Experiences using OpenMP based on Compiler Directed Software DSM on a PC Cluster

Matthias Hess* Gabriele Jost† Matthias Müller* Roland Rühle*

HLRS
Allmandring 30
70550 Stuttgart
Germany

NASA Ames Research Center
Moffett Field
CA 94035-1000
USA.

July 11, 2002

Abstract

In this work we report on our experiences running OpenMP programs on a commodity cluster of PCs running a software distributed shared memory (DSM) system. We describe our test environment and report on the performance of a subset of the NAS Parallel Benchmarks that have been automaticaly parallelized for OpenMP. We compare the performance of the OpenMP implementations with that of their message passing counterparts and discuss performance differences.

1 Introduction

Computer Architectures using clusters of PCs with commodity networking have become a low cost alternative for high end scientific computing. Currently message passing is the dominating programming model for such clusters. The development of a parallel program based on message passing adds a new level of complexity to the software engineering process since not only computation, but also the explicit movement of data between the processes must be specified.

Shared memory parallel processors (SMP) provide a user friendlier programming model. The use of globally addressable memory allows users to exploit parallelism while avoiding the difficulties of explicit data distribution on parallel machines. Parallelism is commonly achieved by multi-threading the execution of loops. Compiler directives to support multithreaded execution of loops are supported on most shared memory parallel platforms. In addition, many compilers provide an automatic parallelization feature taking all the burden of code analysis off the user. Efficiency of compiler parallelized code is often limited, since a thorough dependence analysis is not possible without user information. Alternatively, there are parallelization support tools available which take the tedious work of dependence analysis and generation of directives off the user but allow user guidance for critical parts of the code. An example of such a tool is CAPO [10].

While shared memory architectures provide a convenient programming model for the user, their drawback is that they are expensive and the scalability of the code may be limited due to poor data

*HLRS (High Performance Computing Center, Stuttgart)
†NASA, employee of Computer Sciences Corporation.

locality and possibly large synchronization overhead. During recent years there have been considerable efforts to develop system software to support DSM (Distributed Shared Memory) programming which enables the user to employ the convenient shared memory programming model on a network of processors, thereby maintaining the ease of use while maintaining the low cost of hardware. Examples of such systems are TreadMarks [2] and SCASH [13]. These systems allow the support of OpenMP parallelization on clusters of processors, thereby removing the major impediment to their usage which is the high effort to develop a message passing version from a sequential program. We have installed publicly available DSM software on a commodity cluster of PCs and tested its performance on a set of benchmark kernels. The paper seeks to address the issue of evaluating the efficiency of DSM without explicit hardware support. The rest of the paper is structured as follows: In section 2 we discuss the message passing and the shared address space programming models. In section 3 we describe the hardware platform and system software of our test environment. In section 4 we describe our evaluation strategy and discuss the performance of the individual benchmark kernels. In section 5 we discuss some of the problems we encountered. In section 6 we briefly examine some related work and in section 7 we summarize our conclusions and discuss future work.

2 Programming Models

Currently message passing and shared address space are the two leading programming models for clusters of SMPs.

2.1 Message Passing

Message passing is a well understood programming paradigm. The computational work and the associated data are distributed between a number of processes. If a process needs to access data located in the memory of another process, it has to be communicated via the exchange of messages. The data transfer requires cooperative operations to be performed by each process, that is, every send must have a matching receive. The regular message passing communication achieves two effects: communication of data from sender to receiver and synchronization of sender with receiver.

MPI (Message Passing Interface) [12] is a widely accepted standard for writing message passing programs. It is a standard programming interface for the construction of a portable, parallel application in Fortran or in C/C++, which is commonly used when the application can be decomposed into a fixed number of processes operating in a fixed topology (for example, a pipeline, grid, or tree). MPI provides the user with a programming model where processes communicate by calling library routines to send and receive messages. Pairs of processes can perform point-to-point communication to exchange messages. For increased convenience and performance a group of processes can also call collective communication routines to implement global operations such as broadcasting values or calculating global sums. Global synchronization can be implemented by calls to barrier routines. Asynchronous communication is supported by providing calls for probing and waiting for certain messages. In MPI-1, all communication operations require the sending as well as the receiving side to issue calls to the message passing library.

2.2 Shared Address Space

Parallel programming on a shared memory machine can take advantage of the globally shared address space. Compilers for shared memory architectures usually support multi-threaded execution of a program. Loop level parallelism can be exploited by using compiler directives such as those defined

in the OpenMP standard [14]. Multiple execution threads are automatically created for performing the work in parallel. Data transfer between threads is done by direct memory references. OpenMP provides a fork/join execution model in which a program begins execution as a single process or thread. This thread executes sequentially until a PARALLEL construct is found. At this time, the thread creates a team of threads and it becomes its master thread. All threads execute the statements lexically enclosed by the parallel construct. Work-sharing constructs (DO, SECTIONS and SINGLE) are provided to divide the execution of the enclosed code region among the members of a team. All threads are independent and may synchronize at the end of each work-sharing construct or at specific points either implicitly or explicitly (specified by the BARRIER directive). Exclusive execution mode is also possible through the definition of CRITICAL regions.

This approach provides a relatively easy way to develop parallel programs but has disadvantages. It is often difficult to achieve scalability of the code for a large number of processors due to a lack of data locality and excessive synchronization costs.

3 Hardware Platform and Software Description

Our test environment consists of a cluster of commodity PCs at the High Performance Computing Center of the University of Stuttgart (HLRS). In the following we give some details about hardware and system software.

3.1 Platform description

We have used a cluster at HLRS consisting of 8 NEC 120Ed server nodes as the test platform. The nodes are dual processor systems with two 1 GHz Pentium III and 2 GB of main memory. Each node is equipped with a Myrinet 2000 NIC in a fast 64 bit / 66 MHz PCI slot. The nodes are based on the ServerSet III HE chipset and have a good communication performance to the Myrinet cards. The bandwidth from memory to the card is 409 MB/s for read operations and 480 MB/s for write operations. These data have been acquired with the program 'gm_debug' provided by Myricom. A collection of data for other motherboards and chipsets can be found at [1]. For our evaluation we used only one CPU per node.

In order to compare the performance of SCASH with a true shared memory system, we used a 16-way NEC AzusA. The AzusA is a shared memory system with IA-64 processors. Both systems, the distributed memory cluster and the shared memory AzusA, were running Linux in its 2.4 version. This reduces effects due to different memory managments of different operating systems on the distributed and the shared memory architecture. The performance impact of different memory mangement systems is discussed in In [5].

We did not have a four or eight processor IA-32 system available for the tests.

3.2 SCore

SCore is a parallel programming environment for workstations and PC clusters, developed by the Real World Computing Partnership (RWCP). The project has now been transferred to the PC Cluster Consortium. Amongst other features, SCore provides its own communication layer called PM [19, 20]. It aims at providing a uniform interface to different communication devices like Fast Ethernet, Gigabit Ethernet or Myrinet

SCore also supports different parallel programming paradigms like message passing or shared memory. On the message passing side there is a MPI-implementation based on MPICH with an addi-

tional device specifically designed for the PM layer. Shared memory is supported in two ways. The PM layer has a shared memory device that is intended for SMP systems. It uses memory-mapped shared segements for the communication between processes on a true shared memory system. Additionally, the SCore architecture has a software distributed shared memory system called SCASH [6], that we employed to obtain the results of the tests we present in this paper.

3.3 SCASH

SCASH [6] is a page-based software distributed shared memory system. It is implemented as a user-space runtime library which uses the PM layer for communicating pages between cluster nodes.

It employs an eager release consistency model to ensure the consistency of shared memory on a per-page basis. This means that at memory synchronisation points only modified parts of memory are updated, which usually requires exchange of data between nodes.

The home node of a page is the node that keeps the latest data of the page. If other nodes change the data within a page it must be updated on the home node. To reduce memory transfer, SCASH also provides the possibility to change the home node of a page. It is possible to use two page consistency protocols, an invalidate and an update protocol, which can be chosen dynamically.

To reduce memory transfer between nodes, the nodes use cached copies of requested pages. Only on write operations to the memory can these copies become inconsistent. The update protocol specifies that all copies of a particular page be updated once one node changes its contents.

In the invalidate protocol, the home node of a page notifies all nodes which share that page when a page has been altered and cached copies of that page on other nodes become invalid.

3.4 Omni OpenMP

Omni OpenMP is a collection of programs and libraries that enable OpenMP for back-end compilers that do not support it natively. The front-end to these compilers translates C or Fortran77 OpenMP source texts into multi-threaded C with calls to a runtime library.

One of the main goals of Omni OpenMP is portability, so the translation pass from an OpenMP program to the target code is written in JAVA. The target code is – in turn – compiled by the back-end C compiler on the target platform. For the tests presented here we used the GNU C Compiler as the backend compiler.

The Omni compiler suite can be configured to use several different underlying libraries. For the thread system Solaris Threads or pthreads are supported, but there is also support for StackThreads [18] developed by Real World Computing Project (RWCP). In addition to the support of threads there is support for several shared memory implementations, like UNIX shmem. In our tests we used the support for the SCASH distributed shared memory system which has been described above.

The Omni OpenMP compiler suite is also available for IA-64. For tests on the shared memory Azusa system (see 3.1) we used the Omni compiler, too, again in order to minimize the influence of different software. This way we can attribute certain observations to either the DSM system or the Omni OpenMP compiler.

4 Case studies

For our evaluation we selected a subset of the NAS Parallel Benchmarks [3]. They were designed to compare the performance of parallel computers for computational fluid dynamics (CFD) applications.

The full suite consists of five benchmark kernels and three simulated CFD applications. We selected three of the five benchmark kernels for our study.

4.1 Evaluation Strategy

To evaluate the performance of our test environment we compare the timings of OpenMP implementations of the benchmark kernels to:

1. Timings of their message passing counterparts on the same system.
2. Timings obtained on a true shared memory system but with the same operating system and therefore a comparable memory managment system.

The first OpenMP versus MPI comparison will give us some means to determine how well the DSM software handles memory coherency and synchronization. In the MPI implementation access to remote data is achieved by calls to the message passing library. The user has control over data locality and decides when and how much data to communicate. This provides the opportunity to minimize communication during program execution. Another aspect of the message passing approach is that data communication and synchronization are integrated. The send and receive operations not only exchange data, but also regulate the progress of the processes. In the OpenMP implementation the locations of the data, the amount of data to be communicated, and the synchronization among the threads depends on the DSM system and the compiler. As explained in section 3, the DSM system detects the necessity of communicating data when a page of memory is accessed that has been marked as updated by another process. We will use the number of page requests as an indicator for the amount of communication in the DSM system. Even in the case where a hand-optimized message passing implementation outperforms the DSM system, the ease of application porting may compensate for a certain loss of performance.

The comparison of OpenMP on a cluster versus OpenMP on a shared memory node gives us some estimate of the speedup that can be expected from the OpenMP programming paradigm on a true shared memory architecture. Our test platforms are described in section 3. We use the Omni compiler on both platforms.

The benchmarks come in different classes determined by the problem size. We ran only the small problems of class S,W, and A, since we encountered some problems with the larger sizes which will be discussed in section 5. Since our system is small, consisting of only 8 nodes, it is hazardous to extrapolate the scalability studies to larger systems. However, running the very small benchmark classes allows us to gain some insight into how the computation to communication ratio impacts the performance.

Since the ease of application porting is an important factor in favor of the DSM system, we started out with a sequential version of our benchmark kernels and used the automatic parallelization support tool CAPO [10] to insert OpenMP directives, thereby minimizing the parallelization effort. CAPO was developed at the NASA Ames Research Center. It takes as input a sequential Fortran program. It then performs an extensive dependence analysis over statements, loop iterations, and subroutine calls and generates Fortran code containing OpenMP directives. CAPO is based on the dependence analysis module of the CAPTools [8] parallelization tool. Our starting point for the message passing version of the benchmark kernels was the NPB2.3 [4] release of the NAS Parallel benchmarks. For the OpenMP implementations we started with an optimized serial implementation of the same benchmarks as described in [9]. The structure of the serial code is kept very close to the message passing code. Only slight modifications were applied to the kernels considered in our study and we

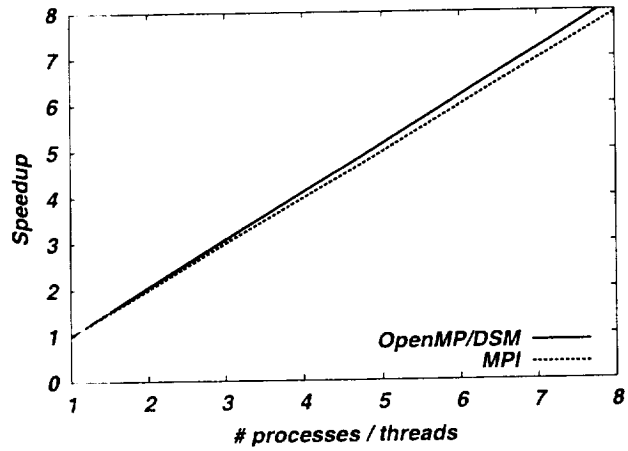


Figure 1: Speedups for class A of the EP benchmark for OpenMP/DSM and MPI

will describe them in the sections below. A good description about how to use CAPO for the OpenMP parallelization of the benchmarks is given in [10].

4.2 The EP benchmark kernel

EP stands for embarrassingly parallel. The kernel generates pairs of Gaussian random deviates according to a specific scheme. As the name suggests, the iterations of the main loop can be executed in parallel. Tool based OpenMP parallelization of the kernel was possible without user interaction. Once the data is distributed, the main loop which generates the Gaussian pairs and tallies the counts does not require access to remote data except for several global sum reductions at the end. In the MPI implementation the global sum is achieved by calls to `mpi_allreduce`. The OpenMP implementation uses the OMP PARALLEL REDUCTION directive. The MPI implementation shows a very low communication overhead, which is less than 1 % even for the smallest benchmark class on 8 nodes. If m denotes the log2 of the number of complex pairs of uniform (0, 1) random numbers, then the problem size of the benchmark classes under consideration is:

Class S: m = 24
Class W: m = 25
Class A: m = 28

The OpenMP/DSM implementation shows a very low number of page requests to the DSM system. As expected, the message passing as well as the OpenMP/DSM implementation show an almost linear speedup for all benchmark classes. For 8 nodes the OpenMP/DSM performance ranges within 97 % to 102% of that of MPI, depending on the benchmarks class. As an example we show the speedup for class for class A, in fig. 4.2.

4.3 The CG benchmark kernel

The CG benchmarks kernel uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. The kernel is useful for testing unstructured grid computations and communications since the underlying matrix has randomly generated locations of entries. Parallelization for message passing and directive based versions occur on the

same level within the conjugate gradient algorithm. The basic parallel operations are: sparse matrix vector multiply, AXPY operations, and sum reductions. The code was parallelized using CAPO without any user interaction. If na denotes the number of rows of the sparse matrix and nz the number of non-zero elements per row, then the problem size of the benchmark classes under consideration are:

Class S: na = 1400 nz = 7
Class W: na = 7000 nz = 8
Class A: na = 14000 nz = 11

In fig. 2 we show the speedup for the three benchmark classes. For class A, the MPI as well as the OpenMP/DSM and OpenMP/SMP implementations show reasonable speedup. The OpenMP/SMP version shows occasional superlinear speedup due to cache effects. For 8 nodes, the OpenMP/DSM efficiency reaches about 75% of that of MPI. The MPI version maintains this speedup for the smaller problem sizes but the performance of the OpenMP/DSM version decreases drastically. For 8 nodes and class W the OpenMP/DSM efficiency is only 35% and for class S is goes down to 6% yielding a speedup of less than 1.

The class S problem size is far too small to serve as a realistic example. However, we have a closer look at the perfromance differences for this class to get an idea about potential scalability issues related to the DSM system.

Our first observation is that the Omni compiler and its runtime library introduce additonal overhead which decreases performance even on a shared memory system. This is demonstrated in fig.2d, where we compare the speedup of class S for the Omni compiler with that of the Intel compiler and Guide, which is part of the KAP/Pro ToolSet of Kuck & Associates/Intel.

To analyse the DSM performance we examine the three major time consuming loops within one conjugate gradient iteration. These loops are the same in the MPI and the OpenMP/DSM implementation. They implement a sparse matrix-vector multiplication (MVM), a dot-product (DOT) , and a loop combining two AXPY operations and a dot-product. Code examples are shown in fig. 3

The sparse matrix A is stored in packed format such that indirect addressing is required for matrix operations. The sparse matrix-vector multiply is a double-nested loop requiring indirect addressing. For OpenMP, it is parallelized by using an `OMP PARALLEL DO` on the outer loop across the rows of the sparse matrix. The dot-product as well as the AXPY's combined with a dot-product are single loop nests, using the OpenMP REDUCTION clause to build the global sum.

The speedups for class S for the three major loops are shown in fig. 4. Both implementations suffer from a large communication to computation ratio for the single nested loops. However, the effect is far more severe for the DSM system. In the MPI version the communication required for the global reduction operations is highly optimized by using non-blocking send and receive to minimize synchronization overhead. The set of processes that communicate with each other is determined in advance. This allows the reduction of the amount of communication within the iteration loop. In the OpenMP/DSM implementation, processing the OpenMP `REDUCTION` clause by the DSM system generates a large communication overhead which is indicated by high number of page requests and manifestes itself by poor speedup as can be seen in fig. 4. The parallel efficiency is bad for the matrix-vector-multiply and disastrous for the dot-product and AXPY operations.

We conclude that the performance loss for the small size problems is due to:

1. additonal overhead due to the Omni compiler,
2. A high communictation to computation ratio which results from short loops and global commuincation operations.

For the more realistic benchmark class A the performance of the DSM system is acceptable.

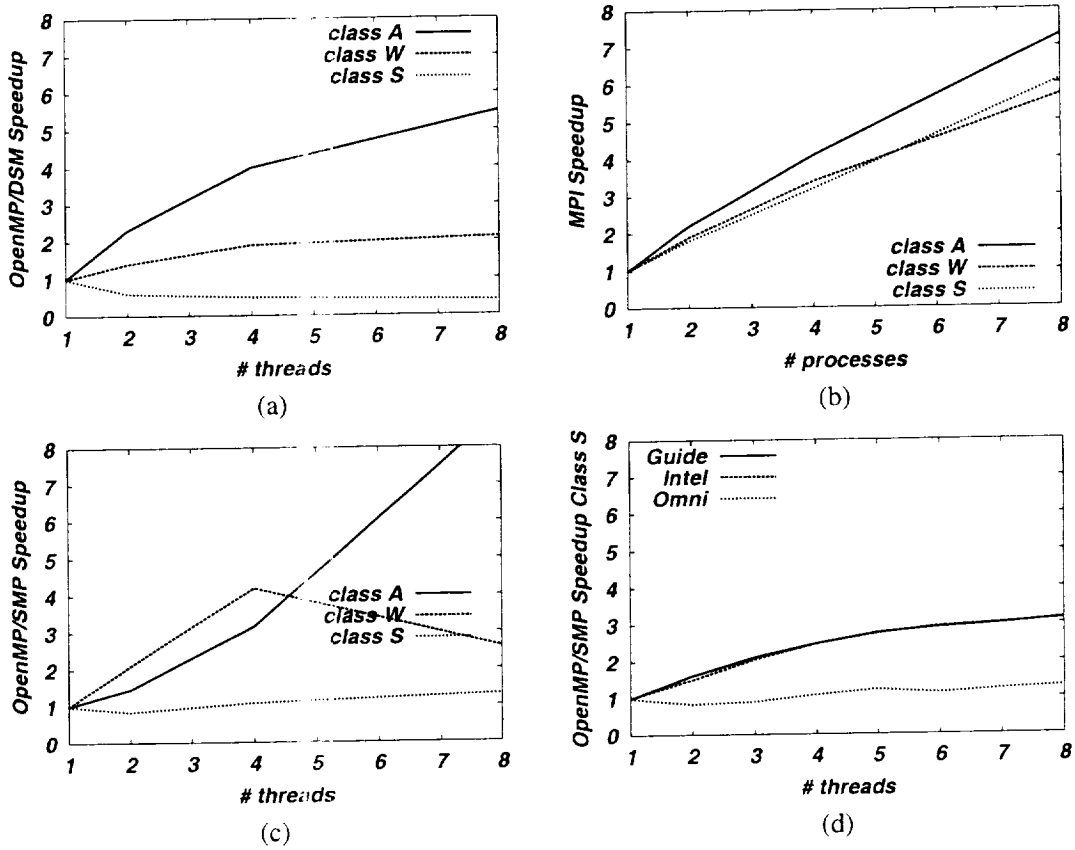


Figure 2: Speedups for different classes of the CG benchmarks. In (a) the speedup for OpenMP/DSM is shown for classes A, W and S. The MPI speedup for the same classes is given in (b). The speedup for a true shared memory system is presented in (c). (d) shows a comparison of the speedup for class S for different compilers on a shared memory platform. The Guide and the Intel compiler both support OpenMP natively.

Matrix-Vector Product:

```
!$omp parallel do default(shared) private(j,k,sum)
         do j=1,lastrow-firstrow+1
            sum = 0.d0
            do k=rowstr(j),rowstr(j+1)-1
               sum = sum + a(k)*p(colidx(k))
            enddo
            q(j) = sum
         enddo
```

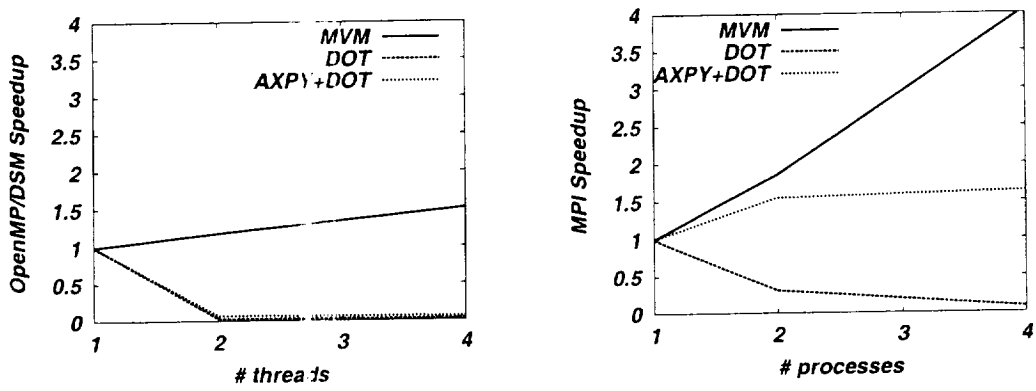
Dot-Product

```
         d = 0.0d0
!$omp parallel do default(shared) private(j) reduction(+:d)
         do j=1, lastcol-firstcol+1
            d = d + p(j)*q(j)
         enddo
```

AXPY/Dot-Product Combination

```
         rho = 0.0d0
!$omp parallel do default(shared) private(j) reduction(+:rho)
         do j=1, lastcol-firstcol+1
            z(j) = z(j) + alpha*p(j)
            r(j) = r(j) - alpha*q(j)
            rho = rho + r(j)*r(j)
         enddo
```

Figure 3: Code examples for multiplication, a dot-product, and a loop combining two AXPY operations and a dot-product



(a) (b)

Figure 4: Details of CG benchmark's class S. Speedups are shown for the matrix-vector multiplication (MVM), for the dot-product (DOT) and AXPY+dot-product (AXPY+DOT). (a) results for DSM, (b) results for MPI

4.4 The FT kernel benchmark

The FT benchmark is the computational kernel of a spectral method based on a 3-D Fast Fourier Transform (FFT). During the setup phase the 3D array is filled with random numbers. Unlike in the other benchmarks, the setup phase is part of the timed code. The serial implementation of FT code was changed to pre-calculate the values for the loop that initializes each data plane. This enables the directive based parallelization of the loop. The main loop in FT could not be parallelized completely automatically. Due to the complicated structure of the loop CAPO had to assume data dependencies that prevented parallelization. In contrast to a compiler CAPO allows interactive user guidance during the parallelization process. Parallelization could be achieved by privatizing certain arrays through the CAPO user interface.

If nx, ny, and nz denote the number of gridpoints in each of the spatial dimensisons,the sizes of the benchmark classes under consideration are given as:

Class S: nx= 64, ny= 64, nz= 64
Class W: nx=128, ny=128, nz= 32
Class A: nx=256, ny=256, nz=128

The speedup for OpenMP/DSM, MPI, and OpenMP/SMP versions for our three benchmark classes is shown in fig. 5. For 8 nodes the OpenMP/DSM implementation achieves about 70% of the MPI speedup, for class W 65% and for class S 50%. The OpenMP/DSM speedup is limited to about 4 out of 8 processes compared to 6 out of 8 for the MPI implementation. To understand the performance difference we examine the different steps of the FT benchmarks in detail. In both implementations, the 3-D FFT is accomplished by performing a 1-D FFT in each of the three spatial dimensions. For each spatial dimension the three-dimensional array is copied into a one-dimensional array, the FFT is performed on the one-dimensional array, and the result is copied back. A code fragment for the first dimension is shown fig. 6.

The OpenMP parallelization is achieved by inserting an OMP PARALLEL DO on the outermost loop. This results in a distribution of the data in dimension of K corresponding to the z-direction. The speedup for the individual three spatial dimensions for the OpenMP implementation on the class A benchmark is shown in fig. 7. While the FFT in x and y dimension reach a speedup of 6 out of

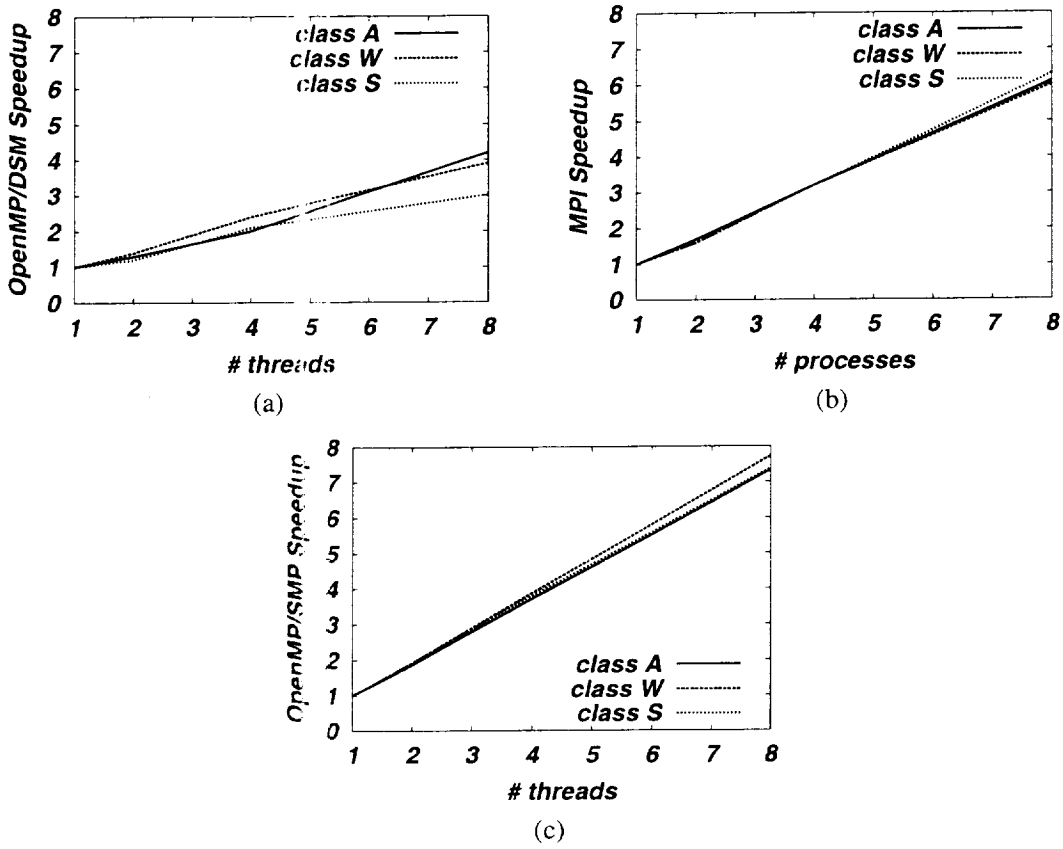


Figure 5: Comparison of MPI and OpenMP/DSM speedups for classes A, W and S of the FT benchmark. (a) Speedup for OpenMP/DSM, (b) MPI Speedup, (c) Speedup on the SMP system

```
do k = 1, n3
      do j = 1, n2
         do i = 1, n1
            w(i) = u(i,j,k)
         enddo

         call fft (w,...)

         do i = 1, d(1)
            u(i,j,k) = w(i)
         enddo
   enddo
enddo
```

Figure 6: Code fragment for the first dimension of FFT

8, the speedup in z-dimension is only 2 out of 8. The performance loss in X and Y dimension is mostly due to communication caused by writing to the shared array U which is indicated by page requests within this loop. Logically there is no communication required for this loop, since only the local part of the array is accessed. The performance decrease for the z-dimension is due to the fact that here the outermost loop of the loop nest from fig. 6 runs in J and not in K dimension. Since the data was distributed in K dimension, parallel execution of the loop requires access to remote data and causes a large number of page requests. The MPI implementation performs a transpose of the three-dimensional array in z dimension, which is achieved by a call to `MPI_ALLTOALL`. This causes some decrease in performance, but not as severe as in the DSM system.

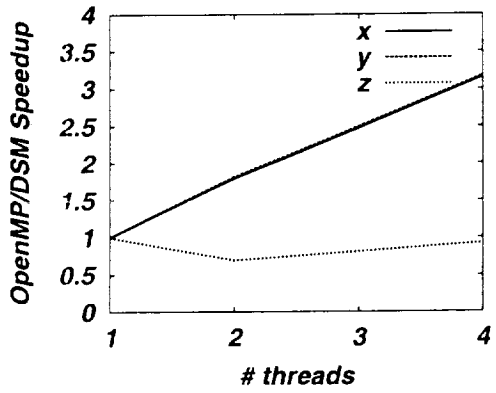


Figure 7: Speedup for different directions of the FFT on the DSM system

5 Problems encountered

The installation of SCore, SCASH and Omni OpenMP was rather straight forward. For the basic SCore installation we tried to use aggressive compiler optimizations whenever possible and we went through an iterative process to find a stable configuration in terms of compiler settings. The SCASH and Omni OpenMP configurations were based on the one found for the basic SCore system. We were able to run all tests and examples delivered with either SCASH or the Omni OpenMP compiler suite successfully.

We ran into problems when trying to run the three kernel benchmarks EP, CG, and FT for larger problem sizes such as they are given class B or C. We also could not run any of the simulated CFD applications BT, SP, and LU that are part of the benchmark suite, even for the small problem size given in class A. The problems we encountered were due to the fact that SCASH was not able to allocate enough of virtual memory. The SCASH system itself uses a large amount of memory for its own memory managment on top of the one provided by the operating sysstem. To improve data exchange performance (i.e. bandwidth and latency) SCASH specifically allocates pin-down memory [21]. For larger benchmark classes it seems that there is not enough pinnable memory available.

Another severe restriction is the 32 bit address-space of the IA32 architecture. With 32 bit addresses the address-space is restricted to at most 2^{32} addresses. Usually the memory managment of

operating systems like Linux or Windows [1] allows a process to use only part of this address-space for its private data. The operating system uses the rest to mirror some internal data structures into the process' virtual address-space. Under Linux a process can only use 2 GB of the theoretical maximum of 4 GB for its private data.

Without additional effort, the kernel itself would suffer from this 4 GB barrier. To enable the use of more main memory, on IA32 Linux uses the PAE capabilities of modern processors to access up to 64 GB. This is achieved by having a three stage page address translation mechanism. But even with this system, only the kernel can handle more than 4 GB. A single process is still restricted to 2 GB of private memory.

A software distributed shared memory system like SCASH that runs in user-mode and uses a 32 bit global address-space will therefore be restricted to a maximum of 4 GB global shared memory.

6 Related Work

Another system supporting the OpenMP paradigm on distributed memory systems is TreadMarks [2]. Comparisons of the TreadMarks systems with message passing programming are given in [7] and [11]. Other systems that support software DSM programming are Cashmere [17] and SMP-Shasta [?] There are a number of papers reporting on comparisons of different programming paradigms. As an example we name [15] and [16] where message passing and shared memory programming are compared on shared memory architectures.

7 Conclusions and Future Work

We have measured the performance of OpenMP/DSM implementations of three of the NAS Parallel Benchmarks on a commodity cluster of PCs, and we compared the speedup to corresponding MPI implementations of the same algorithms. The difference in performance depends on the structure of the application and the problem size. For the largest problem sizes under consideration the observed OpenMP/DSM speedups range between 100% and 70% of the MPI speedup for all benchmarks. Only in cases whith an extremely high communication to computation ratio does the OpenMP/DSM speedup go down to less than 10% of MPI. This occurs in the smallest class of the CG benchmark, where AXPY and dot-product operations for short vector lengths are being parallelized. We have noticed that in this extreme case part of the performance decrease was due to compiler defficiencies which also show on a shared memory system. The memory problems described in section 5 are implementation dependent and we expect them to be resolved in commercial software. Usage of 64 bit system sofwtare and kernel enhancements to support DSM on a system level will improve the general usability of DSM systems.

All in all we are encouraged by the results we obtained considering the fact that we were using public domain software. The DSM system allowed us to take exploit parallelism over all nodes of the cluster by using automatically parallelized code based on OpenMP. We find the performance differences when compared with hand-optimized MPI code acceptable when we take into account the extremely short developement time of the parallel code. Our future plan is to run full size aplications in our testbed environment.

[1] Windows is a registered trademark of Microsoft Corp.

8 Acknowledgments

We would like to thank Jahed Djohmeri and Rob van der Wijngaart of NAS for reviewing the paper and the suggestions they made for improving it. The authors also wish to thank Rob van der Wijngaart for many helpful discussions about the NAS Parallel Benchmarks and DSM systems. Part of this work was supported by NASA contracts NAS 2-14303 and DTTS59-99-D-00437/A61812D with Computer Sciences Corporation.

References

[1] http://www.conservativecomputer.com/myrinet/perf.html.

[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] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NASA Ames Research Center, Moffett Field, CA, 1991.

[4] D. Bailey, T. Harris, W. Saphir, R van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Moffett Field, CA, 1995. http://www.nas.nasa.gov/Software/NPB.

[5] Phillip Ezolt. A Study in Malloc: A Case of Excessive Minor Faults. In *Proceedings of the 5th Annual Linux Showcase & Conference, November 5–10*, 2001.

[6] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic Home Node Reallocation on Software Distributed Shared Memory. In *Proceedings of HPC Asia 2000, Beijing, China*, pages 158–163, May 2000.

[7] Y. C. Hu, H. Lu A. L. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. In *Proceedings of the Thirteenth International Parallel Processing Symposium*, pages 302–310, 1999.

[8] C. S. Ierotheou, S. P. Johnson, M. Cross, and P. F. Leggett. Computer Aided Parallelisation Tools (CAPTools)-Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes. *Parallel Computing*, 22:163–195, 1996.

[9] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementations of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NAS, 1999.

[10] H. Jin, M. Frumkin, and J. Yan. Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes. In *Proceedings of Third International Symposium on High Performance Computing (ISHPC2000), Tokyo, Japan, October 16-18*, 2000.

[11] H. Lu, S. Dwarkdadas, A. L. Cox, and W. Zwaenepoel. Quantifying the Performance Differences Between PVM and TreadMarks. *Journal of Parallel and Distributed Computation*, 43(2):65–78, June 1997.

[12] *MPI 1.1 Standard*. http://www-unix.mcs.anl.gov/mpi/mpich.

[13] *Omni OpenMP and SCASH*. http://www.pccluster.org.

[14] *OpenMP Fortran Application Program Interface*. `http://www.openmp.org`.

[15] H. Shan and J. Pal Singh. A comparison of MPI,SHMEM, and Cache-Coherent Shared Address Space Programming Models on a Tightly-Coupled Multiprocessor. *International Journal of Parallel Programming*, 29(3), 2001.

[16] H. Shan and J. Pal Singh. Comparison of Three Programming Models for Adaptive Applications on the Origin 2000. *Journal of Parallel and Distributed Computing*, 62:241–266, 2002.

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

[18] K. Taura, S. Matsuoka, and A. Yonezawa. StackThreads: An abstract machine for scheduling fine-grain threads on stock CPUs. In *Proceedings of Workshop on Theory and Practice of Parallel Programming*, pages 121–136, 1994.

[19] H. Tezuka, A. Hori, and Y. Ishikawa. Design and Implementation of PM: a Communication Library for Workstation Cluster. In *JSPP'96, IPSJ*, pages 41–48, June 1996. (In Japanese).

[20] H. Tezuka, A. Hori, and Y. Ishikawa. PM: A High-Performance Communication Library for Multi-user Parallel Environments. Technical Report TR-96015, RWC, November 1996.

[21] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. Technical Report TR 97006, Tsukuba Research Center, Real World Computing Partnership, 1997.

Experiences using OpenMP based on Compiler Directed Software DSM on a PC Cluster

M. Hess*, G. Jost, M. Mueller*, R. Ruehle***

**High Performance Computing Center, Stuttgart (HLRS)*

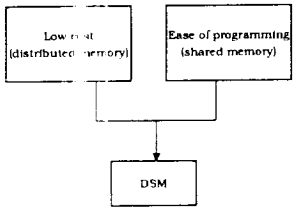
*** CSC/NASA Ames Research Center*

Distributed vs Shared Memory Parallelism

- **Distributed Memory**
  - Commodity hardware
  - Commodity network
  - Low cost alternative for high end scientific computing
  - Currently difficult to program:
    - Data distribution
    - Message Passing required
- **Shared Memory**
  - Globally shared address space
  - Parallelization via compiler directives
  - Incremental parallelization possible
  - High cost of hardware
  - Limited scalability

Distributed Shared Memory (DSM)

- **Software for distributed memory architecture**
- **Enables shared memory programming**
- **Combines**
- **Examples are:**
  - TreadMarks
  - Scash



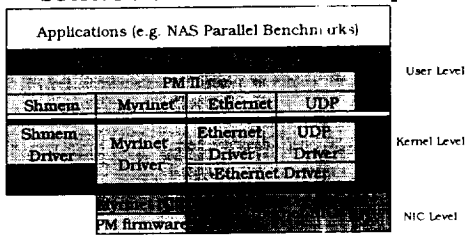
Performance?

Test Environment: Hardware

- **PC Cluster at HLRS**
- **8 NEC 120Ed server nodes.**
- **Each Node:**
  - Dual processor
  - 1 GHz Pentium III
  - 2 GB main memory
- **Network:**
  - Myrinet 2000 NIC
  - 64 bit/66 MHz PCI slot
- **Bandwidth:**
  - 409 MB/s read
  - 480 MB/s write

Test Environment: System Software (1)

- **SCore: PC Cluster Consortium, Japan**



Test Environment System Software (2)

- **SCash:**
  - PC Cluster Consortium
  - Software Distributed Shared Memory System.
  - Based on a high bandwidth communication library (PM)
  - Maintains page based memory consistency
  - Multiple writer release consistency model:
    - Modified pages are transferred at synchronization points (e.g.barriers)

Test Environment System Software (3)

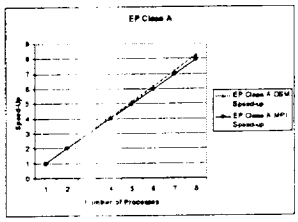
- **Omni Compiler:**
  - OpenMP compiler with C and Fortran front end for SMP
  - SCash based Omni Compiler

Case Studies

- **Three kernel benchmarks from the NAS Parallel Benchmark suite:**
  - Message passing implementation based on MPI (NPB2.3)
  - Automatically parallelized OpenMP code using the CAPO parallelization tool
- **Evaluation strategy:**
  - Run different problem sizes
  - Compare speedup to corresponding message passing implementations
  - Compare speedup to a true SMP system:
    - Same operating system
    - Omni compiler, but not using SCASH
  - Take into account development time for parallel code

The EP Benchmark

- **Embarrassing Parallel:**
  - Generation of random numbers
  - Loop iterations parallel.
  - Global sum reduction at the end
- **MPI implementation:**
  - Global sum built via MPI_ALLREDUCE
  - Low communication overhead (< 1%)
- **OpenMP/DSM:**
  - Little memory access.
  - OMP DO PARALLEL
  - OMP DO REDUCTION



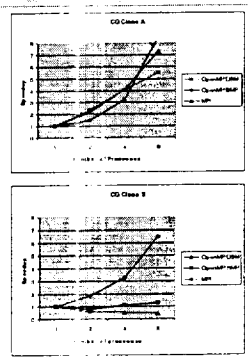
Linear speedup for MPI and OpenMP/DSM. No surprises.

The CG Benchmark

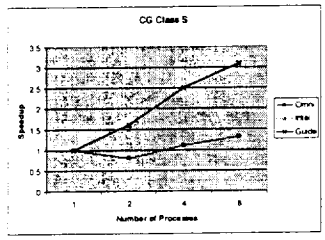
- **Conjugate gradient method to solve an eigenvalue problem**
- **Stresses irregular data access**
- **Major loops:**
  - Sparse Matrix-Vector-Multiply
  - Dot-Product
  - AXPY Operations
- **Same major loops in MPI and OpenMP implementation**
- **Automatic parallelization without user interaction**

CG Benchmark Results (1)

- **OpenMP/DSM efficiency about 75% of that of MPI for Class A**
- **OpenMP/DSM performance bad for Class S:**
  - Inefficiencies in the Omni Compiler
  - Large Communication overhead:
    - Short loops with few calculations
    - Global reduction operations



CG Benchmark Results (2)

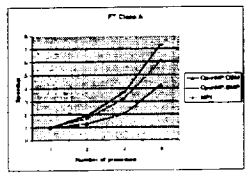


The FT Benchmark

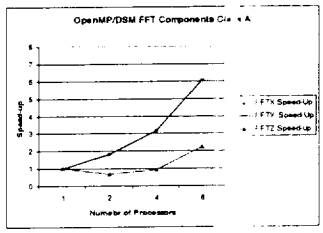
- **Kernel of spectral method based on 3D Fast Fourier Transform (FFT)**
- **3D FFT achieved by a 1D FFT in x, y, and z direction**
- **OpenMP parallelization required some user interaction**

FT Benchmark Results (1)

- **OpenMP/DSM efficiency about 70% of MPI**
- **MPI Parallelization:**
  - MPI_ALLTOALL to achieve transpose of data
- **OpenMP Parallelization:**
  - OMP DO PARALLEL on outer loop
  - Extra communication introduced by DSM system (false page sharing)
  - Remote data access required for FFT in z-dimension



FT Benchmark Results (2)



Problems Encountered

- **Limited Pin-able memory available**
- **Private memory limited to 2GB**
- **Need for:**
  - Enhanced kernel support
  - 64 bit addressing mode

Conclusions:

- OpenMP/DSM delivered acceptable speedup if the communication/computation ratio is not too high
- OpenMP/DSM showed between 70% and 100% of MPI efficiency for benchmarks of Class A
- Large cases could not be run due to memory problems

Related Work:

- TreadMarks
- Cashmere
- SMP-Shasta

Future Work:

- Run full applications under DSM
- Try commercial DSM software once it becomes available (I. E. KAI /Pro Toolset Network Edition)