Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems *

Sotiris Ioannidis and Sandhya Dwarkadas
{si,sandhya}@cs.rochester.edu

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

Abstract. Networks of workstations offer inexpensive and highly available high performance computing environments. A critical issue for achieving good performance in any parallel system is load balancing, even more so in workstation environments where the machines might be shared among many users. In this paper, we present and evaluate a system that combines compiler and run-time support to achieve load balancing dynamically on software distributed shared memory programs. We use information provided by the compiler to help the run-time system distribute the work of the parallel loops, not only according to the relative power of the processors, but also in such a way as to minimize communication and page sharing.

1 Introduction

Clusters of workstations, whether uniprocessors or symmetric multiprocessors (SMPs), offer cost-effective and highly available parallel computing environments. Software distributed shared memory (SDSM) provides a shared memory abstraction on such distributed memory machines, with the advantage of ease-of-use. Previous work [5] has shown that an SDSM run-time can prove to be an effective target for a parallelizing compiler. The advantages of using an SDSM system include reduced complexity at compile-time, and the ability to combine compile-time and run-time information to achieve better performance ([6, 18]).

One issue in achieving good performance in any parallel system is load balancing. This issue is even more critical in a workstation environment where the machines might be shared among many users. In order to maximize performance based on available resources, the parallel system must not only optimally distribute the work according to the inherent computation and communication demands of the application, but also according to available computation and communication resources.

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

In this paper, we present and evaluate a system that combines compiler and run-time support to achieve load balancing dynamically on SDSM programs. The compiler provides access pattern information to the run-time at the points in the code that will be executed in parallel. The run-time uses these points to gather statistics on available computational and communication resources. Based on the access patterns across phases, as well as on available computing power, the run-time can then make intelligent decisions not only to distribute the computational load evenly, but also to maximize locality (based on current processor caching) and minimize communication overhead in the future. The result is a system that uniquely combines compile-time and run-time information to adapt both to changes in access patterns as well as to changes in computational power in order to reduce execution time.

Our target run-time system is TreadMarks [2], along with the extensions for prefetching and consistency/communication avoidance described in [6]. We implemented the necessary compiler extensions in the SUIF [1] compiler framework. Our experimental environment consists of eight DEC AlphaServer 2100 4/233 computers, each with four 21064A processors operating at 233 MHz. Preliminary results show that our system is able to adapt to changes in load, with performance within 20% of ideal.

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 some preliminary results. 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 TreadMarks [2], 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.

2.1 The Base Software DSM Library

TreadMarks [2] is an SDSM system built at Rice University. It is an efficient user-level SDSM system that runs on commonly available Unix platforms. TreadMarks provides parallel programming primitives similar to those used in hardware shared memory machines, namely, process creation, shared memory allocation, and lock and barrier synchronization. The system supports a *release consistent* (RC) memory model [10]. The programmer is required to use explicit synchronization to ensure that changes to shared data become visible. TreadMarks uses a *lazy invalidate* [14] version of RC and a *multiple-writer* protocol [3] to reduce the overhead involved in implementing the shared memory abstraction.

The underlying virtual memory hardware is used to detect accesses to shared memory. Consequently, the consistency unit is a virtual memory page. The *multiple-writer* protocol reduces the effects of false sharing with such a large consistency unit. With this protocol, two or more processors can simultaneously modify their own copy of a shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effects of false sharing. The merge is accomplished through the use of *diffs*. A diff is a run-length encoding of the modifications made to a page, generated by comparing the page to a copy saved prior to the modifications (called a *twin*).

With the *lazy invalidate* protocol, a process invalidates, at the time of an *acquire* synchronization operation [10], those pages for which it has received notice of modifications by other processors. On a subsequent page fault, the process fetches the *diffs* necessary to update its copy.

2.2 Compile-Time Support for Load Balancing

For the source-to-source translation from a sequential program to a parallel program using TreadMarks, we use the Stanford University Intermediate Format (SUIF) [11] 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. The output that we start with is a version of the code parallelized for shared address space machines. The compiler generates a single-program, multiple-data (SPMD) program that we modified to make calls to the TreadMarks run-time library. Alternatively, the user can provide the SPMD program (instead of having the SUIF compiler generate it) by identifying the parallel loops in the program that are executed by all processors.

Our SUIF pass extracts the shared data access patterns in each of the SPMD regions, and feeds this information to the run-time system. The pass is also 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.

Access pattern extraction

In order to generate access pattern summaries, our SUIF pass walks through the program looking for accesses to shared memory (identified using the **sh_** prefix). 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 upper and lower loop bounds along each dimension, and include stride information. This information is combined with the corresponding loop boundaries of that index, and the size of each dimension of the array, to determine the access pattern. Depending on the kind of data sharing among parallel tasks, we follow different strategies of load redistribution in case of imbalance. We will discuss these strategies further in Section 2.3.

Prefetching
The access pattern information can also be used to *prefetch* data [6]. The TreadMarks library offers prefetching calls. These calls, given a range of addresses, prefetch the data contained in the pages in that range, and provide appropriate (read/write) permissions on the page. This prefetching prevents faulting and consistency actions on uncached data that is guaranteed to be accessed in the future, as well as allows communication optimization by taking advantage of bulk transfer.

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 loop iterations performed by each task. To accomplish this, we augment the code with calls to the run-time library before the parallel loops. This call is responsible for changing the loop bounds and consequently the amount of work done by each task.

The compiler can direct the run-time to choose between two partitioning strategies for redistributing the parallel loops. The goal is to minimize execution time by considering both the communication and the computation components.

1. Shifting of loop boundaries: This approach changes the upper and lower bounds of each parallel task, so that tasks on lightly loaded processors will end up with more work than tasks on heavily loaded processors. With this scheme, we avoid the creation of new boundaries, and therefore possible sharing, on the data accessed by our tasks. Applications with nearest neighbor sharing will benefit from this scheme. This policy, however, has the drawback of causing more communication at the time of load redistribution, since data has to be moved among all neighboring tasks rather than only from the slow processor.
2. Multiple loop bounds: This scheme is aimed at minimizing unnecessary data movement. When using this policy, each process can execute non-continuous loop indices by using multiple loop bounds. This policy possibly fragments the shared data among the processors, but reduces communication at load redistribution time. Hence, care must be taken to ensure that this fragmentation does not result in either false sharing or excess true sharing due to load redistribution.

2.3 Run-Time Load Balancing Support

The run-time library is responsible for keeping track of the progress of each process. It collects statistics about the execution time of each parallel task and adjusts the load accordingly. The execution time for each parallel task is maintained on a per-processor basis (`TaskTime`). The relative processing power of the processor (`RelativePower`) is calculated on the basis of current load distribution (`RelativePower`) as well as the per-processor `TaskTime` as described in Figure 1.

```
float   RelativePower[NumOfProcessors];
float   TaskTime[NumOfProcessors];
float   SumOfPowers;


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

for  all Processors i
    RelativePower[i] /= SumOfPowers;
```

Fig. 1. Algorithm to Determine Relative Processing Power

Each processor executes the above code prior to each parallel loop (SPMD region). It is crucial not to try to adjust too quickly to changes in execution time because sudden changes in the distribution of the data might cause the system to oscillate. To make this clear, imagine a processor that for some reason is very slow the first time we gather statistics. If we adjust the load, we will end up sending most of its work to another processor. This will cause it to be very fast the second time around, resulting in a redistribution once again. For this reason, we have added some hysteresis in our system. We redistribute the load only if the relative power remains consistently at odds with current allocation through a certain number of task creation points. Similarly, load is balanced only if the variance in relative power exceeds a threshold. If the time of the slowest process is within N% of the time of the fastest process we don't change the distribution of work. Otherwise, minor oscillations may result as communication is generated due to the adjusted load. In our experiments, we collect statistics for 10 task creation points before trying to adjust, and then if the time of the slowest process is not within 10% of the time of the fastest process, we redistribute the work. These cut-offs were heuristically determined on the basis of our experimental platform, and are a function of the amount of computation and any extra communication.

Load Balancing vs. Locality Management
Previous work [20] has shown that locality management is at least as important as load balancing. This is even more the case in software DSM where the processors are not tightly coupled, making communication expensive. Consequently, we need to avoid unnecessary movement of data and at the same time minimize page sharing. In order to deal with this problem, the run-time library uses the information supplied by the compiler about what loop distribution strategy to use. In addition, it keeps track of accesses to the shared array as declared in previous SPMD regions (in other words, the data currently cached by each processor). Changes in partitioning that might result in excess communication are avoided in favor of a small amount of load imbalance (by comparing the time to

process the data locally against the time to communicate the data in order to effect a balanced computational load). We call this method *Locality-Conscious Load Balancing*.

2.4 Example

Consider the parallel loop in Figure 2. Our compiler pass transforms this loop into that in Figure 3. 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.

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

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

Fig. 2. Initial parallel loop.

3 Experimental Evaluation

3.1 Environment

Our experimental environment consists of eight 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. Each AlphaServer runs Digital UNIX 4.0D with TruCLuster v. 1.5 extensions. The programs, the run-time library, and TreadMarks were compiled with **gcc** version 2.7.2.1 using the **-O2** optimization flag.

3.2 Load Balancing Results

We evaluate our system on two applications: a Matrix Multiply of three 256x256 shared matrices of longs (which is repeated 100 times), and Jacobi, with a matrix size of 2050x2050 floats. Jacobi is an iterative method for solving partial differential equations, with nearest-neighbor averaging as the main computation.

Our current implementation only uses the first loop redistribution policy, shifting of loop boundaries, and does not use prefetching. To test the performance of our load balancing library, we introduced an artificial load on one of the

```
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 */
);

while (There are still ranges) {
    lowerbound = new lower bound for that range;
    upperbound = new upper bound for that range;
    range = range->next;

    for (i = lowerbound; i < upperbound; i += stride)
        sh_dat1[a*i + b] += sn_dat2[c*i + d];
}
```

Fig. 3. 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.

processors of each SMP. This consists of a tight loop that writes on an array of 10240 longs. This load takes up 50% of the CPU time.

Our preliminary results appear in Figures 4 and 5. We present execution times on 1, 2, 4, 8, and 16 processors, using up to four SMPs. We added one artificial load for every four processors except in the case of two processors where we only added one load. The load balancing scheme we use is the shifting of loop boundaries (we do not use multiple loop bounds). The first column shows the execution times for the cases where there was no load in the system. The second column shows the execution times with the artificial load, and finally the last column is the case where the system is loaded but we are using our load balancing library.

The introduction of load slows down both Matrix Multiply and Jacobi by as much as 100% in the case of two processors (with the overhead at 4, 8, and 16 not being far off).

Our load balancing strategy provides a significant improvement in performance compared to the execution time with load. In order to determine how good the results of our load balancing algorithm are, we compare the execution times obtained using 8 processors with load and our load balance scheme, with that using 7 processors without any load. This 7-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 (two of our eight processors are loaded, and operate at only 50%

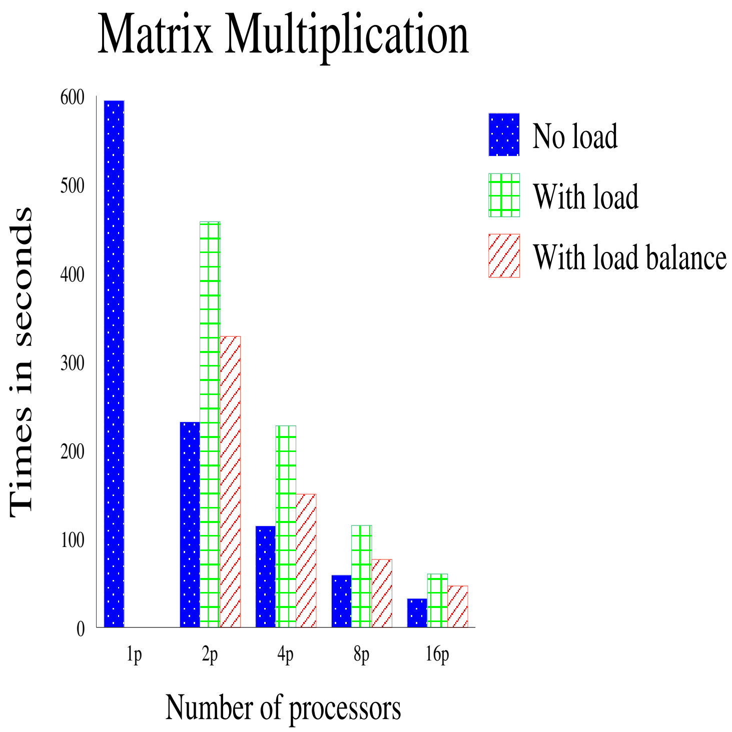

Fig. 4. Execution Times for Matrix Multiply

of their power, giving us the equivalent of seven processors). The results are presented in Figure 6. For Matrix Multiply, our load balancing algorithm is only 9% slower than the seven processor load-free case. Jacobi is 20% slower, partly due to the fact that while computation can be redistributed, communication per processor remains the same.

In Figure 7, we present a breakdown of the normalized execution time relative to that on 8 processors with no load, indicating the relative time spent in user code, in the protocol, and in communication and wait time (at synchronization points). When we use our load balancing algorithm, we reduce 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.

Finally, we conducted experiments to determine the overhead imposed by our run-time system. We ran Matrix Multiply and Jacobi in a load-free environment with and without use of our run-time library. The results are presented in Figure 8. In the worst case, we impose less than 6% overhead.

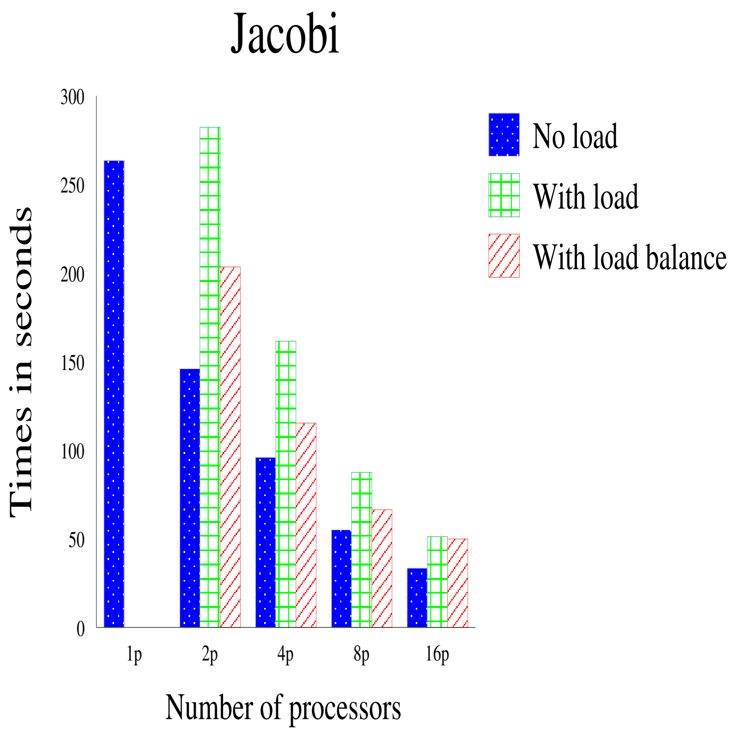

Fig. 5. Execution Times for Jacobi

3.3 Locality-conscious Load Balancing Results

For the evaluation of our locality-conscious load balancing policy, we used Shallow, with an input size resulting in matrices of 514x514 doubles. 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. Shallow operates on the interior elements of the arrays and then updates the boundaries. Compiler parallelized code or a naive implementation would have each process update a part of the boundaries along each dimension in parallel. This can result in multiple processes writing the same pages, or false sharing. A smarter approach is to have the processes that normally access the boundary pages do the updates, thus eliminating false sharing. Our integrated compiler/run-time system is able to make the decision at run-time, using the access pattern information provided by the compiler. It identifies which process caches the data and can repartition the work so that it maximizes locality.

We present our results in Figure 9. In these experiments, we don't introduce any load imbalance into our system, since we want to evaluate our locality-conscious load balancing policy. We have optimized the manual parallelization to eliminate false sharing, as suggested earlier. A naive compiler parallelization that

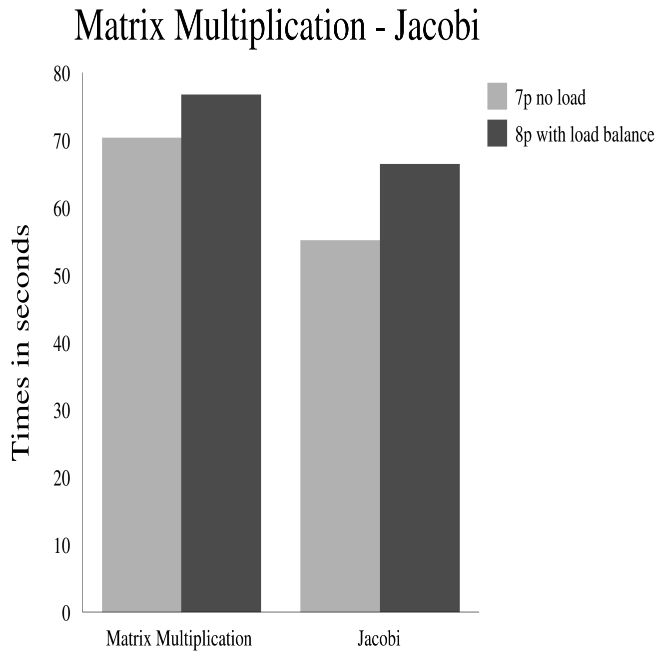

Fig. 6. Comparison of the running times of the applications using our load balancing algorithm on 8 loaded processors, compared to their performance on 7 load-free processors.

does not take data locality into account performs very poorly as the number of processors increases because of multiple writers on the boundary pages. However, when we combine the complier parallelization with our locality-conscious load balancing run-time system, the performance is equivalent to the hand-optimized code.

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* [23], *fixed-size chunking* [15], *guided self-scheduling* [22], and *adaptive guided self-scheduling* [7].

Markatos and Le Blanc in [20], argue that locality management is more important than load balancing in thread assignment. 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 show

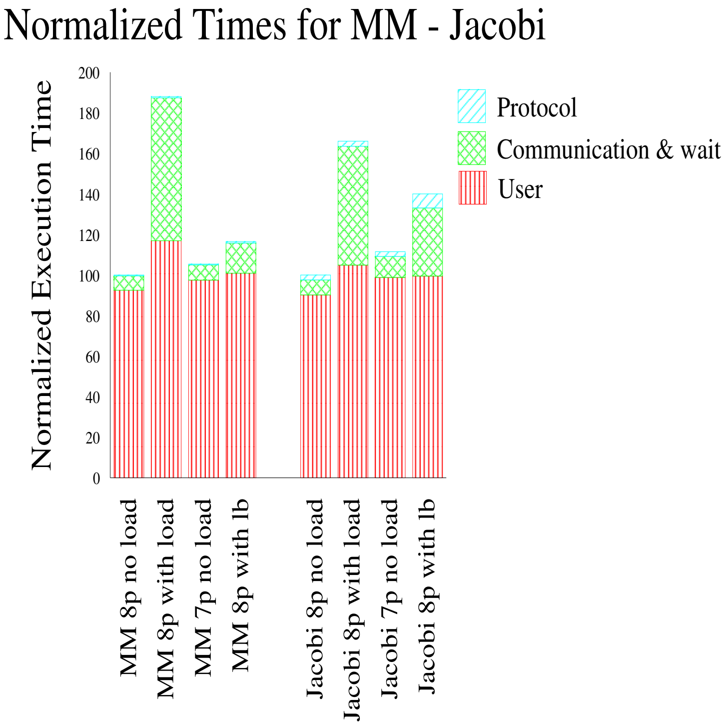

Fig. 7. Break-up of normalized time for Matrix Multiply and Jacobi into time spent in user code, communication and wait at synchronization points, and protocol time.

that the looser the interconnection network, the more important the locality management.

Based on the observation that the locality of the data that a loop accesses is very important, *affinity scheduling* was introduced in [19]. The loop iterations are divided over all the processors equally in local queues. 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 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 removes *1/P* of the iterations in that processor's work queue and executes them.

Building on [19], Yan et al. in [24] suggest *adaptive affinity scheduling*. Their algorithm is similar to affinity scheduling but their run-time system can modify k during the execution of the program. When a processor is loaded, k is increased so that other processors with a lighter load can get loop iterations from the loaded processor's local work queue. They present four possible policies for changing k: an exponential adaptive mechanism, a linear adaptive mechanism, a conservative adaptive mechanism, and a greedy adaptive mechanism.

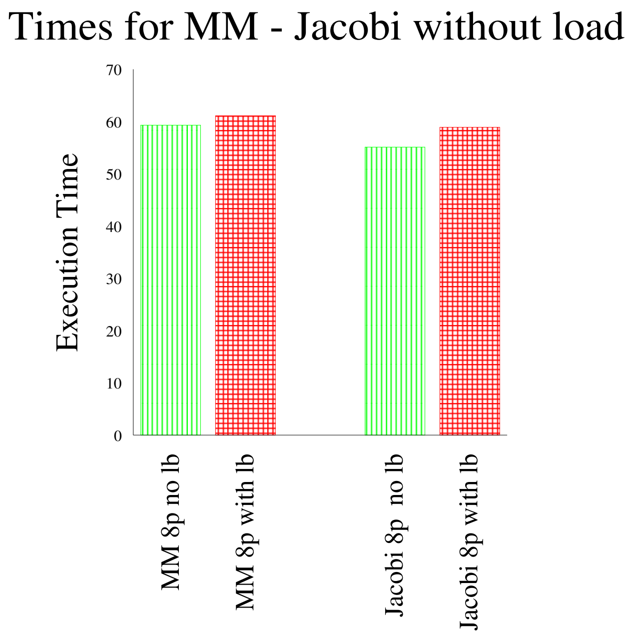

Fig. 8. Running times for Matrix Multiply and Jacobi in a load-free environment, with and without use of our run-time library.

In [4], Cierniak et al. study loop scheduling in heterogeneous environments with respect to programs, processors, and the interconnection networks. Their results indicate that taking into account the relative computation power as well as any heterogeneity in the loop format while doing the loop distribution improves the overall performance of the application. Similarly, Moon and Saltz [21] also looked at applications with irregular access patterns. To compensate for load imbalance, they examine periodic re-mapping, or re-mapping at predetermined points of the execution, and dynamic re-mapping, in which they determine if repartitioning is required at every time step.

In the context of dynamically changing environments, Edjlali et al. in [8] and Kaddoura in [13] present a run-time approach for handling such environments. Before each parallel section of the program they check if there is a need to remap the loop. This is similar to our approach. However, their approach deals with message passing programs.

A discussion on global vs. local and distributed vs. centralized strategies for load balancing is presented in [25]. 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 load balancer is 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.

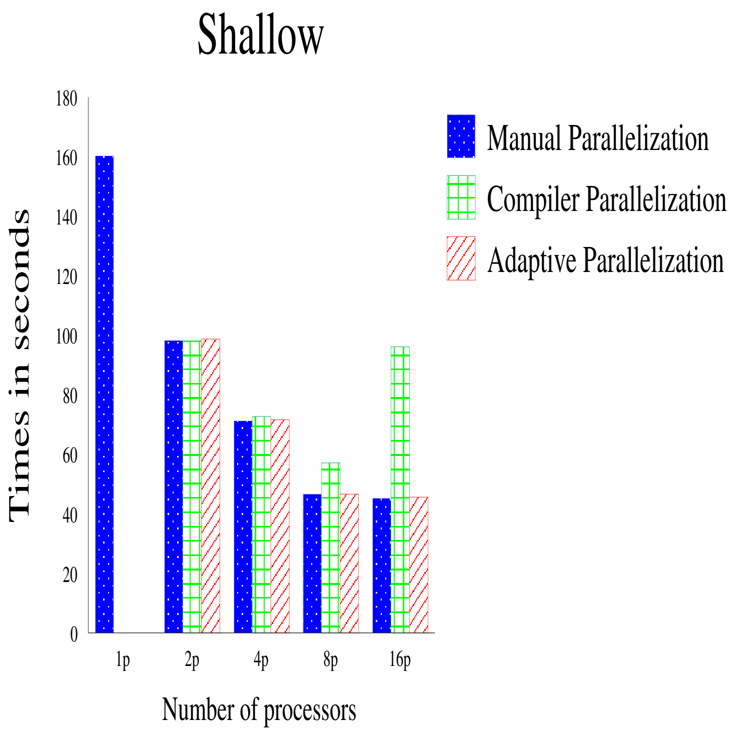

Fig. 9. Running times of the three different implementations of Shallow in seconds. The manual parallelization takes data location into account in order to avoid page sharing. The compiler parallelization does not consider data location. The adaptive parallelization uses the compiler parallelization with our run-time library, which adjusts the workload by dynamically taking the data location into account.

The system that seems most related to ours is Adapt, presented in [17]. Adapt is implemented in concert with the Distributed Filaments software kernel [9], a DSM system. It monitors communication and page faults, and dynamically modifies loop boundaries so that the processes access data that are local if possible. Adapt is able to extract access patterns by inspecting page faults. It can recognize two patterns: *nearest-neighbor* and *broadcast*. In our system, we use the compiler to extract the access patterns and provide them to the run-time system, making our approach capable of handling dynamic phase changes.

Finally, systems like Condor [16] support transparent migration of processes from one workstation to another. However, such systems don't support parallel programs efficiently.

Our system deals with software distributed shared memory programs, in contrast to closely coupled shared memory or message passing. Our load balancing method targets both irregularities of the loops as well as possible heterogeneous processors and load caused by competing programs. Furthermore, our system addresses locality management by trying to minimize communication and page sharing.

5 Conclusions

In this paper, we address the problem of load balancing in SDSM systems by coupling compile-time and run-time information. 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. 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 integrated system uses access information available at compile-time, along with run-time information, to dynamically adjust load at run-time based on the available relative processing power and communication speeds. The same access pattern information is also used to prefetch data. Preliminary results are encouraging. Performance tests on two applications and a fixed load indicate that the performance with load balance is within 9 and 20% of the ideal performance. Additionally, our system is able to partition the work so that processes attempt to limit their accesses to local data, thereby minimizing false sharing. Our system identifies regions where false sharing exists and changes the loop boundaries to avoid it. The performance on our third application, when the number of processors was high, was equivalent to the best possible workload partitioning.

Further work to collect results on a larger number of applications is necessary. In addition, for a more thorough evaluation, we need to determine the sensitivity of our strategy to dynamic changes in load, as well as to changes in the hysteresis factor used when determining when to redistribute work. The tradeoff between locality management and load must also be further investigated.

References

1. S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. The SUIF compiler for scalable parallel machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, February 1995.
2. C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.
3. J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.
4. M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *Fourth International Symposium on High Performance Distributed Computing*, August 1995.

5. A.L. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proceedings of the 11th International Parallel Processing Symposium*, pages 474–482, April 1997.
6. 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.
7. D. L. Eage and J. Zahorjan. Adaptive guided self-scheduling. Technical Report 92-01-01, Department of Computer Science, University of Washington, January 1992.
8. G. Edjlali, G. Agrawal, A. Sussman, and J. Saltz. Data parallel programming in an adaptive environment. In *Internation Parallel Processing Symposium*, April 1995.
9. 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.
10. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.
11. The SUIF Group. An overview of the suif compiler system.
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. 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.
14. P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.
15. C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. In *Transactions on Computer Systems*, October 1985.
16. M. Litzkow and M. Solomon. Supporting checkpointing and process migration outside the unix kernel. In *Usenix Winter Conference*, 1992.
17. D. K. Lowenthal and G. R. Andrews. An adaptive approach to data placement. In *10th International Parallel Processing Symposium*, April 1996.
18. 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.
19. E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEETPDS*, 5(4):379–400, April 1994.
20. E. P. Markatos and T. J. LeBlanc. Load balancing versus locality management in shared-memory mult iprocessors. *PROC of the 1992 ICPP*, pages I:258–267, August 1992.
21. B. Moon and J. Saltz. Adaptive runtime support for direct simulation monte carlo methods on distributed memory architectures. In *Salable High Performance Computing Comference*, May 1994.

22. C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. In *Transactions on Computers*, September 1992.
23. P. Tang and P. C. Yew. Processor self-scheduling: A practical scheduling scheme for parallel computers. In *International Conference On Parallel Processing*, Augoust 1986.
24. Y. Yan, C. Jin, and X. Zhang. Adaptively scheduling parallel loops in distributed shared-memory systems. In *Transactions on parallel and sitributed systems*, volume 8, January 1997.
25. M. J. Zaki, W. Li, and S. Parthasarathy. Customized dynamic load balancing for a network of workstations. Technical Report 602, Department of Computer Science, University of Rochester, December 1995.