Implementing an OpenMP Execution Environment on InfiniBand Clusters

Jie Tao[1], Wolfgang Karl[1], and Carsten Trinitis[2]

[1]Institut für Rechnerentwurf und Fehlertoleranz
Universität Karlsruhe (TH)
76128 Karlsruhe, Germany
E-mail: {tao,karl}@ira.uka.de
[2]Lehrstuhl für Rechnertechnik und Rechnerorganisation
Technische Universität München
Boltzmannstr.3, 85748 Garching, Germany
trinitic@cs.tum.edu

Abstract. Cluster systems interconnected via fast interconnection networks have been successfully applied to various research fields for parallel execution of large applications. Next to MPI, the conventional programming model, OpenMP is increasingly used for parallelizing sequential codes. Due to its easy programming interface and similar semantics with traditional programming languages, OpenMP is especially appropriate for non-professional users.
For exploiting scalable parallel computation, we have established a PC cluster using InfiniBand, a high-performance, de facto standard interconnection technology. In order to support the users with a simple parallel programming model, we have implemented an OpenMP execution environment on top of this cluster. As a global memory abstraction is needed for shared data, we first built a software distributed shared memory implementing a kind of Home-based Lazy Release Consistency protocol. We then modified an existing OpenMP source-to-source compiler for mapping shared data on this DSM and for handling issues with respect to process/thread activities and task distribution. Experimental results based on a set of different OpenMP applications show a speedup of up to 5.22 on systems with 6 processor nodes.

1 Motivation

Clusters are regarded as adequate platforms for exploring high performance computing. In contrast to tightly-coupled multiprocessor systems, like SMPs, clusters have the advantage of scalability and cost-effectiveness. Therefore, they are generally deployed in a variety of both research and commercial areas for performing parallel computation.

As a consequence, we have also established a cluster system using modern processors. More specifically, this cluster is connected via InfiniBand [7], a high-performance interconnect technology. Besides its low latency and high bandwidth, InfiniBand supports Remote Data Memory Access (RDMA), allowing access to remote memory locations via the network without any involvement of the receiver. This feature allows InfiniBand to achieve higher bandwidth for inter-node communication, in comparison with other interconnect technologies such as Giga-Ethernet and Myrinet.

As the first step towards cluster computing, we have built an MPI environment on top of this cluster. However, we note that increasingly, users have no special knowledge about parallel computing and they usually bring OpenMP codes. Since it offers an easier programming interface with semantics similar to that of sequential codes, OpenMP is preferred by non-professional users to develop parallel programs. In order to support these users, we established the OpenMP execution environment on top of our InfiniBand clusters.

As a global memory abstraction is the basis for any shared memory programming model, we first developed ViSMI (Virtual Shared Memory for InfiniBand clusters), a software-based distributed shared memory. ViSMI implements a kind of home-based lazy release consistency model and provides annotations for dealing with issues with respect to parallel execution, such as process creation, data allocation, and synchronization. We then developed Omni/Infini, a source-to-source OpenMP compiler using ViSMI as the supporting interface. Omni/Infini is actually an extended version of the Omni compiler. We have modified Omni in order to replace the thread interface with ViSMI interface, to map shared data on the distributed shared memory, and to coordinate the work of different processes.

The established OpenMP execution environment has been verified using both applications from standard benchmark suites, like NAS and SPLASH-II, and several small kernels. Experimental results show different behavior with applications. However, for most applications, scalable speedup has been achieved.

The remainder of this paper is organized as follows. Section 2 gives an introduction to the InfiniBand cluster and the established software DSM. This is followed by a brief description of Omni/Infini in Section 3. In Section 4 first experimental results are illustrated. The paper concludes with a short summary and some future directions in Section 5.

2 The InfiniBand Cluster and the Software DSM

InfiniBand [7] is a point-to-point, switched I/O interconnect architecture with low latency and high bandwidth. For communications, InfiniBand provides both channel and memory semantics. While the former refers to traditional send/receive operations, the latter allows the user to directly read or write data elements from or to the virtual memory space of a remote node without involving the remote host processor. This scenario is referred to as Remote Direct Memory Access (RDMA).

The original configuration of our InfiniBand cluster included 6 Xeon nodes and 4 Itanium 2 (Madison) nodes. Recently we have added 36 Opteron nodes into the cluster. The Xeon nodes are used partly for interactive tasks and partly for computation, while the others are purely used for computation. These processor nodes are connected through switches with a theoretical peak bandwidth of 10 Gbps.

As the first step towards an infrastructure for shared memory programming, we implemented ViSMI [16], a software-based distributed shared memory system.

The basic idea behind software distributed shared memory is to provide the programmers with a virtually global address space on cluster architectures. This idea is first proposed by Kai Li [13] and implemented in IVY [14]. As the memories are actually distributed across the cluster, the required data could be located on a remote node and also multiple copies of shared data could exist. The latter leads to consistency issues, where a write operation on shared data has to be seen by other processors. For tackling this problem, software DSMs usually rely on the page fault handler of the operating system to implement invalidation-based consistency models.

The concept of memory consistency models is to precisely characterize the behavior of the respective memory system by clearly defining the order in which memory operations are performed. Depending on the concrete requirement, this order can be strict or less strict, hence leading to various consistency models.

The most strict one is sequential consistency [12], which forces a multiprocessor system to achieve the same result of any execution as if the operations of all the processors were executed in some sequential order and the operations of each individual processor appear in the order specified by its programmers. This provides an intuitive and easy-to-follow memory behavior, however, the strict ordering requires the memory system to propagate updates early and prohibits optimizations in both hardware and compilers. Hence, other models have been proposed to relax the constraints of sequential consistency with the goal of improving the overall performance.

Relaxed consistency models [3, 6, 9, 10] define a memory model for programmers to use explicit synchronization. Synchronizing memory accesses are divided into *Acquires* and *Releases*, where an *Aquire* allows the access to shared data and ensures that the data is up-to-date, while *Release* relinquishes this access right and ensures that all memory updates have been properly propagated. By separating the synchronization in this way invalidations are only performed by a synchronization operation, therefore reducing the unnecessary invalidations caused by an early coherence operation.

A well-known relaxed consistency model is Lazy Release Consistency (LRC) [10]. Within this model, invalidations are propagated at the acquire time. This allows the system to delay communication of write updates until the data is actually needed. To reduce the communications caused by false sharing, where multiple unrelated shared data locate on the same page, LRC protocols usually support a multiple-writer scheme. Within this scheme, multiple writable copies of the same page are allowed and a clean copy is generated after an invalidation. Home-based Lazy Release Consistency (HLRC) [17], for example, implements such a multiple-writer scheme by specifying a home for each page. All updates to a page are propagated to the home node at synchronization points, such as lock release and barrier. Hence the page copy on home is up-to-date.

ViSMI implements such a Home-based Lazy Release Consistency protocol. For each shared page a default home is specified during the initialization phase and then the node first accessing the page becomes its home. Each processor can maintain a copy of the shared page, but by a synchronization operation all copies are invalidated. Also at this point, an up-to-date version of the page is created on the home node. For this, the updates of all processors holding a copy must be aggregated. ViSMI uses a *diff*-based mechanism, where the difference (*diffs*) between each dirty copy and the clean copy is computed. This is similar to that used by the Myrias parallel do mechanism [2]. To propagate the updates, ViSMI takes advantage of the hardware-based multicast provided by InfiniBand to minimize the overheads for interconnection traffic. The *diffs* are then applied to the clean copy and the up-to-date version of the page is generated. For further computation page fault signals are issued on other processors and the missing page is fetched from the home node. To handle the incoming communication, each node maintains an additional thread, besides the application thread. This communication thread is only active when a communication request occurs. We use the event notification scheme of InfiniBand to achieve this.

For parallel execution, ViSMI establishes a programming interface for developing shared memory applications. This interface is primarily composed of a set of annotations that handle issues with respect to parallelism. The most important annotations and a short description about them are listed in Table 1.

Annotation	Description
HLRC_Malloc	allocating memory in shared space
HLRC_Myself	querying the ID of the calling process
HLRC_InitParallel	initialization of the parallel phase
HLRC_Barrier	establishing synchronization over processes
HLRC_Acquire	acquiring the specified lock
HLRC_Release	releasing the specified lock
HLRC_End	releasing all resources and terminating

Table 1. ViSMI annotations for shared memory execution.

3 Omni/Infini: Towards OpenMP Execution on Clusters

OpenMP is actually initially introduced for parallel multiprocessor systems with physically global shared memory. Recently, compiler developers have been extending the existing compilers to enable the OpenMP execution on cluster systems, often using a software distributed shared memory as the basis. Well-known examples are the Nanos Compiler [5, 15], the Polaris parallelizing compiler [1], and the Omni/SCASH compiler [18].

Based on ViSMI and its programming interface, we similarly implemented an OpenMP compiler for the InfiniBand cluster. This compiler, called Omni/Infini, is actually a modification and extension of the original Omni compiler for SMPs [11]. The major work has been done with coordination of processes, allocation of shared data, and a new runtime library for parallelization, synchronization, and task scheduling.

Process structure vs. thread structure. The Omni compiler, like most others, uses a thread structure for parallel execution. It maintains a master thread and a number of slave threads. Slave threads are created at the initialization phase, but they are idle until a parallel region is encountered. This indicates that sequential regions are implicitly executed by the master thread, without any special task assignment. The ViSMI interface, on the other hand, uses a kind of process structure, where processes are forked at the initialization phase. These processes execute the same code, including both sequential regions and parallel parts. Clearly, this structure burdens processors with unnecessary work. In order to maintain the conventional OpenMP semantics with parallelism and also to save the CPU resources, we have designed special mechanisms to clearly specify which process does what job. For code regions needed to be executed on a single processor, for example, only the process on the host node is assigned with tasks.

Shared data allocation and initialization. ViSMI maintains a shared virtual space visible to all processor nodes. This space is reserved at the initialization phase and consists of memory spaces from each processor's physical main memory. Hence, all shared data in an OpenMP code must be allocated into this virtual space in order to be globally accessible and consistent. This is an additional work for a cluster-OpenMP compiler. We extended Omni for detecting shared variables and further changing them to data objects which will be allocated to the shared virtual space at runtime.

Another issue concerns the initialization of shared variables. Within a traditional OpenMP implementation, this is usually done by a single thread. Hence, an OpenMP directive SINGLE is often applied in case that such initialization occurs in a parallel region. This causes problems when running the applications on top of ViSMI. ViSMI allocates memory spaces for shared data structures on all processor nodes [1]. These data structures must be initialized before further use for parallel computation. Hence, the initialization has to be performed on all nodes. Currently, we rely on an implicit barrier operation inserted to SINGLE to tackle this problem. With this barrier, updates to shared data are forced to aggregate on the host node and a clean copy is created. This causes performance lost because a barrier operation is not essential for all SINGLE operations. For the next version of Omni/Infini, we intend to enable compiler-level automatic distinction between different SINGLE directives.

Runtime library. Omni contains a set of functions to deal with runtime issues like task scheduling, lock and barrier, reduction operations, environment variables, and specific code regions such as MASTER, CRITICAL, and SINGLE. These functions require information, like thread ID number and number of threads, to perform correct actions for different threads. This information

[1] ViSMI allocates on each processor a memory space for shared variables. Each processor uses the local copy of shared data for computation.

is stored within data structures for threads, which are not available in ViSMI. Hence, we modified all related functions in order to remove the interface to thread structure of Omni and to build the connection to the ViSMI programming interface. In this way, we created a new OpenMP runtime library that is linked to the applications for handling OpenMP runtime issues.

4 Initial Experimental Results

Based on the extension and modification with both sides, the Omni compiler and ViSMI, we have developed this OpenMP execution environment for InfiniBand clusters. In order to verify the established environment, various measurements have been done using our InfiniBand cluster. Since ViSMI is currently based on a 32-bit address space and the Opteron nodes are still in the test phase, the experiments could only be carried out on the six Xeon nodes. We use a variety of applications for examining different behavior. Four of them are chosen from the NAS parallel benchmark suite [4, 8] and the OpenMP version of the SPLASH-2 Benchmark suite [20]. Two Fortran programs are selected from the benchmark suite developed for an SMP programming course [19]. In addition, two self-coded small kernels are also examined. A short description, the working set size, and the required shared memory size of these applications are shown in Table 2.

Application	Description	Working set size	Shared memory size	Benchmark
LU	LU-decomposition for dense matrices	2048×2048 matrix	34MB	SPLASH-2
Radix	Integer radix sort	18.7M keys	67MB	SPLASH-2
FT	Fast Fourier Transformations	64×64×64	51MB	NAS
CG	Grid computation and communication	1400	3MB	NAS
Matmul	Dense matrix multiplication	2048×2048	34MB	SMP course
Sparse	Sparse matrix multiplication	1024×1024	8MB	SMP course
SOR	Successive Over Relaxation	2671×2671	34MB	self-coded
Gauss	Gaussian Elimination	200×200	1MB	self-coded

Table 2. Description of benchmark applications.

First, we measured the speedup of the parallel execution using different number of processors. Figure 2 shows the experimental results. It can be seen that applications behave quite differently. LU achieves the best performance with a scalable speedup of as high as 5.52 on a 6-node cluster system. Similarly, Matmul and SOR also show a scalable speedup with close parallel efficiency on different systems (efficiency is calculated with the speedup divided by the number of processors and reflects the scalability of a system). Sparse and Gauss behave poorly, with either no speedup or running even slower on multiprocessor systems. This is caused by the smaller working set size of both codes. Actually, we selected this size in order to examine how system overhead influences the parallel performance; and we see that due to the large percentage of overhead in the overall execution time, applications with smaller data size can not gain speedup on systems with software-based distributed shared memory.

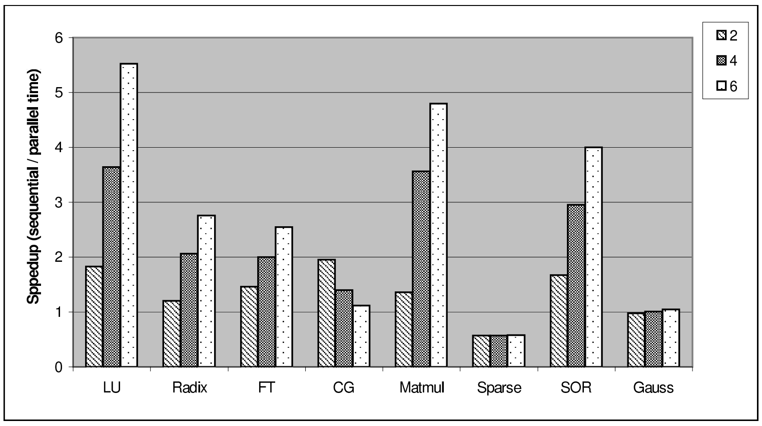

Fig. 1. Speedup on systems with different number of processors.

CG behaves surprisingly with a decreasing speedup as the number of processors increases. For detecting the reasons, we measured the time needed for different activities when running the applications on 6-node systems. Figure 3 shows the experimental results.

In Figure 3, *exec.* denotes the overall execution time, while *comput.* specifies the time for actually executing the application, *page* is the time for fetching pages, *barrier* denotes the time for barrier operations [2], *lock* is the time for performing locks, *handler* is the time needed by the communication thread, and *overhead* is the time for other protocol activities. The sum of all partial times equals to the total *exec.* time.

It can be seen that LU, Matmul, and SOR show a rather high proportion in computation time, and hence achieve better speedup than other applications. Radix and FT behaves worse than them, but most of the time is used for calculation. For Sparse and Gauss roughly only the half time is spent for computation and hence nearly no speedup can be observed. The worst case is with CG, where only 33% of the overall time is used for running the program and more time is spent on other activities like inter-node communication and synchronization. As it is a fact that each processor introduces such overhead, slowdown can be caused with more processors running the code. CG has shown this behavior.

In order to further verify this, we measured the time for different activities with CG also on 2-node and 4-node systems. Figure 4 shows the experimental results. It can be seen that while the time with *handler* and *overhead* is close on different systems, *page* and *barrier* show a drastic increase with more processors on the system. As a result, a decreasing speedup has been observed.

In addition, Figure 3 also shows a general case where page fetching and barrier operations introduce the most overhead. In order to further examine these critical issues, we measured the concrete number of page faults, barriers, and locks. Table 3 depicts the experimental results.

[2] This includes the time for synchronization and that for transferring *diffs*.

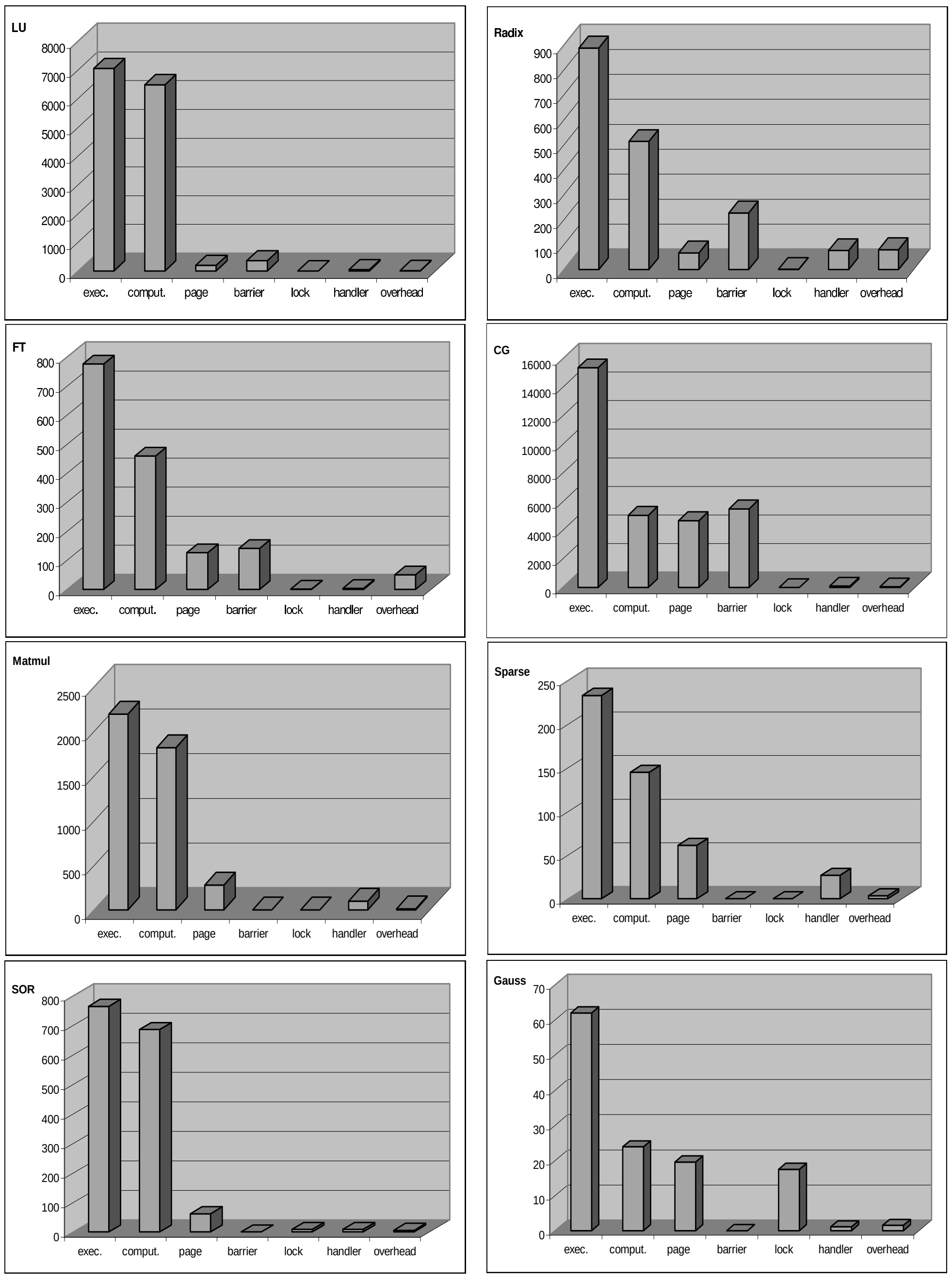

Fig. 2. Normalized execution time breakdown of tested applications.

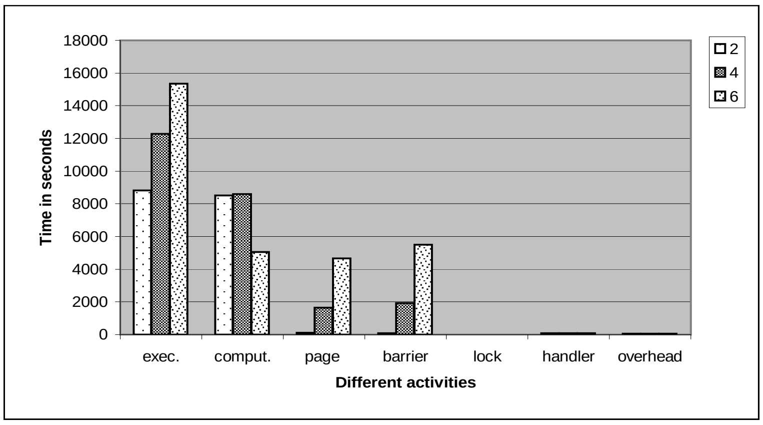

Fig. 3. Execution time of different activities on 2-node and 4-node systems (CG code).

This table shows the number of page fault, barrier operation, and locks. It also gives the information about data and barrier traffic over processors. All applications, except Gauss with smaller working set, show high number of page fault, and hence the large amount of resulted data transfer. In contrast, only fewer barriers have been measured, except the CG code. However, a barrier operation could introduce significant overhead, since all processors have to send updates to the home node, which introduces transfer overhead, and to wait for a reply, which causes synchronization overhead. In addition, the computation can continue only after a clean copy has been created. Therefore, even though only a few of barriers are performed, still large proportion of the overall time is spent on barrier operations, as having been illustrated in Figure 3. However, LU is an exception, where few time is needed for barriers. This can be explained by the fact that with LU rather small amount of *diff*s are created and hence overhead for data transfer at barriers is small.

	page fault	barriers	lock acquired	data traffic	barrier traffic
LU	5512	257	0	14.4MB	0.015M
Radix	6084	10	18	13.9MB	0.12M
FT	3581	16	49	21MB	0.03M
CG	8385	2496	0	8.3MB	0.03M
Matmul	4106	0	0	17.4MB	0
Sparse	1033	0	0	1.4MB	0
SOR	2739	12	0	4.7MB	0.006M
Gauss	437	201	0	0.17MB	0.02M

Table 3. Value of several performance metrics.

In order to reduce the overhead with page fault and barrier, we propose adaptive approaches. Actually, page fault occurs when a page copy has to be invalidated, while a clean copy of this shared page is created. The current HLRC implementation uses an approach, where all processors send the *diffs* to the home node. The home node then applies the *diffs* to the page and generates a clean copy. After that, other processors can fetch this copy for further computation. A possible improvement to this approach is to multicast the *diffs* to all processors and apply them directly on all dirty copies. In this way, the number of page fault and the resulted data transport could be significantly reduced, while at the same time the overhead for multicasting is much smaller due to the special hardware-level support of InfiniBand. This optimization can also reduce the overhead for barriers, because in this case synchronization is not necessary; rather a processor can go on with the computation, as soon as the page copy on it is updated. We will implement such optimizations in the next step of this research work.

The last experiment was done with the laplace code provided by the Omni compiler. The goal is to compare the performance of OpenMP execution on InfiniBand clusters with that of software DSM based OpenMP implementation on clusters using other interconnection technologies. For the latter, we apply the data measured by the Omni/SCASH researchers on both Ethernet and Myrinet clusters. Figure 5 gives the speedup on 2, 4, and 6 node systems.

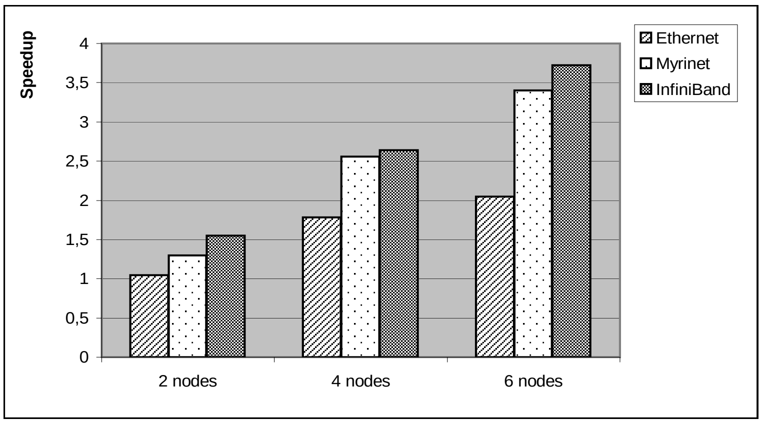

Fig. 4. Speedup comparison with other SDSM-based OpenMP implementation.

It can be seen that our system provides the best performance, with an average improvement of 62% to Ethernet and 10% to Myrinet. This improvement shall be contributed by the specific properties of InfiniBand.

5 Conclusions

OpenMP is traditionally designed for shared memory multiprocessors with physically global shared memory. Due to the architectural restriction, however, such machines suffer from scalability. On the other hand, cluster systems are widely used for parallel computing, raising the need for establishing OpenMP environments on top of them.

In this paper, we introduce an approach for building such an environment on InfiniBand clusters. First, a software DSM is developed, which creates a shared virtual memory space visible to all processor nodes on the cluster. We then modified and extended the Omni OpenMP compiler in order to deal with issues like data mapping, task scheduling, and the runtime. Experimental results based on a variety of applications show that the parallel performance depends on applications and their working set size. Overall, a speedup of up to 5.22 on 6 nodes has been achieved.

Besides the optimization with page fault and barrier, we also intend to apply more features of InfiniBand to further reduce the system overhead. In addition, the software DSM will be extended to 64-bit address space, allowing a full use of the whole cluster for OpenMP execution.

References

1. A. Basumallik, S.-J. Min, and R. Eigenmann. Towards OpenMP Execution on Software Distributed Shared Memory Systems. In *Proceedings of the 4th International Symposium on High Performance Computing (ISHPC 2002)*, pages 457–468, 2002.
2. M. Beltrametti, K. Bobey, and J. R. Zorbas. The Control Mechanism for the Myrias Parallel Computer System. *ACM SIGARCH Computer Architecture News*, 16(4):21–30, 1988.
3. A. L. Cox, S. Dwarkadas, P. J. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.
4. D. Bailey et. al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, Department of Mathematics and Computer Science, Emory University, March 1994.
5. Marc Gonzàlez, Eduard Ayguadé, Xavier Martorell, Jesús Labarta, Nacho Navarro, and José Oliver. NanosCompiler: Supporting Flexible Multilevel Parallelism in OpenMP. *Concurrency: Practice and Experience*, 12(12):1205–1218, 2000.
6. L. Iftode and J. P. Singh. Shared Virtual Memory: Progress and Challenges. In *Proceedings of the IEEE, Special Issue on Distributed Shared Memory*, volume 87, pages 498–507, 1999.
7. InfiniBand Trade Association. InfiniBand Architecture Specification, Volume 1, November 2002.
8. H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.
9. P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory On Standard Workstations and Operating Systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, January 1994.
10. P. J. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Department of Computer Science, Rice University, January 1995.

11. K. Kusano, S. Satoh, and M. Sato. Performance Evaluation of the Omni OpenMP Compiler. In *Proceedings of International Workshop on OpenMP: Experiences and Implementations (WOMPEI)*, volume 1940 of LNCS, pages 403–414, 2000.
12. L. Lamport. How to Make a Multiprocessor That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):241–248, 1979.
13. K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, September 1986.
14. K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing, Vol. II Software*, pages 94–101, 1988.
15. Xavier Martorell, Eduard Ayguadé, Nacho Navarro, Julita Corbalán, Marc González, and Jesús Labarta. Thread Fork/Join Techniques for Multi-Level Parallelism Exploitation in NUMA Multiprocessors. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 294–301, Rhodes, Greece, June 1999.
16. C. Osendorfer, J. Tao, C. Trinitis, and M. Mairandres. ViSMI: Software Distributed Shared Memory for InfiniBand Clusters . In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (IEEE NCA04)*, pages 185–191, September 2004.
17. M. Rangarajan and L. Iftode. Software Distributed Shared Memory over Virtual Interface Architecture: Implementation and Performance. In *Proceedings of the 4th Annual Linux Showcase, Extreme Linux Workshop*, pages 341–352, Atlanta, USA, October 2000.
18. M. Sato, H. Harada, and A. Hasegawa. Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. *Scientific Programming*, 9(2-3):123–130, 2001.
19. R. K. Standish. SMP vs Vector: A Head-to-head Comparison. In *Proceedings of the HPCAsia 2001*, September 2001.
20. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.