Transparent Adaptive Parallelism on NOWs using OpenMP

Alex Scherer[1], Honghui Lu[2], Thomas Gross[1,3], and Willy Zwaenepoel[4]

[1]Departement Informatik, ETH Zürich, CH 8092 Zürich
[2]Department of Electrical and Computer Engineering, Rice University, Houston, TX 77005
[3]School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213
[4]Department of Computer Science, Rice University, Houston, TX 77005

Abstract

We present a system that allows OpenMP programs to execute on a network of workstations with a variable number of nodes. The ability to adapt to a variable number of nodes allows a program to take advantage of additional nodes that become available after it starts execution, or to gracefully scale down when the number of available nodes is reduced. We demonstrate that the cost of adaptation is modest; the system allows a program to adapt at a moderate rate without much performance loss.

Two ideas underlie the efficiency of our design. First, we recognize that OpenMP programs exhibit convenient *adaptation points* during their execution, points at which the cost of adaptation can be much reduced. Second, by allowing a process a certain *grace period* before it must leave a node, we insure that most adaptations can occur at these adaptation points, and thus at low cost. Migration of a process, a much more expensive method for providing adaptivity, is used only as a back-up solution, when the process cannot reach an adaptation point within the grace period.

Our implementation consists of an OpenMP pre-processor that generates TreadMarks distributed shared memory (DSM) programs, and a version of TreadMarks modified to adapt to a variable number of nodes. Using a DSM as the underlying substrate facilitates the data (re-)distribution necessary after an adaptation.

Effort sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287, and by the National Science Foundation under awards CDA-9502791, CCR-9521735, and CDA-9626318, and by a grant from Tech-Sym Corporation. Part of this work was performed while Honghui Lu and Willy Zwaenepoel were visiting ETH Zürich. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

1 Introduction

Networks of workstations (NOWs) may be used as computational engines [3], but, unlike in dedicated computing environments, individual nodes in a NOW 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 (e.g., a parallel computer with a changing job mix), 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 [14] or OpenMP [21] 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 [21]. In an OpenMP program, the programmer specifies, roughly speaking, 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 sys-

tem [16] to avoid any need for user intervention in this task. 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 [18]. 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.

The outline of the rest of this paper is as follows. Section 2 provides some basic information about OpenMP and TreadMarks. Section 3 demonstrates how transparent adaptation is achieved. Section 4 describes the implementation of the system, and Section 5 assesses its performance. Related work is covered in Section 6, and Section 7 offers some discussion and concluding remarks.

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 [21].

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`, as well as `Tmk_join`; these primitives are specifically tailored to the fork–join, master–slave style of parallelism that is 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 [1] preprocessor. The body of each parallel loop (or, more generally, each parallel construct) 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 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

[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*.

```
-- this code is executed sequentially and only by the master --

#pragma omp parallel for
  for (i=0; i<MAX; i++) {
     -- the iterations of this loop are divided among all processes --
  }

-- this code is executed sequentially and only by the master --
```

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

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 task with a higher-priority. 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. For a normal leave, only the data that are still needed by other processes need to be moved.

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 it with the process already running on that node 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. The node is, however, shared with the computation that occupies a node.

Although all of the foregoing describes the joining or leaving of a single node, the system also supports multiple adaptations. In fact, one of the additional advantages of postponing adaptations is that many of them can be handled at once, leading to reduced overhead. 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 the support added to the standard TreadMarks system to implement adaptation. 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 [18].

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.

For simplicity, most of the following discussion focuses on the implementation of a single join event or a single leave event. The system is, however, capable of dealing with multiple join and leave events at a single adaptation point.

The principal implementation challenge is to make sure that after an adaptation memory continues to be shared consistently between the remaining processes. In other words, any shared memory information residing solely on leaving processes, whether shared memory data or shared memory state information, must be recorded in one or more of the remaining processes. Furthermore, any newly joining processes must receive the necessary shared memory state information so that they can locate the data that they need to access later.

To simplify this task, a central idea of our implementation is to use the *garbage collection* mechanism, already present in the non-adaptive TreadMarks system. 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, typically at a barrier or at a `Tmk_fork`. The garbage collection removes all these internal data structures, and leaves each mem-

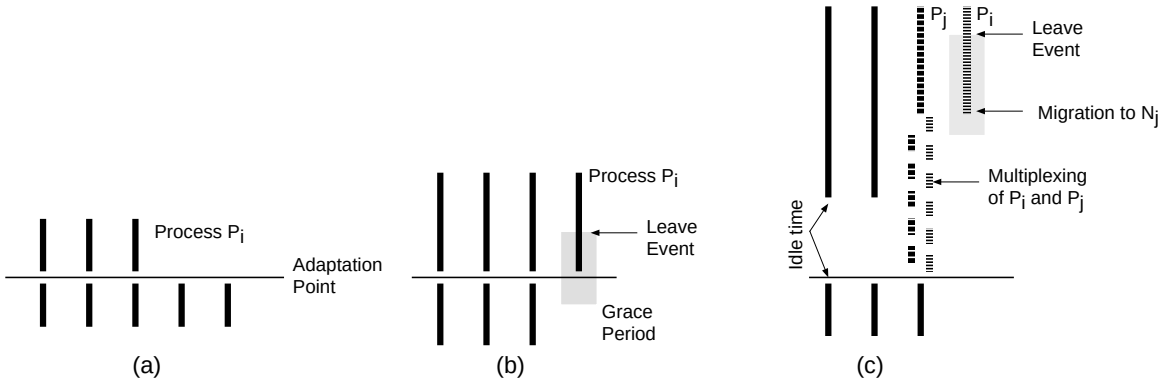

Figure 2: (Normal) join (a), normal leave (b), and urgent leave (c).

ory 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. By executing a garbage collection when an adaptation event takes place, we considerably reduce the complexity of adaptation and the amount of data that needs to be exchanged. In particular, no consistency data needs to be sent at the time of an adaptation.

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.

When all current processes have arrived at the adaptation point, the master initiates a garbage collection. Afterwards, the master sends the joining process a message describing where an up-to-date copy of every shared memory page is located.

The process identifiers are (re-)assigned, and the total number of processes is reset. 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 causes in many cases 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 Normal 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. It then suffices to move all pages exclusively valid at the leaving process elsewhere. To expedite this transfer, each of the remaining processes is sent a roughly equal number of these pages, allowing the data to be sent in parallel. Furthermore, several pages are sent in a single message, further reducing the cost. The master then informs all processes of the identity of the new owners of these pages. The remainder of the leave procedure is identical to that for a join, including reassignment of process identifiers, the `Tmk_for k` message sent by the master, and any data re-distribution.

As an additional optimization, if an equal number of processes join and leave the computation at the same adaptation point, the joining processes try to "take the place" of the leaving processes: the pages exclusively valid at a leaving process are sent immediately to a newly joining process.

4.3 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 [22]. `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.

4.4 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 [7], or some form of message logging [13], 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.

4.5 Current limitations

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. It is also possible for the user to explicitly disable 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 byte 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. We use TreadMarks versions 1.1.0 in our experiments. In this version, the time to acquire a lock varies between 178 and 272 microseconds. An 8-node barrier takes 359 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.

5.2 Applications

We use a group of standard application kernels to assess the performance of our adaptive DSM system. Jacobi and Modified Gramm-Schmidt (MGS) are simple numerical codes. 3D-FFT comes from the standard NAS benchmark suite [4]. 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. Water and Barnes are two applications from the Splash benchmark suite [23].

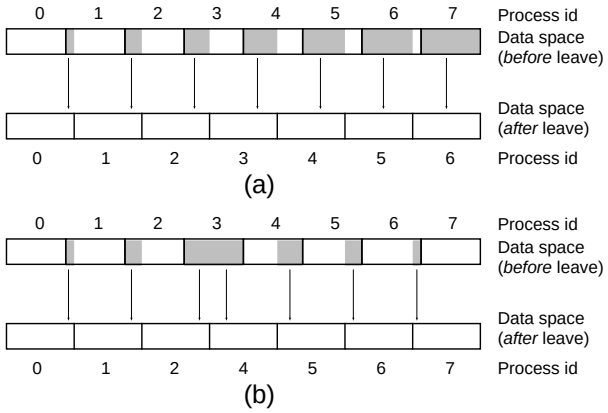

Figure 3: Effect of process id of leaving node on the amount of data moved at adaptation time, with a simple block distribution of the data among processors. The shaded areas indicate the data to be moved. Up to 50% of the data space is moved if node 7 leaves (a), only up to 30% if node 3 leaves.

5.3 Key overall results

In this section we discuss the cost of join and leave events. In the next section we analyze these costs based on detailed implementation measurements.

5.3.1 Overhead for providing adaptivity

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 execution times for the non-adaptive TreadMarks system and for 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.

5.3.2 Cost of normal joins and leaves

It is not straightforward to quantify the cost of adaptation. 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 in Figure 3. Many other factors affect the cost of joins and leaves, as documented in Section 5.4.

To provide an idea of the overhead of adaptation, we periodically cause an adapt event to occur during the execution of an application. We report results for a range of periods at which adapt events occur. Figure 4 shows the results for

	Size shared memory	Iterations	# Nodes	Time (seconds) Standard	Adaptive	Number/amount of transfers Pages (4k)	MB	Messages	Diffs
MGS	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
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
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
NBF	131072 atoms	100	8	535.89	534.74	353,056	1,388.27	1,182,292	0
	80 partners, 52MB		4	714.78	715.36	183,600	721.85	618,443	0
Barnes	32768 bodies	20	8	96.18	96.92	45,239	465.74	932,051	428,127
	6 MB		4	150.76	152.93	23,750	246.88	331,213	145,769
Water	1728 molecules	20	8	123.81	123.32	69,012	277.43	211,841	5,323
	1.5 MB		4	232.97	235.30	35,085	139.28	110,215	1,393

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

Application	Time
MGS	1.93
Jacobi	2.29
3D-FFT	1.30
NBF	1.32
Barnes	2.46
Water	0.08

Table 2: Normal leave times (in seconds).

Application	Time
MGS	2.30
Jacobi	1.73
3D-FFT	1.16
NBF	0.48
Barnes	1.08
Water	0.01

Table 3: Normal join times (in seconds).

each application. In these experiments, the grace period is made large enough such that all adapt events can occur at adaptation points. The application starts out with 4 or 8 processes. Alternately, at the end of a period, a process leaves, or a process joins. The leaving process is always the one with the highest process id, exhibiting a worst-case for many applications (see Figure 3). The x-axis shows the period at which adapt events occur. The y-axis shows the execution time. The figures demonstrate that, for these applications, the system can support up to several adaptations per minute with only modest performance loss.

Tables 2 and 3 provide a different view of the results. To obtain these results, a single adapt event was made to occur during the execution of an application. Again, in the case of a leave, the last process was made to leave. To produce a representative result, we performed this experiment a number of times, changing the time at which the adaptation occurred. The adaptation delay, presented in Tables 2 and 3 is calculated by subtracting the execution time of a non-adaptive execution starting at the adaptation point from the execution time of the adaptive execution starting at the same adaptation point. The main result is that the cost of an adaptation is typically on the order of 0.1 to 3 seconds for this set of applications.

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 MGS, Jacobi and 3D-FFT, 1.4 seconds for Water, 2.6 seconds for NBF, and about 4.4 seconds for Barnes.

We conclude that with reasonable values of the grace period, the system supports rates of adapt events of several adaptations per minute without significant performance degradation. In addition, the cost of adaptation has to be weighed against the flexibility of using additional nodes as they become available, or the ability to continue when a node withdraws.

5.3.3 Cost of migration

The cost of adaptation by migration alone is substantially higher than that of adaptation by normal leaves and joins.

We measured the cost of migrating a process from one node to another. 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 process image (at a rate of approx. 7.8 MByte/s). The results are reported in Table 4. We see that, with the exception of Barnes, where the costs are comparable when we include the cost to create a new process on the new host, the cost of migration is substantially higher than that of a normal leave.

The total cost of an urgent leave is the sum of the migration cost above *plus* the cost of a normal leave (performed at

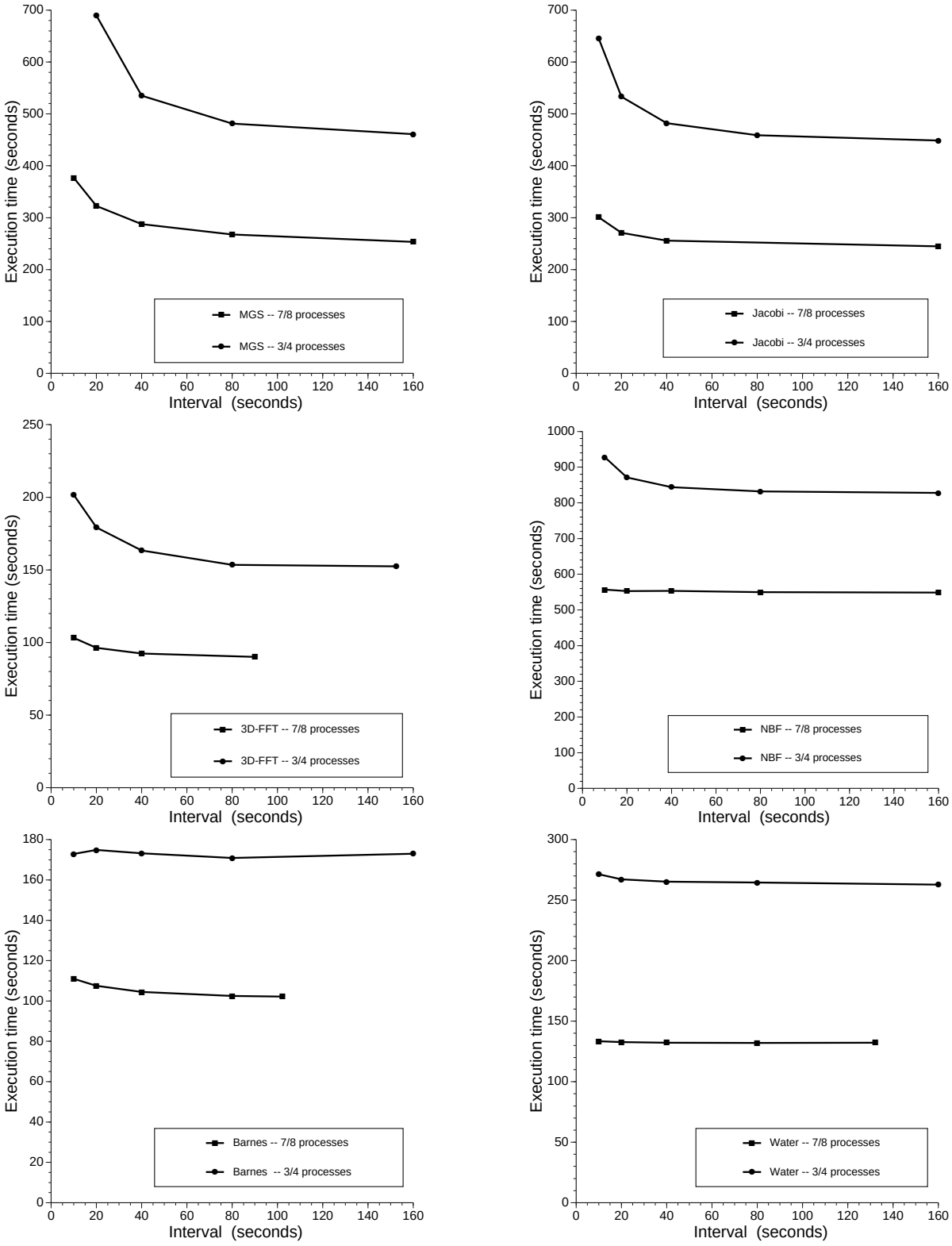

Figure 4: Execution times for different intervals between adapt events.

Application	Migration time
MGS	6.90
Jacobi	6.70
3D-FFT	6.13
NBF	7.66
Barnes	1.75
Water	0.93

Table 4: Migration cost (in seconds).

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 detailed execution statistics in Tables 5 through 7.

The advantage of a DSM – automatic data movement — also makes it difficult to obtain accurate statistics 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 on a leave may cause certain page fetches not to occur after adaptation, while they would have occurred in a non-adaptive execution. Furthermore, since we move several pages in a single message when handling a leave event, it is actually possible that a program runs faster with adaptation than without (see, e.g., Water in Table 5), if the leave event moved pages that would have been fetched in any case.

We therefore use the following method to measure the data movement associated with a single adaptation from m to n processes. We perform an adaptive run, and we start recording statistics at the adaptation point. Then, we perform a non-adaptive run with n processes with statistics-recording started at the same point during the execution as where the adaptation occurred in the adaptive run. The difference between the statistics recorded in these two experiments reflects exactly the effects of the adaptation. At the time statistics recording starts, the application has performed exactly the same amount of work in both experiments, albeit with m processors in the adaptive run and n in the non-adaptive run. The statistics recorded therefore show exactly the same amount of remaining work performed on n processes, plus the data movement as a result of adaptation in the case of the adaptive run.

For each of the applications, Tables 5 and 6 provide detailed statistics obtained using these experiments. In particular, Table 5 provides statistics for one process leaving, or two processes leaving at the same adaptation point, always starting with eight processes. In each case, a maximum and a minimum case are reported, depending on the process id of the leaving processes. The statistics reported are the difference in execution time, the number of pages moved when handling the leave event (Pages-System) and the extra number of pages that are fetched subsequently during execution of the application (Pages-Application). Table 6 shows statistics for one process joining and two processes joining at the same adaptation point, in each case ending up with a total of eight processes after adaptation. We also show the case of one process leaving and one process joining at the same adaptation point. Finally, Table 7 shows for a single application, 3D-FFT, the same statistics but for differing numbers of processes before and after adaptation.

5.4.1 Data transfer

The key component in handling an adaptation is additional network traffic. In almost all cases, the number of extra pages transfered by the application after the adaptation (labeled Pages-Application) dominates the number of pages transfered during the actual adaptation (Pages-System). This observation suggests that better process id re-assignment strategies, which reduce the amount of application data transfer, could have an important beneficial effect on adaptation times.

5.4.2 Size of the system

The cost of adaptation decreases as the number of processes increases. Table 7 shows the cost of handling leaves (top) and joins (bottom); the cost increases as the number of processes is reduced. For many applications, the total amount of data to be re-distributed for any adaptation from x to $x+n$ processes tends to grow as x decreases. Furthermore, fewer links are available to carry this traffic.

5.4.3 Batching joins and leaves

The cost per adaptation decreases as more processes join or leave at the same time. Looking at Table 5 and 6, we see that handling two events together is cheaper than handling two single events [2]. This observation applies when the two events are both leaves, or both joins, but it especially applies

[2] The exception is Water, where adaptations are very cheap to begin with. Also, such minor time differences may be due to the measurement setup.

	Minimum, one leave			Maximum, one leave			Minimum, two leaves			Maximum, two leaves		
	Time (sec)	Pages System	Pages Appl.	Time (sec)	Pages System	Pages Appl.	Time (sec)	Pages System	Pages Appl.	Time (sec)	Pages System	Pages Appl.
MGS	1.62	576	3955	1.93	576	3956	2.04	1152	3258	2.24	1152	3660
Jacobi	1.53	1526	3383	2.29	1526	5883	1.89	3046	4071	3.22	3050	8656
3D-FFT	0.93	672	2148	1.3	672	3036	1.07	1408	2012	1.68	1408	4882
NBF	1.14	1552	3239	1.32	1552	6082	0.96	3104	2229	2.23	3104	8911
Barnes	1.75	0	538	2.46	0	887	1.23	0	890	1.44	0	1386
Water	-0.41	34	-28	0.08	35	54	0.03	69	-20	0.21	72	113

Table 5: Costs for one leave ($8 \rightarrow 7$) and two leaves ($8 \rightarrow 6$).

Application	One process joining		Two processes joining		One leave and one join		
	Time (sec)	Pages Appl.	Time (sec)	Pages Appl.	Time (sec)	Pages System	Pages Appl.
MGS	2.30	4040	1.76	3460	1.00	768	2
Jacobi	1.73	6101	2.2	9158	1.42	2030	8
3D-FFT	1.16	3144	1.39	5088	0.8	850	271
NBF	0.48	6327	1.16	9557	1.57	2068	72
Barnes	1.08	958	1.27	1413	1.13	0	787
Water	0.01	-61	0.05	16	0.03	47	22

Table 6: Costs for one join ($7 \rightarrow 8$), two joins ($6 \rightarrow 8$) and a leave/join ($6 \rightarrow 6$).

where there is a leave and a join, because of the optimization applied in this case (see Section 4.2).

As a result, since the numbers reported in Section 5.3 always report on a single leave or a single join at an adaptation point, they must be considered somewhat pessimistic.

5.4.4 Other factors

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.

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 [20], Condor [17], and many process migration systems such as, e.g. Sprite [8]. Our work distinguishes itself from these systems by its support for parallel computations.

Cilk-NOW [5], Dataparallel-C [19], Piranha [6], and various migration-based systems (e.g., Millipede [12] or versions of PVM [15]) support adaptive parallel computation on NOWs.

Blumofe and Lisiecki [5] 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 [19] describe a system for executing Dataparallel C [10] 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

# Processes before & after adaptation			Time (sec)	Pages System	Pages Appl.
6	5	Min	1.28	1210	1776
6	5	Max	1.89	1210	3338
6	4	Min	1.33	2420	1755
6	4	Max	2.64	2420	4787
4	3	Min	2.03	1760	1868
4	3	Max	2.33	1760	2874
4	2	Min	2.51	3520	1504
4	2	Max	4.28	3520	3584
5	6		1.30	n/a	3585
4	6		1.62	n/a	5490
3	4		2.12	n/a	3473
2	4		2.95	n/a	5378

Table 7: 3D-FFT: Costs for leaves and joins.

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, Data-parallel 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 [6] supports adaptive parallelism for programs using the Linda tuple space. In Piranha, the programmer needs to provide three routines: a `fee der` 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 `retre at` 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 [12, 15], we use migration only if the grace period expires. We have documented the benefits of iteration re-partitioning over migration and demonstrated that iteration re-partitioning is possible to do most of the time.

Fully automatic data management distinguishes our approach from systems such as Adaptive Multiblock PARTI [9], where the application programmer must add communication schedules 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 [11]. 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 does not handle the departure of a node.

7 Discussion and concluding remarks

Our system provides transparent adaptive parallel execution of OpenMP programs on NOWs. This system is based on a version of the TreadMarks DSM augmented to support adaptivity. The system works through iteration re-partitioning at the beginning or 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 [24].

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.

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 re-assignment strategies offer much room for improved performance. 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. De-

pending 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.

Acknowledgements

We appreciate the comments, feedback, and assistance from Vikram Adve, Alan L. Cox, Charlie Y. Hu, Girija Narlikar, and Juan Navarro.

References

[1] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and C. W. Tseng. An Overview of the SUIF Compiler for Scalable Parallel Machines. In *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 662–667, San Francisco, 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 NOW (Networks of Workstations). *IEEE Micro*, 15(2):54–64, February 1995.

[4] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report TR RNR-91-002, NASA Ames, August 1991.

[5] 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*, pages 133–147, January 1997.

[6] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1):40–49, January 1995.

[7] 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.

[8] F. Douglis and J. Ousterhout. Process Migration in the Sprite Operating System. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 18–25, September 1987.

[9] 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.

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

[11] 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.

[12] A. Itzkovitz, A. Schuster, and L. Shalev. Thread Migration and its Applications in Distributed Shared Memory Systems. *Journal of Systems and Software*, 42(1):71–87, 1997.

[13] 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.

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

[15] 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.

[16] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[17] M. Litzkow, M. Livny, and M. Mutka. Condor — a Hunter of Idle Workstations. In *Proc. 8th Intl. Conf. Distributed Computing Systems*, pages 104–111, June 1988.

[18] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on Networks of Workstations. In *Proc. Supercomputing '98*, Orlando, FL, November 1998. ACM/IEEE.

[19] N. Nedeljkovic and M.J. Quinn. Data-parallel Programming on a Network of Heterogeneous Workstations. *Concurrency: Practice & Experience*, 5(4):257–268, June 1993.

[20] D.A. Nichols. Using Idle Workstations in a Shared Computing Environment. In *Proc. 10th ACM Symp. Operating Systems Principles*, pages 5–12, November 1987.

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

[22] 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.

[23] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical Report CSL-TR-91-469, Stanford University, April 1991.

[24] 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.