Adaptive Parallelism for OpenMP Task Parallel Programs

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

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

Abstract. We present a system that allows task parallel OpenMP programs to execute on a network of workstations (NOW) with a variable number of nodes. Such adaptivity, generally called *adaptive parallelism*, is important in a multi-user NOW environment, enabling the system to expand the computation onto idle nodes or withdraw from otherwise occupied nodes.
We focus on task parallel applications in this paper, but the system also lets data parallel applications run adaptively.
When an adaptation is requested, we let all processes complete their current tasks, then the system executes an extra OpenMP join-fork sequence not present in the application code. Here, the system can change the number of nodes without involving the application, as processes do not have a compute-relevant private process state.
We show that the costs of adaptations is low, and we explain why the costs are lower for task parallel applications than for data parallel applications.

1 Introduction

We present a system supporting adaptive parallelism for task parallel OpenMP programs running in a multi-user network of workstations environment, permitting the efficient use of a continually changing pool of available machines. As other users start and stop using machines, resources which otherwise would be idle are used productively, while these users retain priority.

Adaptive parallelism also allows for other flexible usage models: A certain percentage of machines may be reserved without having to specify which machines, or a second parallel application may be started without having to abort some on-going long-running program, simply by reducing this application's allocated resources and letting the new application use them.

We have described how we achieve transparent adaptive parallelism for data parallel programs in [15], therefore we focus more on task parallel applications in this paper.

We use the *OpenMP [14]* programming model, an emerging industry standard for shared memory programming. OpenMP frees the programmer from having to deal with lower-level issues such as the number of nodes, the data partitioning or the communication of data between nodes. We recognize that the system can easily adjust the number of compute threads at the boundaries of each OpenMP parallel construct *without* having to involve the application. We call such points *adaptation points.*

In task parallel applications, each process solves tasks retrieved from a central task queue, starting with the next assignment whenever one is done. Typically, each process works at its own pace, there is no global barrier-type synchronization, as opposed to data parallel applications. When an adaptation is requested, we let each process complete its current task, then the system creates an adaptation point by executing an *extra* OpenMP join-fork sequence not present in the application code, allowing for a transparent adaptation.

For the application, the only requirement to support adaptivity is for processes to indicate whenever they have completed a task, so the system has the opportunity to transparently insert an adaptation point. For this purpose, one system call is inserted at task boundaries in the application code. This minor code change is done automatically by a preprocessor.

We have tested the system using Quicksort and TSP as example applications running on various NOW environments of PCs with Linux. Even frequent adaptations, such as every 3 seconds, only increase Quicksort's and TSP's runtimes by about 10-20% and 2-5%, respectively.

This paper then presents the following contributions:

1. The design of a transparent adaptive parallel computation system for task parallel applications using an emerging industry-standard programming paradigm (OpenMP). Only one function call is added to the application specifically to obtain adaptivity. This change is done automatically by a preprocessor.
2. Experimental evidence that the system provides good performance on a moderate-sized NOW, even for frequent rates of adaptation.
3. An analysis of the key adaptation cost components, showing why adaptations in task parallel applications are generally cheaper than in data parallel applications.

2 Background

2.1 Related Work

Various approaches have been examined to use idle time on networked nodes for parallel computations.

Much work has been done to support variable resources by using *load balancing*: Systems such as DOME [3], Dataparallel-C [7, 13], Charm++ [8, 9], and various versions of PVM [10] can adjust the load per node on partially available workstations, but the processor set for the computation is fixed, once started, as opposed to our system.

Cilk-NOW [4] and Piranha [5] support adaptive parallel computation on NOWs in the sense that the pool of processors used can vary during the computation, as in our system. However, the Cilk-NOW system is restricted to *func-*

tional programs, and Piranha requires the adoption of the Linda tuple space as a parallel programming model and special code to achieve adaptivity.

Another class of systems including Adaptive Multiblock PARTI (AMP) [6] and Distributed Resource Management System (DRMS) [12] provide data distribution directives for the reconfiguration of the application to varying numbers of nodes at runtime. Our system distinguishes itself from these approaches by offering fully automatic data management.

All of the above systems require the use of specialized libraries or paradigms, in contrast to our use of an industry-standard programming model.

2.2 OpenMP and Task Parallel Applications

OpenMP uses the fork-join model of parallel execution. In the task queue model, each process [1] executes tasks from a shared queue, repeatedly fetching a new task from the queue until it is empty, no global synchronization between the processes is needed. Therefore, an OpenMP task parallel application typically only has one OpenMP fork at the beginning and one OpenMP join at the end.

2.3 OpenMP on a NOW

We used the TreadMarks DSM system [2] as a base for our implementation. TreadMarks is a user-level software DSM system that runs on commonly available Unix systems and on Windows NT, and it supports an OpenMP fork-join style of parallelism with the `Tmk_fork` and `Tmk_join` primitives for the master process, and the `Tmk_wait` primitive for the slave processes. We use the SUIF compiler toolkit [1] to automatically translate OpenMP programs into TreadMarks code [11]. Each OpenMP parallel construct is replaced by a call to `Tmk_fork` followed by a call to `Tmk_join`.

An important advantage of using the shared memory paradigm is the *automatic data distribution*, including the redistribution after an adaptation, relieving the programmer from this task.

2.4 Transparent Support of Adaptivity

In our model, slave processes perform all work either inside tasks or, as in data parallel applications, within other OpenMP parallel sections containing no tasks.

To allow for a transparent adaptation whenever an adapt event occurs while the application is busy with a task queue, we let each process finish its current task, then we let the system execute an *extra OpenMP join-fork sequence.* Having all slave processes' work for the current OpenMP parallel section being contained in the tasks ensures that slave processes do not have any compute-relevant private process state when the adaptation is performed. We introduce

[1] In our case, "process" and the OpenMP documentation's term "thread" are synonyms. In our implementation of OpenMP, these threads execute as Unix processes on various nodes, where a node is a machine.

Original code	Automatically modified code
`expr1;`	`expr1;`
`while (expr2) {`	`while (expr2 && !Tmk_leave()) {`
`   statement`	`   statement`
`   expr3;`	`   expr3;`
`}`	`}`
`do {`	`do {`
`   statement`	`   statement`
`} while(expression);`	`} while(expression && !Tmk_leave());`
`for (expr1; expr2; expr3)`	`for (expr1; expr2 && !Tmk_leave(); expr3)`

Table 1. Loop condition code modifications needed for adaptations. These transformations are done automatically by a preprocessor.

a new TreadMarks primitive `Tmk_leave` which the application calls to indicate completion of a task. This call returns true if a process is to leave, false otherwise. The preprocessor inserts this call at task boundaries. More precisely, the preprocessor modifies the termination-condition of top-level loops of the functions called by OpenMP forks according to the rules in Table 1. If a forked function does not have any other (compute-relevant) top-level statements besides a loop which retrieves and adds tasks, as in the applications investigated, then the preprocessor can perform the correct code modifications *automatically* (Figure 1).

Adaptations are completely *transparent* to the application, as the only application code modification is the insertion of `Tmk_leave`. There, leaving processes may terminate while continuing processes experience a slight delay while the system performs the adaptation, and joining processes begin execution of the forked function.

In our current model, task queues are maintained by the application, as the OpenMP standard does not explicitly support task queues. However, KAI have proposed their WorkQueue model [16] as an addition to the OpenMP standard, offering two new pragmas, `taskq` and `task`, for task queues and tasks, respectively. Following the acceptance of the proposal, we may modify our system accordingly, *eliminating* the need for the `Tmk_leave` primitive, as the system will recognize task boundaries through use of the `task` pragma.

The WorkQueue model allows nested task queues. In our model, we permit adaptations only in top-level task queues, other task queues are completed non-adaptively, avoiding the complexity of dealing with compute-relevant slave process states, such as letting another process complete some half-finished task of a leaving process.

3 Functionality

Processes may be added to or withdrawn from the computation, actions called *join events* and *leave events*, or collectively *adapt events*. The system performs

Code executed by all threads:

```
void _Worker_func(struct Tmk_sched_arg *_my_arguments)
{
    ...
    do {
        if (PopWork(&task) == -1) {
            break;
        }
        QuickSort(task.left, task.right);
    } while (1);  /* original code */
    /* modified line below replaces above line */
    } while (!Tmk_leave());
}
```

Code executed by master:

```
...
Tmk_sched_fork(_Worker_func, &Tmk_arguments);
...
```

Fig. 1. Example structure of a task queue application (Quicksort) showing modification for adaptations according to rules in Table 1.

requested adaptations at the next adaptation point. If several processes wish to leave and/or are ready to join when an adaptation point is reached, then these adapt events are all performed simultaneously. Such a scenario is actually much cheaper than performing the events sequentially, as the total cost per adaptation does not vary in proportion to the total number of leaves and/or joins performed at once. New processes require about 0.5-1 seconds from the join request until they are ready to join, but during this time all other processes proceed with their computations.

The only limitation in the current implementation is that the master process cannot be terminated.

4 Implementation

We have modified the TreadMarks version 1.1.0 system to support adaptivity. The current version of the system supports adaptive parallelism for both data parallel and task parallel applications, but we focus primarily on task parallel applications in the following description.

Join and leave requests may be sent to the system from any external source via a TCP/IP connection.

4.1 Join Events

For a join event, the master spawns a new process p_{new} on the designated machine and all processes set up network connections to p_{new} while still continuing with their work, i.e. any slow low-level process initializations do not affect the on-going computation. Once p_{new} is ready to begin work, the master starts an *adaptation phase*: It notifies the other processes of the adapt event, whereupon all processes continue until they reach the next *adaptation point*, either `Tmk_leave` or a regular OpenMP join present in the application code.

Here, all processes perform an OpenMP join and fork, with slaves receiving adaptation information such as new process identifiers. This extra join-fork is initiated by the system and is therefore not in the application source code. Also, the master does not send a pointer with a compute function to the old slaves, only the new process p_{new} receives a pointer to the current compute-function. Now, all old processes perform a garbage collection. This mechanism causes diffs to be fetched and applied, then each node discards internal memory consistency information (such as twins, diffs, write notices, intervals lists [2]). A garbage collection typically costs only a few milliseconds. Thereafter, all shared memory pages are either up-to-date or discarded. In the latter case, an access will cause the page to be fetched from another process with an up-to-date copy.

The system now performs three barrier synchronizations. The first barrier's departure message to the new process includes all non-default page-state information for all pages. This barrier guarantees that garbage collection is completed before page state information is used subsequently.

Next, a second barrier is performed, then all necessary reassignments are performed, including the redistribution of lock managers and lock tokens, and all memory consistency information is cleared. This second barrier ensures that any duplicate departure messages of the first barrier are not sent after some process has already begun with any reassignments.

Thereafter a third barrier is performed, ensuring that no process can proceed with its computation before all processes have performed reassignments and cleared their old consistency information. This barrier concludes the adaptation phase, and processes resume or begin work.

4.2 Leave Events

The handling of leave events is similar to the handling of join events: When a leave request arrives, the master begins the adaptation phase by notifying all processes. Once all processes have reached an adaptation point, an OpenMP join and fork is executed, followed by a garbage collection.

The system then performs three barrier synchronizations as previously, but with some additions: All pages that are exclusively valid on a leaving process must be transfered to a continuing process. For this, all old slaves include page-state information in the arrival-message for the first barrier, then the master allocates an approximately equal number of such pages among the continuing processes and includes this information in the barrier departure messages. This barrier guarantees that garbage collection is completed before page state information is used and before any pages are moved off leaving processes. Processes now in parallel fetch these pages as allocated by the master and assume ownership.

After a second barrier, reassignments are again done and consistency information is cleared. This second barrier ensures that any page transfers off leaving processes are completed, so leaving processes can now terminate, and the third barrier is performed without participation of leaving processes.

4.3 Multiple Adapt Events

Join and leave requests may arrive anytime. Leave requests are always given priority over join requests, as a compute process may need to be cleared off some machine rapidly. Requests are therefore handled according to the policy of including any adapt event that can be included in the current adaptation without delaying any pending leave, other requests are postponed until completion of the adaptation.

The system starts an adaptation phase immediately upon receipt of a leave request, unless the system is already in an adaptation phase or a sequential phase. This policy causes about-to-join processes which, at the adaptation point, are still busy setting up network connections, to be aborted and restarted after the adaptation.

Any deferred or aborted adapt events are performed upon completion of the adaptation phase, with leave requests being handled first.

4.4 Special Cases

Consider the scenario where some processes arrive at an OpenMP join $J1$ belonging to the application while, due to an adapt request, other processes first arrive at the adaptation point and are executing another OpenMP join $J2$ constituting the adaptation point, before having reached $J1$. Space limitation does not permit a detailed discussion here, but the system handles such cases correctly.

Consider further an adaptation request arriving while the system is in a sequential phase, i.e. in-between an OpenMP join and fork. In this case, the adaptation is performed immediately when the sequential phase is over. Any such delay is not so tragic, as a process wishing to withdraw is idle during this phase and is not using compute resources.

5 Overview of Performance

5.1 Experimental Environment

Our testbed consists of 8 400MHz Pentium II machines with 256MB of memory, and we run Linux 2.2.7. For the communication, we use UDP sockets, and the machines are connected via two separate switched, full-duplex Ethernet networks with bandwidths of 100Mbps and 1Gbps, respectively. The 1Gbps network only offers extra bandwidth compared to the 100Mbps network, as the latency is very similar in both networks. We exploit this by increasing the page size from 4K to 16K when using the 1Gbps network.

			Shared memory		Avg.	Avg. time (sec.)		Number/amount of transfers			
	Size	Network	# Pages	Page size	# Tasks	Total	Per task	Pages	MB	Messages	Diffs
Traveling	19 cities	100Mbps	108	4k	619	7.63	0.10	7018	28.75	23363	651
Salesman		1Gbps	27	16k	619	5.94	0.08	4320	68.98	15319	654
Quicksort	10'000'000	100Mbps	9798	4k	613	10.86	0.14	31369	124.77	129417	739
	integers	1Gbps	2450	16k	613	6.46	0.08	8882	142.82	40886	1393

Table 2. Application characteristics and network traffic for 8-thread runs on the non-adaptive or on the adaptive system without any adapt events.

5.2 Applications

We use the two task queue applications from the standard TreadMarks distribution: Quicksort and TSP (Table 2).

Quicksort sorts an array of integers by adding and retrieving tasks of not-yet-sorted subarrays to and from a central task queue, respectively. Each array is repeatedly split into two subarrays around a selected pivot-value: The shorter one is put on the task queue and the thread recurses on the longer one, until its length falls below a threshold, then it is sorted locally.

TSP uses a branch-and-bound algorithm to solve the traveling salesman problem. Tasks representing partial tours are repeatedly added to and retrieved from a central task queue. Subtours of a given maximum length are solved recursively locally, while longer ones are split into subproblems and added to the task queue.

5.3 No Overhead for Providing Adaptivity

The provision of adaptivity costs virtually nothing compared to the standard non-adaptive TreadMarks system [2] , as no extra messages are sent in the absence of adapt events.

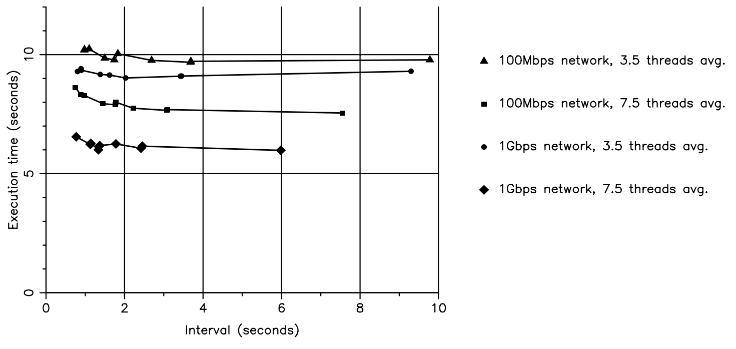
5.4 Measurement Methodologies

For multiple adaptations during the course of execution, we first calculate the average number of processes used during the whole run (e.g. 7.55) by measuring the times in-between each adaptation, then we adjust the runtime to represent a desired average (e.g. 7.5), using a speedup-curve obtained from non-adaptive runs. The adaptation overhead is the difference in runtime compared to a (theoretical) non-adaptive run of the same average, as calculated in the speedup-curve.

To quantify in detail a single adaptation from p to q processes, we collect statistics beginning only at a point immediately preceding the adaptation and compare the results with a non-adaptive run of q processes. We ensure that the number of tasks completed (i.e. the average amount of work done) during statistics measurements is equal in both cases. For the adaptive run, the measured data layout is initially pre-adaptation, but all measured work is done post-adaptation. The difference between the adaptive and non-adaptive run reflects the cost of the adaptation.

Obviously, the two tested applications have a non-deterministic execution, as any task may be executed by any process, and the length and number of

[2] Our measurements do not show any difference.



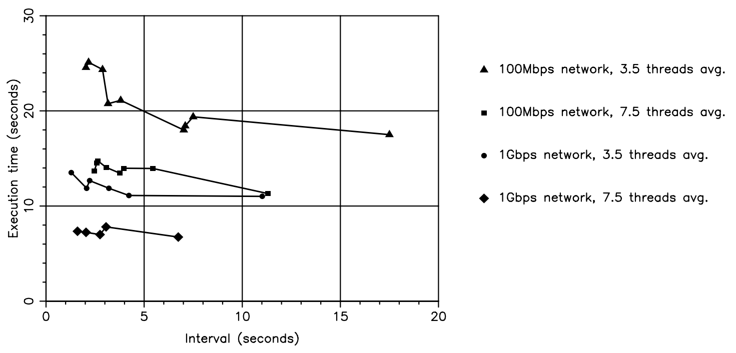


Fig. 2. Execution times for different intervals between adapt events, for TSP (above) and Quicksort (below).

individual tasks varies both within one test run and between different test runs, especially for Quicksort, which uses a random input. However, the variations are small enough for our methodologies to show clear trends, especially in combination with the averaging of results obtained from several test runs.

5.5 Cost of Joins and Leaves

To provide an idea of the overhead of adaptations, we periodically caused an adapt event to occur. Figure 2 shows how the total runtime varies as a function of the interval between successive adapt events. Starting with 8 or 4 processes, we let the system alternately perform a leave or a join event at the end of each interval, resulting in about 7.5 or 3.5 processes, on average. For the leaves, we let each of the slave processes leave in turn.

Variations in execution time due to the non-deterministic nature of the applications are apparent in Figure 2, as the points in the graphs represent individual test runs. Nevertheless, the trend of an increase in runtime in proportion with an increase in adaptation frequency, as expected, is evident: Every adaptation adds a similar delay to the total runtime.

In TSP, even frequent adaptations of one every second hardly increase the total runtime. In Quicksort, one adaptation every 5 seconds may increase the

	Avg. # procs.	Network	Pre-adapt.-delay cost	Adapt. cost	Total
TSP	3.5 or 7.5	100Mbps or 1Gbps	0.05	0.05	0.1
Quicksort	7.5	100Mbps	0.05	0.5	0.55
	3.5		0.05	1	1.05
Quicksort	7.5	1Gbps	0.05	0.2	0.25
	3.5		0.05	0.4	0.45

Table 3. Typical average costs (in seconds) per adaptation (For TSP, the exact differences between the various setups are difficult to quantify precisely, as the absolute costs are small in all cases).

runtime by perhaps 10%. The graphs also show how adaptations in the 7.5 process runs are cheaper than in the 3.5 process runs, for equal adaptation frequencies, as explained in the next section, and how the faster network offers significantly better performance, both in total runtime and in adaptation costs.

Table 4 provides detailed results for individual adaptations, obtained using the measurement methodology for single adaptations described in the previous section. Table 3 is a summary of Table 4.

We show the number of extra adaptation-induced page fetches occurring during the course of computation after an adaptation (P_{appl} or *Pages Appl.* in the table), as the application experiences extra access misses, and the number of pages explicitly moved off any leaving process by the system (P_{system} or *Pages System*, i.e. all pages of which only the leaving processes have valid copies). The table further shows the cost in seconds for these page transfers [3] .

As both applications execute non-deterministically, such that variations in runtime of 0.5 seconds for identical test runs of Quicksort are not uncommon, we show a lower and upper bound for each adaptation, giving a *range of values*: The numbers were computed by comparing the best- and worst-case adaptive results with the average of the corresponding non-adaptive results. For each adaptation, in different runs, we adapted at several different times during each application's execution, and we repeated each individual test case several times. Negative values show that an adaptation can even lead to less data transfers and an earlier completion of the computation than a comparable non-adaptive run.

The *total cost of an adaptation* is the sum of the cost of the P_{appl} and, if applicable, the P_{system} transfers, plus a *pre-adaptation delay* incurred by waiting at an adaptation point for all processes to arrive. P_{system} page transfers obviously only occur if at least one process is leaving. The pre-adaptation delay is the overall compute time lost before the adaptation begins, i.e. the average of all processes' idle times, occurring after completion of a task, while a process is waiting at the OpenMP join which initiates the adaptation.

[3] The *Time Pages Appl.* values actually include other adaptation-related costs such as garbage collection and management of data structures. We do not present these separately, as their share of the total costs is minimal, on the order of 1%.

The pre-adaptation delay cost obviously varies with the length of the tasks. For the applications tested, it is typically in the range of 0-0.1 seconds, and in a few percent of the cases, it is around 0.2 seconds. Only Quicksort rarely has significantly longer delays: In about 1% of the cases, the cost is on the order of 0.5 seconds.

The results shown in Figure 2, which contain all costs, confirm that the pre-adaptation costs are small: Given the total runtime increase and the frequency of adaptations in the graphs, one can easily estimate an *average cost per adaptation* C_{avg} and verify that these costs are hardly higher than the costs for the P_{appl} plus the P_{system} transfers reported in Table 4. At the same time, these C_{avg} results also validate the measurement methodology for single adaptations used for Table 4.

We observe that the costs for TSP are very small in all cases, so the absolute values are not very meaningful, especially given the large range of measured values compared to the absolute upper and lower bounds. Table 2 shows that TSP uses little shared memory, causing little data redistribution at an adaptation. The conclusion therefore is that in the absence of large data redistributions, adaptations are very cheap, i.e. there are no significant other costs.

For Quicksort, we observe both positive and negative values. On average, adaptations for this application also cost only a fraction of a second. We analyse the results in more detail in the next section.

Table 4 further shows that the percentage of shared memory pages moved extra due to an adaptation is very small in nearly all cases for Quicksort (a few %), so the absolute costs remain small compared to our previous results of data parallel applications, where redistribution of 30%-60% of all shared memory pages is common [15].

To sum up, Table 4 shows that the costs of an adaptation are typically less than 0.1 seconds for TSP and less than 0.5 seconds for Quicksort even in the slower of the two environments tested, when using around 8 processes.

6 Analysis of Performance

The key cost component of an adaptation is the *additional network traffic* for the data redistribution caused by this event. We therefore analyse the *extra* page transfers attributable to the adaptation, as compared to the non-adaptive case.

Furthermore, we point out the main differences between *independent* and *regular* applications: We call applications where the data layout is independent of process identifiers *independent* applications, as opposed to *regular* applications which have a block or block-cyclic data distribution. In regular applications, a process' data partition is determined by the process identifier and the current number of processes, and the process performs most work using the data in its partition. Adaptations generate a large shift in each process' assigned data partition, and in general all pages that were not in the pre-adaptation data partition have to be fetched extra after the adaptation.

	Traveling Salesman								Quicksort							
Adaptation	Time Pages Appl.		Pages Appl.		Time Pages System		Pages System		Time Pages Appl.		Pages Appl.		Time Pages System		Pages System	
100Mbps Ethernet environment with page size of 4K																
8 → 8	0.01	0.20	-34	307	0.00	0.00	1	15	0.06	0.19	-264	674	0.30	0.63	874	1781
8 → 7	0.05	0.23	21	367	0.00	0.01	1	28	-0.50	-0.15	-743	586	0.27	0.43	685	1205
8 → 6	0.02	0.05	-35	104	0.00	0.00	1	6	-0.54	-0.26	-1557	-618	0.47	0.83	2168	3676
7 → 8	-0.04	0.05	-61	41		-		-	0.19	0.59	-485	1404		-		-
6 → 8	-0.04	0.05	-41	53		-		-	0.31	0.41	119	1531		-		-
4 → 3	0.03	0.07	15	51	0.00	0.00	0	4	-1.50	0.27	-2204	280	0.36	0.68	1044	1983
3 → 4	-0.01	0.07	4	78		-		-	0.81	2.78	1322	3440		-		-
1Gbps Ethernet environment with page size of 16K																
8 → 8	0.03	0.17	-58	330	0.00	0.00	0	9	-0.08	0.04	-162	91	0.08	0.10	274	354
8 → 7	0.01	0.15	0	309	0.00	0.01	0	18	-0.04	0.04	-137	50	0.07	0.09	252	337
8 → 6	-0.04	0.01	-65	10	0.00	0.00	0	0	-1.30	-0.20	-964	-56	0.07	0.13	394	704
7 → 8	0.02	0.06	10	43		-		-	0.00	0.25	-37	539		-		-
6 → 8	-0.03	0.06	-10	31		-		-	1.17	1.54	710	978		-		-
4 → 3	-0.03	0.02	-42	12	0.00	0.00	0	0	-0.26	0.90	-598	680	0.17	0.21	606	739
3 → 4	-0.01	0.02	-32	10		-		-	-0.64	0.11	-486	365		-		-

Table 4. Typical costs for various adaptations (excluding pre-adaptation delay) in two test environments, in seconds and number of 4k or 16k pages. For each case, we show the lower and upper bound of values measured in a series of representative tests. We performed one or two leaves from 8 processes (8 → 7, 8 → 6), one or two joins to 8 processes (7 → 8, 6 → 8), one simultaneous leave and join with 8 processes (8 → 8), and one leave from or one join to 4 processes (4 → 3, 3 → 4).

In contrast, in independent applications such as Quicksort and TSP tasks are not bound to processes, any task may be solved by any process, so the probability that a first-time page access of a task is for a locally-valid page depends on issues such as the number of different tasks using the same page and whether pages are mostly read (as in TSP) or also written to (as in Quicksort). As there are no assigned data partitions, an adaptation does not cause data repartitioning.

Adaptations in general are much cheaper when more processes are involved: Not only does a join or a leave of $7 \rightarrow 8$ or $8 \rightarrow 7$ processes cause less data transfer than a join or a leave of $3 \rightarrow 4$ or $4 \rightarrow 3$ processes, but more significantly, with a larger number of processes, the number of page transfers *per process* and equally *per network link* is much lower, so far more page transfers occur in parallel. Table 4 shows that the range of values for P_{appl} transfer costs are higher for less processes involved.

We examine more specific effects of joins and leaves in regular and independent applications in the following two subsections.

6.1 Join Events

Adding new compute processes may cause the following data movements: (1) the faulting-in of pages by the joining processes, as all their pages are invalid initially, and (2) the data redistribution among the "old" processes, when the total number of processes changes, i.e. when the number of joining and leaving processes is not equal.

In regular applications, in most cases all shared memory pages are accessed repeatedly many times. Joining processes therefore generally have to page in the complete partition assigned to them, typically $1/n$ of all pages for n processes, which is more than the number of extra pages fetched by any other process due to the data redistribution. The transfers are less only if not all of the partition's data is accessed anymore during the rest of the computation. As each process typically performs the same amount of work within one OpenMP parallel section, the bottleneck is the process fetching the largest number of pages, i.e. the paging-in of the joining processes' data partitions constitutes the bottleneck.

Independent applications however do not assign data partitions. In TSP, where many tasks reside in the same shared memory page and most accesses are read-only, processes often have a valid copy of most of the pages used overall. Any joining process therefore needs to page-in all these pages extra, so with several processes joining, the total extra data transfer may exceed 100% of the shared memory pages in use. In Quicksort, with many write-accesses to pages, most valid pages are in the exclusive state, only a few in the shared state, and each of n processes typically has about $1/n$ of all used pages valid. As each page is accessed only a few times, as joins occur closer to the end, a new process pages in much less than $1/n$ of all pages in use, as most pages are not needed anymore. Furthermore, due to the absence of data redistribution, independent applications experience less traffic among the "old" processes when adapting, compared to no adaptation: With a join of $m \rightarrow n$ processes $(m < n)$, on average more pages are valid per process for m than for n processes. As expected, Table 4 shows that the number of P_{appl} transfers as a percentage of all shared memory pages (cf. Table 2) is much smaller than for regular applications, where percentages of 40-60% are common [15].

Another more significant difference between independent and regular applications is the fact that processes compute tasks at their own pace in independent applications, therefore a larger number of page fetches by one process, such as a join, does not cause all other processes to wait, so only the fetching and the sending process lose compute time, as opposed to regular applications, where all processes lose compute time, waiting at the next OpenMP join.

In conclusion, due to the above reasons join events are significantly cheaper in independent applications than in regular applications.

6.2 Leave Events

A leave of processes may cause the following data movements: (1) All pages P_{system} exclusively valid on the leaving processes are moved to continuing processes, and (2) the data repartitioning among the continuing processes generates page fetches P_{appl}. This may include some of the P_{system} pages, as the system allocates these without knowledge of any data partitioning.

The share of P_{system} transfers is comparable for regular and independent applications: In both cases, in applications with little read-only sharing, given n processes before the adaptation, a leaving process often has about $1/n$ of the pages in use in an exclusively-valid state, so these pages are evenly distributed among the continuing or joining processes. Table 4 shows that the share of P_{system} pages is in the expected percentage range (cf. Table 2), and the numbers for two leaves from $8 \rightarrow 6$ processes are about double the numbers for one leave from $8 \rightarrow 7$ and $8 \rightarrow 8$ processes.

Thereafter, regular applications experience data repartitioning, as in the case of joins. The number of P_{appl} transfers are less than for joins, because generally no process has to page in its complete data partition as a join does, but the data repartitioning still affects around 30-50% of all shared memory pages [15].

In independent applications however, after having received the P_{system} pages, the continuing processes each have about the same share of valid pages as in the corresponding non-adaptive case, where the P_{system} pages are valid on some process already. Therefore, P_{appl} is around zero, as there is also no data repartitioning. Table 4 shows that the P_{appl} values often vary within ranges of both positive and negative values.

When sending P_{system} pages, the system batches several pages into one message, whereas P_{appl} page transfers only contain one page per message. However, all P_{system} pages have to be fetched from the one (or few) leaving process(es), so these transfers occur less in parallel than the P_{appl} transfers. In addition, no process is performing any work while any P_{system} transfers are in progress. In contrast, in independent applications, any P_{appl} transfers occur while other processes continue working (while in regular applications they may be idle).

In conclusion, while the cost of leaves is dominated by the P_{appl} transfer costs in regular applications and this component is around zero for independent applications, while the P_{system} transfer costs are similar in both cases, leave events are generally significantly cheaper in independent applications than in regular applications.

7 Discussion and Conclusions

We have developed a system providing transparent adaptive parallel execution of OpenMP applications on NOWs. Our system combines the convenience of an industry standard programming model, OpenMP, with a flexible and user-friendly usage. Users can easily grant or revoke use of a machine at any time using a graphical user interface, or the system can even be controlled automatically, but we do not analyse user behavior in this paper.

Obviously, the performance of a software DSM system cannot match the performance of a dedicated hardware shared memory system. Rather, our system should be assessed as enabling otherwise idle machines to be used productively - especially for longer-running computations - thanks to the flexibility which adaptivity offers, something previously impossible due to conflicting resource requirements in a multi-user environment. In many cases, our system eliminates the need for a reservation of machines for parallel processing. When using existing NOWs, no hardware costs arise, and running existing OpenMP applications means no software development costs are incurred either.

We have demonstrated that the cost of adaptation is modest and that it is significantly lower for independent applications, where the data distribution is independent of the process identifiers, than for regular applications.

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. J. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. . Stephan. Dome: Parallel programming in a heterogeneous multi-user environment. Technical Report CMU-CS-95-137, Computer Science Department, Carnegie Mellon University, April 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*, pages 133–147, January 1997.
5. N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and piranha. *IEEE Computer*, 28(1):40–49, January 1995.
6. 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, January 1997.
7. P. J. Hatcher and M. J. Quinn. *Data-parallel Programming on MIMD Computers.* The MIT Press, Cambridge MA, 1991.
8. L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I - Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.
9. L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II - The Runtime System. *IEEE Transactions on Parallel and Distributed Systems*, 1994.
10. 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.
11. H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on networks of workstations. In *Proc. Supercomputing '98*, Orlando, FL, November 1998. ACM/IEEE.
12. J. E. Moreira, V. K. Naik, and R. B. Konuru. A system for dynamic resource allocation and data distribution. Technical Report RC 20257, IBM Research Division, October 1995.
13. N. Nedeljkovic and M.J. Quinn. Data-parallel programming on a network of heterogeneous workstations. *Concurrency: Practice & Experience*, 5(4):257–268, June 1993.
14. OpenMP Group. http:/www.openmp.org, 1997.
15. A. Scherer, H. Lu, T. Gross, and W. Zwaenepoel. Transparent Adaptive Parallelism on NOWs using OpenMP. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, Atlanta, May 1999. ACM.
16. S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible Control Structures for Parallelism in OpenMP. In *First European Workshop on OpenMP (EWOMP '99)*, Lund, Sweden, September/October 1999. Kuck & Associates, Incorporated.