Transparent Adaptive Parallelism on NOWs using OpenMP

Alex Scherer[1], Honghui Lu[2], Thomas Gross[1,3], and Willy Zwaenepoel[2]

[1]Departement Informatik
ETH Zürich
CH 8092 Zürich

[2]Department of Computer Science
Rice University
Houston, TX 77251

[3]School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

1 Introduction

Networks of workstations (NOWs) have repeatedly been suggested as computational engines [3]. In such an environment, however, individual nodes become available or unavailable as the workstation owner goes away or returns. To be truly useful, a parallel processing system for a NOW must be able to adapt to a continually changing pool of available nodes. Ideally, this adaptation should be transparent, allowing the user to program in a relatively standard way, without requiring any special-purpose code in the application. Such an adaptive parallel processing system is also useful in other environments, but in this paper we focus on a NOW, because there adaptivity is a requirement, not just an added feature.

Recent parallel programming models like HPF [13] or OpenMP [20] shelter the user from having to deal with some aspects of parallel programming, such as the number of nodes, the low-level details of iteration or data partitioning, or the communication of data between nodes. These properties simplify parallel programming, but, in addition, they also provide the foundation for *transparent* adaptive parallelism. Since aspects like the number of nodes are handled by the system and not the user, it becomes possible to change them adaptively without user intervention.

We focus in this paper on the emerging OpenMP standard [20]. In an OpenMP program, the programmer, roughly speaking, specifies what pieces of the code can be run in parallel. The number of processes executing these parallel constructs need not be hardwired. Therefore, adjusting the number of processes at runtime can be done transparently. Furthermore, OpenMP's execution model, consisting of a succession of sequential code and parallel constructs, naturally suggests efficient *adaptation points* at the beginning or the end of these parallel constructs.

One of the main technical challenges in supporting adaptivity on a NOW is to transparently move application data around at the time of adaptation, either moving it to newly joining nodes or moving it off leaving nodes. We rely on a software distributed shared memory (DSM) runtime system [15] to automate this process and to avoid any need for user intervention. Automatic data distribution is the main advantage of such systems, regardless of adaptivity. Here, that same feature is used to support automatic data re-distribution after an adaptation has taken place.

Our adaptive system is an extension of the TreadMarks DSM system [2], and uses the SUIF compiler toolkit [1] to generate TreadMarks code from OpenMP programs [17]. We demonstrate the performance of our system for different rates of adaptation and compare the results with non-adaptive runs of the same applications on the non-adaptive base TreadMarks system. We analyze the factors contributing to the cost of node joins and leaves. We conclude that for moderate rates of adaptation, the cost of adaptation is well within acceptable range and is a cost well worth paying for the added flexibility and functionality.

This paper then presents the following contributions:

1. The design of a transparent adaptive parallel computation system using an emerging industry-standard

programming paradigm (OpenMP). No code is added to the application specifically to obtain adaptivity;

2. Experimental evidence that the system provides good performance on a moderate-sized NOW and for moderate rates of adaptation.

2 Background

OpenMP uses the fork-join model of parallel execution. An OpenMP program begins execution as a single process, called the *master process* [1]. When the master process enters a *parallel construct*, it forks a *team* of t processes (one of them being the master process), and work is continued in parallel among these processes. Upon exiting the parallel construct, the processes in the team synchronize (join the master), and only the master continues execution (see Figure 1). A program may fork and join in this way any number of times. Each OpenMP parallel construct is thus executed using t processes, with the opportunity to change the number t at every new fork. The degree of parallelism need only be constant during the execution of one parallel construct [20].

```
-- this code is executed sequentially and only by the master --
#pragma OMP for
for(i=0; i<MAX; i++) {
-- the iterations of this loop are divided among all processes --
}
#pragma OMP end for
-- this code is executed sequentially and only by the master --
```

Figure 1: Pseudo-code for OpenMP C parallel for construct.

OpenMP is designed for a shared memory environment. To run OpenMP programs on a NOW, we compile OpenMP to the TreadMarks distributed shared memory (DSM) system [2]. TreadMarks is a user-level software DSM system that runs on commonly available Unix systems and on Windows NT. TreadMarks provides multithreaded parallel programming primitives similar to those used in hardware shared memory machines, namely, process creation, shared memory allocation, and lock and barrier synchronization.

To support OpenMP-like environments, recent versions of TreadMarks include `Tmk_wait`, `Tmk_fork` and `Tmk_join` primitives, specifically tailored to the fork-join style of parallelism expected by OpenMP and most other shared memory compilers [1]. `Tmk_wait` causes the slaves to wait for the next `Tmk_fork` issued by the master. In the master, `Tmk_wait` has no effect. `Tmk_fork` is a one-to-all synchronization: the master process causes all waiting slave processes to start executing. `Tmk_join` is the converse all-to-one synchronization: all processes, including both the master and the slaves, need to execute `Tmk_join` before the master can continue. The slaves return to the `Tmk_wait` state after they execute `Tmk_join`.

Compiling an OpenMP C program to TreadMarks is fully automated. The compiler is based on the SUIF pre-processor [1]. The body of each parallel loop is encapsulated into a new procedure. In the master, the loop is replaced by a call to `Tmk_fork` with as argument a reference to the procedure embodying the parallel loop. Additional code generated inside this procedure lets each process figure out, based on its TreadMarks process identifier and the total number of processes, which iterations of the loop it should execute. The procedure terminates with a call to `Tmk_Join`, which causes a return to the waiting state in the slaves. When

[1]The OpenMP document uses the term *thread*. In our distributed implementation of OpenMP, these threads execute as Unix processes on different nodes. We therefore consistently use the term *process*.

all processes have executed `Tmk_Join`, the master proceeds with any further sequential code and/or calls `Tmk_fork` to execute the next parallel loop.

3 Transparent adaptation

We present the added functionality for transparent adaptation in this section; the implementation is discussed in Section 4.

Each node normally executes one process. When a new node becomes available and starts participating in a computation, this is called a *join event*. When a node withdraws, we speak of a *leave event*. An *adapt event* is either a join or a leave event.

A request for an adapt event may occur anytime, but it is usually only executed at the next *adaptation point*. OpenMP provides natural adaptation points at the beginning or the end of the execution of a parallel construct. At these points, joins and leaves can be handled efficiently by increasing or decreasing the number of processes and re-partitioning the loop iterations among them.

Join events have the nice property that the system can always delay its response. If the event arrives while the system is not at an adaptation point, the system simply ignores the availability of an extra node until the computation reaches the next adaptation point. We call this behavior a (normal) *join*; Figure 2.a depicts such a situation.

Leave events are more complicated to handle since presumably the workstation node is needed for some other, higher-priority task. For a leave event, if the computation can reach the next adaptation point within a specifiable time limit, termed the *grace period*, we let the leave events take effect there. In this case, handling the leave event is greatly simplified since it is processed at a time when the system is free to determine the number of processes. We call this a *normal leave*. Figure 2.b depicts this case: a leaving process reaches an adaptation point within the grace period and is terminated at the adaptation point.

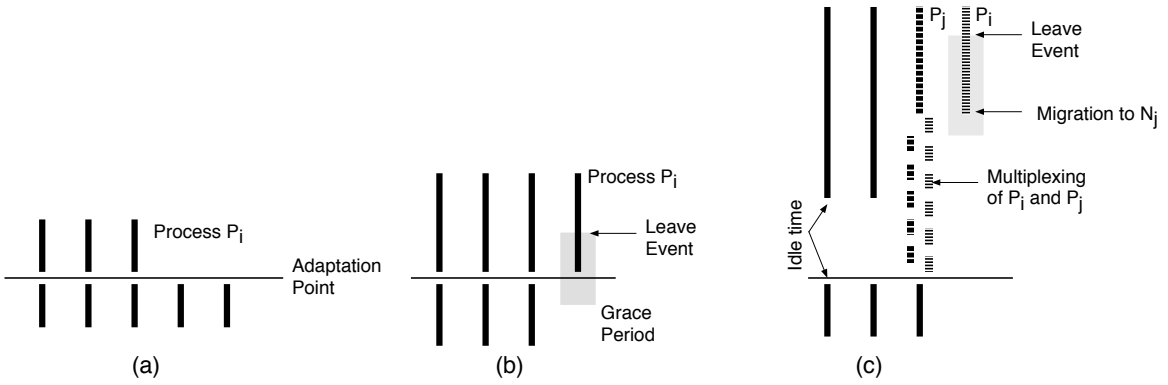

Figure 2: (Normal) join (a), normal leave (b), and urgent leave (c).

If the computation does not reach an adaptation point within the grace period, then the current process is migrated to another node in the system. The process is then executed on that node by multiplexing until the next adaptation point is reached. At that time, processing proceeds as in the case of a normal leave. We call this sequence an *urgent leave*, it is depicted in Figure 2.c. Urgent leaves cause much more data to be moved than normal leaves, because all intermediate data of the migrating process needs to be transferred. In addition, if a computation is balanced for t processes, multiplexing one node may idle the $t - 2$ non-multiplexed nodes for some time.

Since adaptation points are reached fairly frequently (in many applications several adaptation points are reached per second), urgent leaves are typically not needed. The concept of a grace period allows fine tuning the management of the workstation nodes, since this period can be node-specific (and may even vary during

a day). The owner of a workstation node is not denied service during the grace period; however, the node is shared with the computation that occupies a node.

Finally, the system also provides checkpointing to recover from catastrophic failures such as a crash, power flicker, or a machine reboot.

4 Implementation of the adaptive system

In this section we elaborate on how to run applications adaptively and how we have added support for adaptivity to the standard TreadMarks system. These changes are purely TreadMarks-internal, without *any* changes to the standard TreadMarks API or the operating system, so we can run standard TreadMarks programs, and we can convert OpenMP programs *automatically* to TreadMarks programs, using the same compiler we used for the non-adaptive version [17].

Each node recognizes join and leave events and communicates those to the master. How these events are generated is beyond the scope of this paper. E.g., a daemon may generate events at set times according to an operational schedule, or a load sensor may be employed to make load-dependent decisions.

4.1 Join events

The master spawns a new process on the specified host. While all processes continue normally, the new process asynchronously sets up network connections first to all other slave processes, then to the master. Therefore, when the master receives this connection request, it knows that the new process has set up all its other connections and is ready to join the computation.

A central idea of our implementation is to use the *garbage collection* mechanism from the non-adaptive TreadMarks system to simplify the adaptation. During execution, both the non-adaptive and the adaptive version of TreadMarks accumulate a variety of consistency information, primarily twins, diffs, and write notices [2]. When the memory allocated for these data structures becomes exhausted, TreadMarks initiates a garbage collection. This step removes all these internal data structures, and leaves each memory page either valid and up-to-date, or invalid but with its "owner field" pointing to a node with a valid copy of the page.

When all current processes have arrived at the adaptation point, the master initiates a garbage collection. This step updates the shared memory state and removes the memory consistency information, so much less data needs to be propagated to the joining node. In particular, it suffices for the master to send the joining process a message describing where an up-to-date copy of every shared memory page is located and what protocol is used (single or multiple writer).

The process identifiers are (re)assigned, and the total number of processes is reset (as applicable). Then the master sends the next `Tmk_fork` message to the new set of processes, and each process determines a (new) iteration partitioning based on its process identifier and the total number of processes, using the code generated by the OpenMP compiler.

Such a re-partitioning of the iteration space typically causes a re-distribution of the data. This re-distribution, if any, happens during further execution of the application, as a result of processes fetching pages on a page fault, using the normal DSM mechanisms.

4.2 Leave events

The handling of normal leave events is similar to the handling of join events. When all processes have reached the adaptation point, the master initiates a garbage collection, as for a join. Here too, the garbage collection leads to a substantial simplification of the adaptation. As a result, it suffices for the master to fetch all pages exclusively owned by the leaving process and invalid on the master, and to send a message to all other processes that it is now the owner of those pages. Finally, the master sends the `Tmk_fork` message,

as for a join.

Urgent leave events

When a process needs to migrate to another host, a new process is first created on that host, and the interprocess communication connections are set up to this process. All processes then wait for the completion of the migration. We rely on a modified version of the `libckpt` library to implement migration [21]. `Libckpt` is designed for checkpointing to disk and for recovery from that checkpoint, but we have modified it to write out the heap and the stack of the leaving process to the newly created process, and then start that process.

4.3 Fault tolerance

We use checkpointing for fault tolerance and include a brief description of it here because fault tolerance is a natural complement to adaptivity.

Whereas a distributed computation normally requires a consistent checkpoint [6] or some form of message logging [12] to guarantee correct recovery, we can avoid much of this complication by limiting checkpoints to the OpenMP adaptation points. At these points in the execution, the slave processes do not have any private "process" state (such as a stack) that needs to be recovered; they only have shared memory state. Only the master process has process state that needs to be recovered.

Checkpointing is therefore done periodically but only at an adaptation point. First, a garbage collection is invoked to bring shared memory into a well-defined state. Second, the master collects all pages for which it does not have a valid copy. Finally, the master uses the `libckpt` library to checkpoint itself to disk. No checkpointing by slave processes is required, avoiding the considerable complexity of checkpoint and recovery coordination.

4.4 Current limitations

The master node, which executes the master process, can migrate but it currently cannot perform a normal leave.

Adaptivity is limited to programs that repeatedly fork and join. Applications that fork once at the beginning and join once at the end run non-adaptively. We are adding support for adaptive execution of such applications in our next implementation.

It should also be understood that it is quite possible in OpenMP for the user to explicitly code the iteration partitioning in terms of the process identifiers and the number of processes. Clearly, adaptivity will not have any benefit for such applications or if the user explicitly inhibits adaptivity by setting the switch that OpenMP provides for this purpose.

5 Performance

5.1 Experimental environment

We use a switched, full-duplex 100Mbps Ethernet network, connecting 8 300Mhz Pentium II machines. Each machine has a 512K bytes secondary cache and 256M bytes of memory. The machines run FreeBSD 2.2.6, and use UDP sockets to communicate with each other. The roundtrip latency for a 1-byte message is 126 microseconds. The time to acquire a lock varies between 178 and 272 microseconds. The time for getting a diff varies between 313 and 1,544 microseconds, depending on the size of the diff. A full page transfer takes 1,308 microseconds.

	Size (shared memory)	Iterations	# Nodes	Time(seconds) Standard	Time(seconds) Adaptive	Number/amount of transfers Pages (4k)	MB	Messages	Diffs
Gauss	3072 x 3072	3072	8	243.46	242.14	80,577	320.54	236,453	0
	(48 MB)		4	398.07	397.23	41,463	164.62	129,021	0
			1	1,404.20	1,408.95	0	0	0	0
Jacobi	2500 x 2500	1000	8	215.06	216.17	58,041	254.50	221,631	27,993
	(47.8 MB)		4	361.38	362.88	30,741	131.17	115,840	11,994
			1	1,283.63	1,287.02	0	0	0	0
3D-FFT	128 x 64 x 64	100	8	83.50	81.95	198,471	779.23	416,570	0
	(42 MB)		4	138.20	133.51	170,115	667.16	354,018	0
			1	289.90	285.94	0	0	0	0
NBF	131072 atoms	100	8	535.89	534.74	353,056	1,388.27	1,182,292	0
	80 partners		4	714.78	715.36	183,600	721.85	618,443	0
	(52 MB)		1	2,398.79	2,299.20	0	0	0	0

Table 1: Execution times and network traffic on non-adaptive and adaptive system with no adapt events. Network traffic is identical on both systems.

	Average time per adaptation (seconds)							
Leaving process	Gauss		Jacobi		3D-FFT		NBF	
	8 proc.	6 proc.	8 proc.	6 proc.	8 proc.	6 proc.	8 proc.	6 proc.
end	4.19	4.60	2.77	3.78	1.87	2.50	1.01	2.81
middle	5.13	5.38	6.25	8.75	4.17	5.07	1.79	3.96

Table 2: Average cost of repeated adaptations between n and $n-1$ processes for $n = 8$ and $n = 6$.

5.2 Applications

We use a group of standard application kernels to assess the performance of our adaptive DSM system. Jacobi and Gauss are simple numerical codes. 3D-FFT comes from the standard NAS benchmark suite. It performs a 3-dimensional FFT transform using a sequence of 3 1-dimensional transforms, with a transposition of the matrix between the second and the third transform. NBF (Non-Bonded Force) is the kernel of a molecular dynamics programs. It simulates the force interactions between molecules. It is included as an example of an irregular application (i.e., an application in which the array indices are not linear expressions in the loop variables).

5.3 Key overall results

In the absence of adapt events, there is no cost to supporting adaptivity compared to the non-adaptive base system.

Table 1 shows the applications and the amount of shared memory used, and compares the runtimes on the non-adaptive TreadMarks 1.1.0 system and our adaptive system *without* any adaptations. The results demonstrate that in the absence of adapt events the overhead of the adaptive system is virtually nil. The slight performance gain of our system in some cases is coincidental. More importantly, the network traffic is identical in both systems.

Using a reasonable grace period (3 seconds), the system supports rates of adapt events of several adaptations per minute without significant performance degradation.

The average adaptation delay is calculated by comparing the measured runtime for the adaptive run with the computed time of a non-adaptive run for the same average number of nodes. Since the average number of nodes is always an integer in the non-adaptive case (but the average is a real number with adaptivity), we interpolate the results of the non-adaptive executions to obtain the reference execution time.

The cost of an adapt event varies, even for a single application. For instance, the process id of the leaving process may significantly affect the amount of data to be moved, as demonstrated by the schematic example

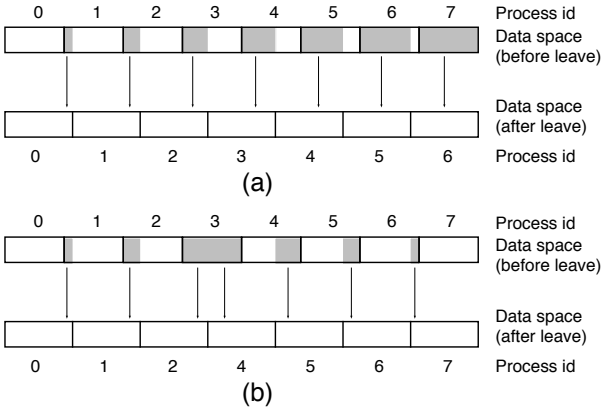

Figure 3: Effect of process id of leaving node: node 7 (a) and node 3 (b) require different data re-distribution (shaded). Up to 50% of the data space is moved for node 7, up to 30% for node 3.

in Figure 3. Many other factors affect the cost of joins and leaves, as documented in Section 5.4.

Table 2 provides an indication of the the range of the adaptation cost for each application. Leaves and joins are performed alternately, at most a single join or a single leave per adaptation point. Leaves are performed either by the "end" process, the one with the highest process id, or by the "middle" process with the id 4 or 3.

The main result is that the cost of an adaptation is typically on the order of 2-5 seconds for applications such as those in Table 1. We performed measurements ranging from 1 to 20 adapt events in one application run. Space limitations prohibit a detailed discussion, but the important results show that adaptation with 8 processes is always cheaper than with 6 processes, and even the worst case for any number of adaptations with 6 processes is below 10 seconds per adaptation.

The cost of adaptations is low enough so that a moderate rate of adaptations can be tolerated with reasonable performance. This cost has to be weighed against the flexibility of using additional nodes as they become available, or the ability to continue when a node withdraws.

The above measurements are performed with a sufficiently long grace period to ensure that all leaves are normal leaves. For the applications in Table 1, the average time between successive adaptation points is 0.1-0.2 seconds for Gauss, Jacobi and 3D-FFT, and about 2.5 seconds for NBF.

The cost of adaptation by migration alone is substantially higher.

We measured the cost of migrating a process from one node to another. These measurements address this "what-if" scenario: what is the overhead if all leaves are urgent leaves?

Two components determine the direct cost of migration: (i) the cost to create a new process on the new host (approximately 0.6 to 0.8 seconds), and (ii) the cost to move the processes image (at a rate of approx. 8.1 MByte/s). For Jacobi, this cost is about 6.7 seconds, for 3D-FFT 6.13 seconds, for Gauss 6.9 seconds, and for NBF 7.66 seconds. The total cost of an urgent leave is the sum of the migration cost above *plus* the cost of a normal leave (performed at the next adaptation point) *plus* the cost of multiplexing until the adaptation point is reached (see Figure 2.c).

The main benefit of adapting with normal leaves and joins is that the application can *change the number of processes* during its execution. That processing of the joins and normal leaves is a few seconds faster than the direct cost of migration is an additional advantage. Furthermore, if we only had migration, the execution time after the adaptation might well double after the adaptation; the multiplexing shown in Figure 2.c continues until the end or until another node becomes available.

5.4 Micro analysis of adaptation costs

To understand the cost of handling adapt events, we report some micro measurements. The advantage of a DSM (the flexibility provided to the programmer) also makes it difficult to obtain accurate data for the real cost of adaptation (the sum of the cost of maintaining the consistency information and the transfer cost for all pages). Unfortunately, simply counting the number of page fetches in both adaptive and non-adaptive runs does not provide a good indication of the overhead. For instance, when a page fetch occurs after an adaptation, it is difficult to distinguish whether this page fetch would have occurred even without the adaptation, or whether it was caused by the adaptation. In fact, the garbage collection and the relocation of data to the master on a leave may cause certain page fetches not to occur after adaptation, while they would have occurred in a non-adaptive execution.

We therefore use the following method to measure the overhead of single adapt events from m to n processes: We record statistics starting at the adaptation point where the adaptation occurs, and we subtract the results of a non-adaptive run of n processes with statistics-recording started at the same adaptation point. The difference reflects exactly the effects of the adaptation: In the adaptive run we begin our measurements with a data distribution for m processes, but the application does no more work on m processes before adapting, so the statistics show exactly the work performed on n processes plus the data re-distribution and timing.

(The full paper will have more detailed results to support the claims of this section - they are omitted here due to lack of space.)

Key cost component: the cost of adaptation is proportional to the maximum network traffic per link generated by the adapt event.

As we use a switched Ethernet, the network performance of individual links is independent of each other, so the link with the most traffic is the bottleneck.

An application's problem set size is typically proportional to the amount of shared memory used. Running the applications with smaller problem set sizes results in proportionally smaller adaptation costs.

The cost of adaptation decreases as the number of processes increases.

For a given application run, the total amount of data to be re-distributed is nearly constant for any adaptation from x to $x + n$ processes, regardless of x. Since almost the same amount of data is distributed over a larger number of links as x increases, the resulting network traffic is spread over more links with increasing x, and the maximum per link decreases significantly.

The cost per adaptation decreases as more processes join or leave at the same time.

All adapt event signals received between two successive adaptation points are handled at the next adaptation point, so the system may perform several joins or leaves or both simultaneously. Handling multiple adapt events together is much cheaper than adapting at successive adaptation points, because every additional concurrent join and/or leave costs less. As such, the numbers reported in Table 2 are a little pessimistic, since they always report on a single leave or a single join at an adaptation point.

The cost of adaptation decreases as more adaptations happen during the execution.

Many applications do not access all the shared memory they own in every iteration. At a leave, only the accessed pages need to be transferred, even if the leaving process' data partition is much larger. So the more recently a process had previously joined, the fewer data must be transferred.

Other factors

Furthermore, the process id reassignment algorithm, the ids of leaving/joining processes, the number of pages owned by leaving processes, the number of pages accessed after adaptation — all affect the cost of adaptation.

6 Related work

Various systems have been developed to allow sequential computations to use idle time on networked nodes. These systems include, among others, Butler [19], Condor [16], and many process migration systems such as, e.g. Sprite [7]. Our work distinguishes itself from these systems by its support for parallel computations.

Cilk-NOW [4], Dataparallel-C [18], Piranha [5], and various migration-based systems (e.g., Millipede [11] or versions of PVM [14]) support adaptive parallel computation on NOWs.

Blumofe and Lisiecki [4] describe the Cilk-NOW system for adaptive and reliable parallel execution of *functional* Cilk programs on networks of workstations. Programs need to be written in the Cilk language. Only functional Cilk programs are supported by Cilk-NOW. Nodes may join and leave an ongoing computation at any time, although, as with all systems, there is some "lag time" before the computation effectively leaves the node. Joining nodes "steal" work, under the form of a closure, from a randomly chosen node that is already participating in the computation. Leaving nodes return their unfinished closures to one of the remaining nodes. Various mechanisms make this work stealing efficient. Fault tolerance is achieved by a combination of transactional subcomputations and checkpointing. In contrast to Cilk-NOW, our system is not restricted to functional programs. Furthermore, although unmodified Cilk programs can be run on Cilk-NOW, our system has the advantage of supporting unmodified programs written in an industry standard form, based on a more general programming model. Finally, we take advantage of the inevitable "lag time" to attempt a more efficient form of adaptation.

Nedeljkovic and Quinn [18] describe a system for executing Dataparallel C [9] programs on NOWs. Dataparallel C contains data distribution statements, and programs are compiled to execute in parallel on *virtual* processors. Each virtual processor executes a C program augmented with communication statements inserted by the compiler. The runtime system then allocates some number of virtual processors to physical processors. To adapt to various load conditions, the number of virtual processors on a particular physical processor is adjusted. In addition, the data sections associated with a moving virtual processor must be moved as well. In the absence of an underlying shared memory platform, Dataparallel C is limited to programs for which the compiler can fully analyze the program to the point where it can generate communication statements, and discover exactly what data must be moved when a virtual processor moves. Furthermore, the use of virtual processors deprives the compiler of optimization opportunities. Our system avoids this limitation by using an underlying shared memory platform.

Piranha [5] supports adaptive parallelism for programs using the Linda tuple space. In Piranha, the programmer needs to provide three routines: a `feeder` routine to oversee the adaptive computation, a `piranha` routine to perform the actual computation, and a `retreat` routine to output to the tuple space whatever information is necessary for the computation to continue after a process leaves. Piranha processes may come and go at any time. Piranha requires the adoption of the tuple space as a parallel programming model, and, in addition, requires the programmer to write special code to achieve adaptivity (the `retreat` routine, and some modifications to the `piranha` routine for mutual exclusion). Instead, our system uses an industry standard programming model, and requires no modifications.

As compared to migration-based systems [11, 14], we use migration only if the grace period expires. We have documented the benefits of doing so, compared to the relatively small cost.

Fully automatic data management distinguishes our approach from systems such as Adaptive Multiblock PARTI [8], where the application programmer must add communication schedules for this purpose by hand. Also, this system requires a skeleton process to be left on a leaving node where our system can completely remove processes from a node.

Ioannidis and Dwarkadas use a DSM as a platform for load balancing [10]. To adjust the load (e.g., in response to competing use on a node), the iterations of a loop are partitioned based on a sophisticated strategy that tries to avoid rebalancing the computation too often. Their system explicitly deals with competing loads on a node but therefore does not handle the departure of a node.

7 Discussion and concluding remarks

We provide transparent adaptive parallel execution of OpenMP programs on NOWs. This system is a version of the TreadMarks DSM augmented to support adaptivity. The system works through iteration re-partitioning at the end of parallel loops – its efficient mode of operation – or through migration – a less efficient procedure only executed when necessary.

It is instructive to observe what properties of the resulting system are dependent on which design decisions. The use of OpenMP, besides being an industry-standard, allows transparent iteration re-partitioning, because the compiler generates the iteration-partitioning code such that it is executed at the beginning of each parallel construct. It is possible for the user to write TreadMarks code that does the same, but this is not the typical way TreadMarks programs are written (iteration partitioning is done once in many programs, as soon as the number of nodes becomes known). This result is not specific to OpenMP: the same techniques can be used for other contexts that do not fix the number of processes (nodes), e.g., when compiling the Fortran90 or HPF array statement[22].

The use of TreadMarks allows automatic distribution and communication of data, both during regular computation and after adaptation. Otherwise, the compiler would face the difficult task of generating communication code, or the user would have to write it. The use of a grace period allows the system to most often execute adaptations by iteration re-partitioning, with migration as a backup solution. This aspect of the system is not specific to TreadMarks, but could be achieved by any suitable DSM system.

We have described a prototype implementation and demonstrated that it exhibits good performance in a small NOW. There is no cost for the provision of adaptivity in the absence of adaptation. The performance penalty incurred for moderate rates of adaptations appears acceptable in the face of the augmented functionality.

Our current system delivers good performance but many opportunities for improvement are yet to be explored. Better process id reassignment strategies and a reduction of the bottleneck of transferring a leaving process' pages via the master process offer much room for improved performance in the future. The grace period also gives rise to a new use of compiler optimization that we have started to explore. In this paper, we equate adaptation points with the entry or exit into a concurrent construct, i.e., in many cases, the start or end of a parallel loop written by the user. However, the compiler can control the frequency of adaptation points by transformations similar to loop tiling or strip mining. Depending on the degree of flexibility required, the compiler can generate code that determines at runtime the trip counts or tiling of the loops, subject to the characteristics of the execution environment.

Shared memory on a NOW provides an attractive platform because it provides application-transparent access to a large memory. Transparent management of leaving and joining nodes allows computations to continue for a long period of time; they are no longer bounded by the time an individual workstation is present in the pool of compute servers. Both features, a large memory and a long lifetime, are essential to execute demanding applications on networks of commodity workstations.

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, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] T.E. Anderson, D.E. Culler, and D.A. Patterson. A case for NOWs. *IEEE Micro*, February 1995.

[4] R.D. Blumofe and P.A. Lisiecki. Adaptive and reliable parallel computing on network of workstations. In *Proceedings of the USENIX 1997 Annual Technical Symposium*, January 1997.

[5] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and piranha. *IEEE Computer*, 28(1), January 1995.

[6] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[7] F. Douglis and J. Ousterhout. Process migration in the sprite operating system. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 18–25, September 1987.

[8] G. Edjlali, G. Agrawal, A. Sussman, J. Humphries, and J. Saltz. Compiler and runtime support for programming in adaptive parallel environments. *Scientific Programming*, 6(2):215–227, Jan 1997.

[9] P. J. Hatcher and M. J. Quinn. *Data-parallel Programming on MIMD Computers*. The MIT Press, Cambridge MA, 1991.

[10] S. Ioannidis and S. Dwarkadas. Compiler and run-time support for adaptive load balancing in software distributed memory systems. In *Languages, Compilers, and Run-Time Systems for Scalable Computers (Proc. 4th Intl. Workshop LCR'98)*, pages 107–122, Pittsburgh, PA, May 1998. Springer Verlag.

[11] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its applications in distributed shared memory systems. *Journal of Systems and Software*, 1997.

[12] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[13] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[14] R. Konuru, S. Otto, and J. Walpole. A migratable user-level process package for pvm. *Journal of Parallel and Distributed Computing*, 40(1):–81–102, Jan 1997.

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

[16] M. Litzkow, M. Livny, and M. Mutka. Condor — a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[17] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on networks of workstations. In *Supercomputing '98*, November 1998.

[18] N. Nedeljkovic and M.J. Quinn. Data-parallel programming on a network of heterogeneous workstations. *Concurrency: Practice & Experience*, 5(4):257–268, June 1993.

[19] D.A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 5–12, November 1987.

[20] OpenMP Group. http:/www.openmp.org, 1997.

[21] J.S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the 1995 Winter Usenix Conference*, pages 213–223, January 1995.

[22] L. Wang, J. Stichnoth, and S. Chatterjee. Runtime performance of parallel array assignment: An empirical study. In *Proc. Supercomputing '96*, Pittsburgh, PA, November 1996. ACM/IEEE.