounds of the loops, as well as their stride, and the format of the expressions for the indices).

The **redistribute** computes the relative powers of the processors (using the algorithm shown in Figure 1), and then uses the access pattern information to decide how to distribute the workload. It then creates the ranges of loop indices that each task has to access. Finally, the access pattern information can also be used to *prefetch* data [7] [2].

[2] The results presented do not exploit this feature, however.

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

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

Figure 3: Initial parallel loop. Shared data is indicated by `sh_`.

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

redistribute(
    list of shared arrays, /* sh_dat1, sh_dat2 */
    list of types of accesses,   /* read/write */
    list of lower bounds,       /* lower_bound */
    list of upper bounds,       /* upper_bound */
    list of strides,            /* strides     */
    list of coefficients and
    constants for indices        /* a, c, b, d */
);

    lowerbound = new lower bound for that range;
    upperbound = new upper bound for that range;

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

Figure 4: Parallel loop with added code that serves as an interface with the run-time library. The run-time system can then change the amount of work assigned to each parallel task.

3 Experimental Evaluation

3.1 Environment

Our experimental environment consists of four DEC AlphaServer 2100 4/233 computers. Each AlphaServer is equipped with four 21064A processors operating at 233 MHz and with 256 MB of shared memory, as well as a Memory Channel network interface. The Memory Channel is a PCI-based network, with a point-to-point bandwidth of approximately 33MBytes/sec. One-way latency for a 64-bit remote-write operation is 4.3 μsecs. The 21064A's primary data cache is 16 Kbytes, and the secondary cache size is 1Mbyte. Each AlphaServer runs Digital UNIX 4.0D with TruCluster v. 1.5 extensions. The programs, the run-time library, and Cashmere were compiled with **gcc** version 2.7.2.1 using the **-O2** optimization flag. In Cashmere, a page fetch operation takes 500 μs on an unloaded system, twin operations require 200 μs, and a diff operation ranges from 485–760 μs, depending on the size.

3.2 Results

We evaluate our system on four applications: Jacobi, Matrix Multiply (MM), Shallow, and Transitive-Closure (TC). Jacobi is an iterative method for solving partial differential equations, with nearest-neighbor averaging as the main computation. MM is a basic program that multiplies two matrices over several iterations. Shallow is the shallow water benchmark from the National Center for Atmospheric Research. The code is used in weather prediction and solves difference equations on a two-dimensional grid. TC computes the transitive closure of a directed graph. Table 1 provides the data set sizes and the uniprocessor execution times for each application.

Program	Problem Size	Sequential Time (sec.)
Jacobi	2048x2048	269.2
Matrix Multiply (MM)	256x256	398.6
Shallow	512x512	434.6
Transitive Closure (TC)	2048x2048	497.6

Table 1: Data set sizes and sequential execution time of applications.

The compiler passes transform each parallel loop in each of the applications in a manner similar to that shown in Figure 4. As stated in Section 2.2.2, the compiler currently directs the run-time system to choose a blocked distribution of the loops. The run-time system then dynamically adjusts the block boundaries according to perceived load while attempting to avoid any false sharing and optimizing for locality.

Figure 5 presents speedups for the four applications in the **absence** of load. The first (`Plain`) bar in each group is the performance of a compiler-based parallelization strategy without any run-time support. The second (`Balance`) shows speedups with the load balancing algorithm in place. The third bar (`Loc+Bal`) presents speedups using both the load balancing algorithm as well as communication minimization optimizations that avoid false sharing and attempt to perform computation on home nodes when beneficial. We present speedups on 4 (one SMP), 8 (4 processors on each of two SMPs), and 16 processors (4 processors on each of four SMPs).

Since these experiments were performed in the absence of load, they provide a measure of the perceived overhead of using our load-balancing scheme, as well as any benefits from the elimination of false sharing and scheduling based on locality. The benefits from using our communication minimization optimizations is most visible in Shallow. Shallow shows a 16% improvement in speedup with all CRAUL optimizations (`Loc+Bal`) in the absence of load, when compared to a direct parallelization by the compiler (`Plain`). The application has 13 shared 2-dimensional arrays. All the parallelized loops are partitioned in a blocked fashion. The application consists of a series of loops that operate either only on the interior elements of its matrices or on the boundary rows and columns. Compiler parallelized code or a naive implementation (`Plain`) would have each process update a part of the boundary rows and columns along each dimension in parallel. In the case of the rows, this can result in multiple processes writing the same pages since each row fits in half a page for our benchmark data set. This results in false sharing when work is distributed across nodes. Our run-time algorithm (`Loc+Bal`) is able to detect this false sharing and limits the distribution of the work in the parallel region to a single node. Further, the work is performed on the processor that currently owns and accesses the updated data. This effect can be seen in the reduction in the number of page transfers, and in the number of redistribution decisions (`Comm. Redists.`) made due to SDSM communication optimization in the noload `Loc+Bal` case (see Table 2).

Additionally, each loop iteration in Shallow accesses half a page for our data set (each row of 512 doubles spans half a page). While a balanced partitioning results in the data accessed by each processor being aligned on a page boundary, when load is redistributed, there is a possibility of additional false sharing if SDSM optimizations are not applied. This is seen by the reduction in the number of page transfers from the `Balance` to the `Loc+Bal` case

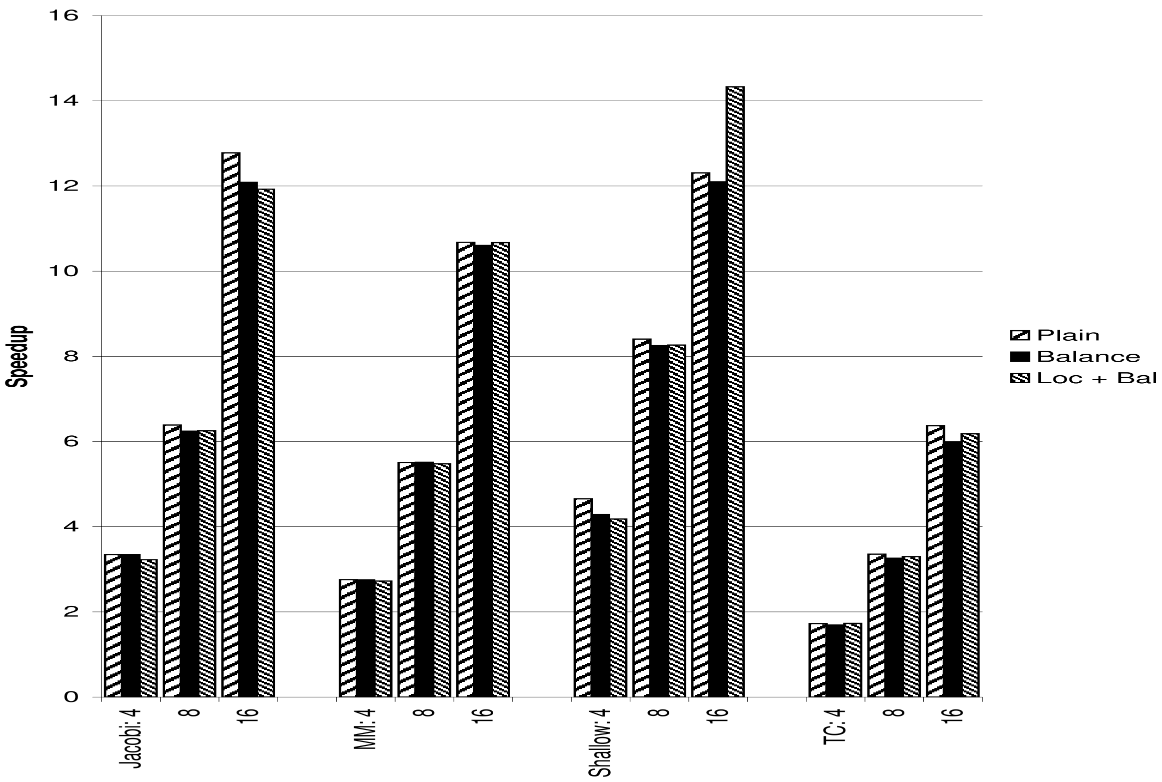

Figure 5: Speedups at 4, 8, and 16 processors in the absence of load.

(∼25K pages to ∼13K pages) in the presence of a computational load.

Figure 6 presents speedups for the same four applications in the **presence** of three different types of load — pure computation, memory and computation, and network and computation. The pure computational load consists of a program executing in a tight loop, and was used to create contention for the processor on which it executes. The memory and computational load consists of a sequential matrix multiply of two 512x512 matrices containing long integers. The program's working set is larger than our second-level cache and so introduces memory bus contention in addition to processor load. This load was used to determine the effect of contention for processor, memory, and bus resources. The network and computational load consists of a program that attempts to consume 8MB/s of the available bandwidth communicating with its peer on another node, in addition to contending for the processor on which it executes (note that this program will create some memory bus traffic as well in order to access the PCI bus). In order to test the performance of our load balancing library, we introduced one of the above processes on one of the processors of each SMP. This load takes up 50% of the CPU time in each case, in addition to the memory and network utilization in the second and third load type.

Once again, for each type of load, the first (`Plain`) bar in each group is the performance of a compiler-based parallelization strategy without any run-time support. The second (`Balance`) shows speedups with the load balancing algorithm in place. The third bar (`Loc+Bal`) presents speedups using both the load balancing algorithm as well as communication minimization optimizations. Speedups are presented on 4 (one SMP), 8 (4 processors on each of two SMPs), and 16 processors (4 processors on each of four SMPs). Table 2 presents detailed statistics including total data transferred, number of page transfers, and the number of redistribution decisions made due to load and due to excess communication for each type of load in the case of `Loc+Bal`. The statistics for `Plain` in the absence of load are also presented as a reference.

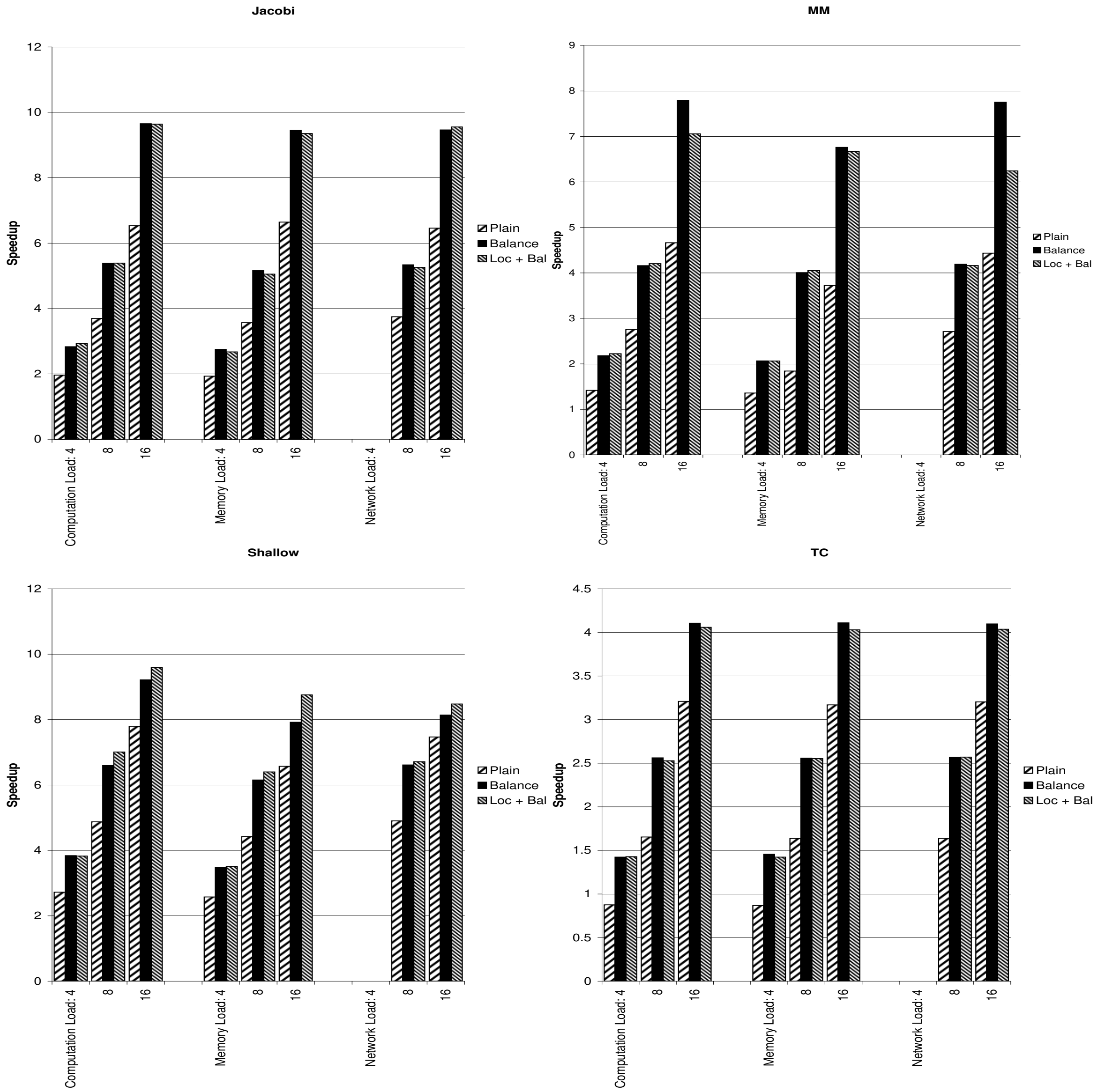

Figure 6: Speedups at 4, 8, and 16 processors with computational, memory, and network load.

The introduction of load slows down the application by as much as 92—188% in the case of 16 processors. Our load balancing strategy provides a 23—81% improvement in speedup at 16 processors compared to the `Plain` with load. In all cases, the loads that add memory and network contention result in a higher toll on performance in comparison to a pure computational load. In order to determine how good the results of our load balancing algorithm are, we compare the execution times obtained using 16 processors with load and our load balance scheme, with that using 14 processors without any load. This 14-processor run serves as a bound on how well we can perform with load balancing, since that is the best we can hope to achieve (four of our sixteen processors are loaded, and operate at only 50% of their power, giving us the rough equivalent of fourteen processors). The results are presented in Figure 7.

The 16-processor load balanced Shallow and Jacobi executions are respectively 10% and 18% slower than the 14-processor run. This difference is partly due to the fact that while computation can be redistributed, in both applications the communication per processor remains the same, which favors the 14-processor runs.

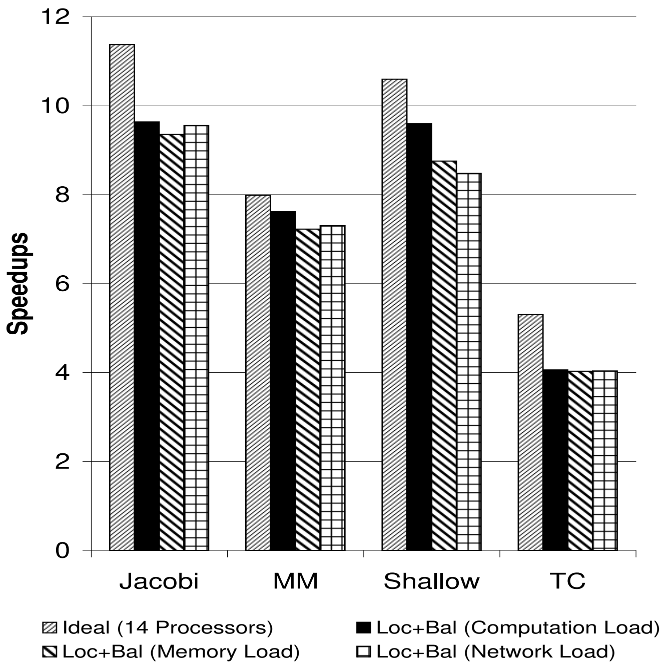

Figure 7: Comparison of the speedups of the applications using our load balancing algorithm on 16 loaded processors, compared to their performance on 14 load-free processors.

In Figure 8, we present a breakdown of the normalized execution time when adding a computational load with and without using CRAUL, relative to that on 16 processors with no load. Task and synchronization time represent application computation and time spent at synchronization points, respectively. Stall time measures the time required to obtain up-to-date data from remote nodes. Message time indicates the time spent handling protocol messages from remote nodes. Load balance time indicates the time spent in the code that implements the load redistribution. Protocol time covers the remaining time spent in the Cashmere library, consisting mainly of the time to propagate modifications.

Our load balancing algorithm reduces the time spent waiting at synchronization points relative to the execution time with load and no load balance because we have better distribution of work, and therefore improve overall performance. Task time also drops when load balancing is applied, although, as expected, not to the same level as the unloaded base case because of the computational load that adds to overall execution time (since we measure wall clock time). The drop in task time can be attributed to a reduction in the impact of the computational load on the execution time of the parallel application due to better load distribution. The time spent executing our load redistribution algorithm is between 1.3 and 16%. Shallow and TC spend a larger proportion of time in the load balancing code because of the smaller granularity of each parallel region. Load balancing almost always increases the amount of data transferred (see Table 2 - except for Shallow, where the communication optimizations help reduce the number of page transfers due to reduction in false sharing). However, overall performance is improved due to the reduction in synchronization and task time.

Program		Barriers	Data (MBytes)	Page Transfers	Load Redists.	Comm. Redists.
Jacobi	no load `Plain`	204	22.0	2662	0	0
	no load `Loc+Bal`	204	34.2	4125	6	0
	comp. load `Loc+Bal`	204	34.8	4209	12	0
	mem. load `Loc+Bal`	204	32.6	3938	6	0
	net. load `Loc+Bal`	204	37.2	4487	20	0
MM	no load `Plain`	104	2.4	288	0	0
	no load `Loc+Bal`	104	2.4	288	0	0
	comp. load `Loc+Bal`	104	6.5	777	10	0
	mem. load `Loc+Bal`	104	6.5	778	10	0
	net. load `Loc+Bal`	104	6.5	781	10	0
Shallow	no load `Plain`	904	112.1	13370	0	0
	no load `Loc+Bal`	904	103.5	12337	49	396
	comp. load `Loc+Bal`	904	109.7	13109	12	440
	mem. load `Loc+Bal`	904	106.6	12720	49	396
	net. load `Loc+Bal`	904	122.5	14673	12	440
TC	no load `Plain`	2052	64.1	7645	0	0
	no load `Loc+Bal`	2052	75.5	8869	5	0
	comp. load `Loc+Bal`	2052	78.1	9178	9	0
	mem. load `Loc+Bal`	2052	78.0	9166	10	0
	net. load `Loc+Bal`	2052	77.8	9138	7	0

Table 2: Application statistics covering number of barrier synchronizations, the amount of data transferred, the number of page transfers, number of work redistributions due to load, number of redistributions due to communication optimization, at 16 processors — in the absence of load for `Plain` and `Loc+Bal`, and in the presence of computational, memory, and network load for `Loc+Bal`.

4 Related Work

There have been several approaches to the problems of locality management and load balancing, especially in the context of loop scheduling. Perhaps the most common approach is the task queue model. In this scheme, there is a central queue of loop iterations. Once a processor has finished its assigned portion, more work is obtained from this queue. There are several variations, including *self-scheduling* [27], *fixed-size chunking* [16], *guided self-scheduling* [25], and *adaptive guided self-scheduling* [8]. These approaches tend to work well only for tightly coupled environments, and in general do not take locality and communication into account.

Markatos and LeBlanc [23] show that locality management is more important than load balancing for thread assignment in a thread-based scheduling system. They introduce a policy they call *Memory-Conscious Scheduling* that assigns threads to processors whose local memory holds most of the data the thread will access. Their results are simulation-based, and show that the looser the interconnection network, the more important the locality management. This work led to the introduction of *Affinity scheduling* [22], where loop iterations are scheduled over all the processors equally in local queues in a manner that maximizes the use of local memory. When a processor is idle, it removes *1/k* of the iterations in its local work queue and executes them. *k* is a parameter of their algorithm, which they define as *P*

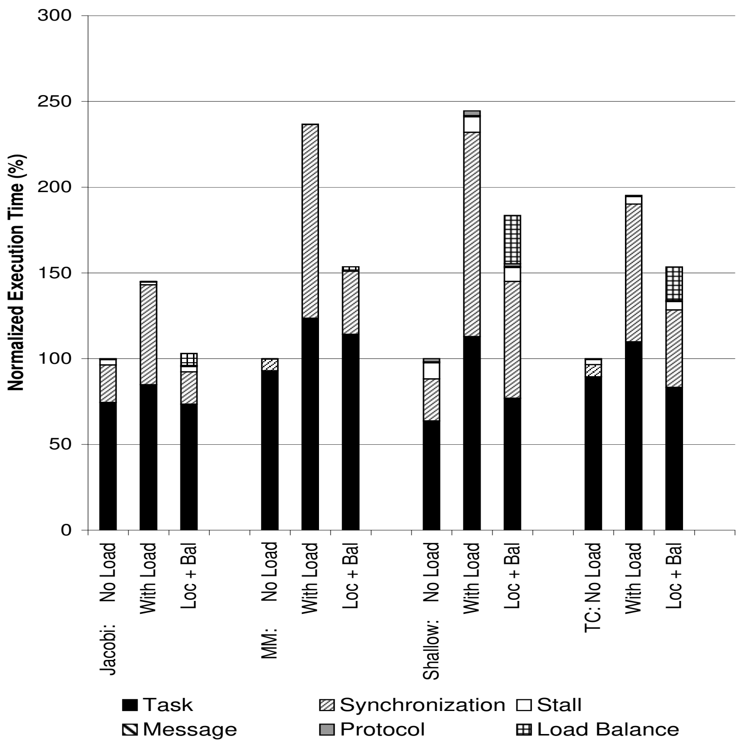

Figure 8: Normalized breakdown of execution time for base system with no load, with computational load, and the full CRAUL system running under computational load. Time is broken down into application computation (Task), wait at synchronization points (Synchronization), stall while waiting for up-to-date data to be obtained (Stall), handling of incoming messages (Messages), miscellaneous protocol functions such as the issuing of write notices (Protocol), and CRAUL load balancing overhead (Loc+Bal).

in most of their experiments, where *P* is the number of processors. If a processor's work queue is empty, it finds the most loaded processor and steals *1/k* of the iterations in that processor's work queue and executes them. Yan *et al.* [28] builds on affinity scheduling, by using *adaptive affinity scheduling*. Their algorithm is similar to affinity scheduling, but their run-time system can modify *k* during the execution of the program in order to change the chunks of work grabbed from loaded processors based on the relative processor load.

Cierniak *et al.* [5] study loop scheduling in heterogeneous environments with imbalances in the task time, processing power, and network bandwidth. While they do handle variations in the task time per loop that are statically determinable, they do not, however, address how dynamic changes in the underlying system are handled. Their results are also based on message-based applications. Moon and Saltz [24] examined applications with irregular access patterns. To compensate for load imbalance, they either re-map data periodically during pre-determined points in the execution, or at every time step.

In the context of dynamically changing environments, Edjlali *et al.* [9] and Kaddoura [14] present a run-time approach that checks to see if there is a need to redistribute work prior to each parallel section, and attempt to minimize communication. This is similar to our approach in CRAUL. However, their approach deals with message passing programs.

Zaki *et al.* [29] present an evaluation of global vs. local and distributed vs. centralized strategies for load balancing on distributed memory message-based systems. The strategies are labeled local or global based on the information they use to make load balancing decisions. Distributed and centralized refers to whether the decision-making is centralized at one master processor, or distributed among the processors. The authors argue that depending on the application and system parameters, different schemes can be the most suitable for best performance. Our approach assumes a distributed strategy with global information.

The system that seems most related to CRAUL is Adapt [20]. Adapt is implemented in concert with the Distributed Filaments [10] software DSM system. Adapt uses run-time information to extract access patterns by inspecting page faults. It can recognize and optimize for two access patterns: *nearest-neighbor* and *broadcast*. A cyclic distribution is used in the case of broadcast sharing with varying execution times, and a blocked distribution is used otherwise. In other work [19], Adapt is also used to optimize not only the current (local) parallel region, but also to attempt to ensure optimal distributions globally for other parallel regions as well. CRAUL uses both compile-time information on access patterns in the parallel region to be executed, as well as current run-time information on cached data, in order to balance load as well as to minimize communication due to false sharing and to maintain locality when beneficial.

Finally, systems like Condor [18] support transparent migration of processes from one workstation to another. However, they do not address load distribution in a single parallel application.

CRAUL deals with software distributed shared memory programs, in contrast to closely coupled shared memory or message passing. The load balancing mechanism targets both heterogeneous processors and processor, memory, or network load caused by competing programs. Furthermore, CRAUL minimizes communication and page sharing by taking the coherence granularity and current caching information into account.

5 Conclusions

In this paper, we address the problem of load balancing in SDSM systems to balance load in loop-based applications. SDSM has unique characteristics that are attractive: it offers the ease of programming of a shared memory model in a widely available workstation-based message passing environment, and allows dynamic caching of accessed data. However, multiple users and loosely connected processors challenge the performance of SDSM programs on such systems due to load imbalances and high communication latencies.

Our load-balancing system, CRAUL, combines information from the compiler and the run-time. It uses future access information available at compile-time, along with run-time information on cached data, to dynamically adjust load based on the available relative processing power and communication speeds. Performance tests on four applications and different types of load (which consume either memory, processor, or network resources) indicate that the performance with our load balancing strategy is within 5—23% of the ideal.

CRAUL is also able to optimize work partitioning even in the absence of load by taking advantage of caching information to avoid excess communication due to false sharing. The run-time system identifies regions where false sharing exists and determines if the resulting communication overhead would be larger than the computational imbalance from eliminating the false sharing. It then changes the work distribution by adjusting the loop boundaries to avoid the false sharing if beneficial, while respecting locality. Future work will examine extensions to the system in order to handle different types of work distributions and sharing patterns.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):67–76, December 1996.

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

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

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[5] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *Fourth International Symposium on High Performance Distributed Computing*, August 1995.

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

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

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

[9] G. Edjlali, G. Agrawal, A. Sussman, and J. Saltz. Data parallel programming in an adaptive environment. In *Internation Parallel Processing Symposium*, April 1995.

[10] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First USENIX Symposium on Operating System Design and Implementation*, pages 201–213, November 1994.

[11] Richard Gillett. Memory channel: An optimized cluster interconnect. *IEEE Micro*, 16(2):12–18, Feb 1996.

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

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

[14] M. Kaddoura. Load balancing for regular data-parallel applications on workstation network. In *Communication and Architecture Support for Network-Based Parallel Computing*, pages 173–183, February 1997.

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

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

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

[18] M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Usenix Winter Conference*, 1992.

[19] D. K. Lowenthal. Local and global data distribution in the filaments package. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, July 1998.

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

[21] H. Lu, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Software distributed shared memory support for irregular applications. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, pages 48–56, June 1996.

[22] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

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

[24] B. Moon and J. Saltz. Adaptive runtime support for direct simulation monte carlo methods on distributed memory architectures. In *Scalable High Performance Computing Conference*, May 1994.

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

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

[27] P. Tang and P. C. Yew. Processor self-scheduling: A practical scheduling scheme for parallel computers. In *1986 International Conference on Parallel Processing*, August 1986.

[28] Y. Yan, C. Jin, and X. Zhang. Adaptively scheduling parallel loops in distributed shared-memory systems. In *IEEE Transactions on Parallel and Distributed Systems*, volume 8, January 1997.

[29] Mohammed Javeed Zaki, Wei Li, and Srinivasan Parthasarathy. Customized dynamic load balancing for a network of workstations. *Journal of Parallel and Distributed Computing*, June 1997.