First Experiences with Intel Cluster OpenMP

Christian Terboven Dieter an Mey Dirk Schmidl
Marcus Wagner
RWTH Aachen University, Center for Computing and Communication
Seffenter Weg 23, 52074 Aachen, Germany
{terboven, anmey, schmidl, wagner}@rz.rwth-aachen.de

Abstract

MPI and OpenMP are the de-facto standards for distributed-memory and shared-memory parallelization, respectively. By employing a hybrid approach, that is combing OpenMP and MPI parallelization in one program, a cluster of SMP systems can be exploited. Nevertheless, mixing programming paradigms and writing explicit message passing code might increase the parallel program development time significantly. Intel Cluster OpenMP is the first commercially available OpenMP implementation for a cluster, combining the ease of use of the OpenMP parallelization paradigm with the cost efficiency of a commodity cluster. In this paper we present our first experiences with Intel Cluster OpenMP.

1 Introduction

The main advantage of shared-memory parallelization with OpenMP [2] over MPI [1] is that data can be accessed by all instruction streams without reasoning whether it must be transferred beforehand. This allows for an incremental parallelization approach and leads to shorter parallel program development time. Complicated dynamic data structures and irregular and possibly changing data access patterns make programming in MPI more difficult, whereas the level of complexity introduced by shared-memory parallelization is lower in many cases. As OpenMP is a directive-based language, the original serial program can stay intact, which is an advantage over other shared-memory parallelization paradigms.

The downside of OpenMP and other shared-memory paradigms as well is that the resulting parallel programs are normally restricted to execute in a single address space. Bus-based multi-processor machines typically do not scale well beyond four processors for memory-intense applications. Larger SMP and ccNUMA systems require to employ scalable and thus expensive interconnects. Because of that, several attempts to bring OpenMP to clusters have been made in the past.

In [8] an OpenMP implementation for the TreadMarks software has been presented, which supports only a subset of the OpenMP standard. In [10] an OpenMP implementation on top of the page-based distributed shared-memory (DSM) system SCASH has been presented for the Omni source-to-source translator. In this approach, all accesses to global variables are replaced by accesses into the DSM and all shared data is controlled by the DSM. Although the full OpenMP specification is implemented, support for the C++ programming language is missing. In 2006, Intel made the first commercial implementation of OpenMP for clusters available, named Intel Cluster OpenMP [6], sometimes referred to as ClOMP in this paper. The full OpenMP 2.5 standard for Fortran, C and C++ is implemented, although nested parallel regions are not yet supported.

This paper is organized as follows: In section 2 we give an overview of OpenMP and Intel Cluster OpenMP. In section 3 we present micro-benchmark [3, 9] measurements of OpenMP and ClOMP constructs and discuss which types of applications we expect to profit from running on a cluster. In section 4 we present results of four applications with Intel Cluster OpenMP. The current tool support for ClOMP is discussed briefly in section 5. We draw our conclusions and touch on future plans in section 6.

2 OpenMP

This section will present an overview of OpenMP and Intel Cluster OpenMP and discuss some aspects of the memory model of both.

2.1 Overview

OpenMP consists of a collection of compiler directives, library functions and a few environment variables. It applies the so-called fork/join programming model, thus an OpenMP program starts as a single thread. At the entrance

of a *parallel region*, additional worker threads are created, thus forming a team of threads together with the initial thread, which becomes the master of that team. The worker threads are suspended at the end of the parallel region and are ready to be reused at the next opportunity.

Unless specified otherwise, all threads execute the whole code within the parallel region redundantly. If, for example, a loop inside a parallel region is enclosed by an OpenMP work-sharing loop construct, the loop iterations are distributed across the threads of the current team. The way in which the loop iterations are distributed among the threads can be controlled elegantly via the *schedule* clause. Other work-sharing constructs are available as well as are reduction type operations and synchronization constructs.

As OpenMP is a shared-memory parallelization paradigm, all threads share a single address space, but still can have thread local storage to hold private data. It is the programmer's responsibility to control the scoping, that is the classification of variables into shared and private, of all variables that are used within a parallel region.

At the beginning and the end of any parallel region, all threads of a team are implicitly synchronized. At *barrier* synchronization points all threads have to wait until every team member has arrived, before any thread may continue. One thread modifying a shared variable and other threads reading or writing the same variable without careful synchronization may lead to so-called *data races*. A data race causes the program's output to depend on the actual interleaving of threads, which cannot be predicted. It is the programmer's responsibility to use the synchronization construct provided by OpenMP in order to make sure that modifications of shared data are properly reflected to all threads.

2.2 Memory Model

OpenMP provides a relaxed memory consistency model similar to the weak ordering memory model [5]. Each thread has a *temporary view* of the memory that is not required to be consistent with the memory at all times. Writes to memory are allowed to overlap other computation and reads from memory are allowed to be satisfied from a local copy of memory, under some circumstances. For example, if within one synchronization period the same memory location is read again, this can be done from fast local storage (the temporary view, e.g. a cache). Thus, it is possible to hide the memory latency within an OpenMP program to some extent. This also allows Intel Cluster OpenMP to fulfill reads from local memory under certain circumstances, instead of accessing remote memory in all cases, as will be explained in the following subsection 2.3.

The *flush* construct of OpenMP serves as a memory synchronization operation, as it enforces consistency between the temporary view and the memory, by writing back a set of variables or even all thread's variables to the memory. All reads and writes from and to the memory are unordered with respect to each other (except for those being ordered by the semantics of the base language), but ordered with respect to an OpenMP flush operation. All OpenMP barriers also contain an implicit flush operation.

2.3 Intel Cluster OpenMP

Beginning with version 9.1, the Intel C/C++ and Fortran compilers for Linux are available with Cluster OpenMP. The distributed shared-memory (DSM) system of Intel Cluster OpenMP is based on a licensed derivative of the TreadMarks software.

Intel has extended OpenMP with one additional directive: The *sharable* directive. It identifies variables that are referenced by more than one thread and thus have to be managed by the DSM system. While certain variables are automatically made sharable by the compiler, some variables have to be declared sharable explicitly by the programmer, e.g. file-scope variables in C and C++. Thus, the programmer's responsibility for variable scoping has been extended to finding all variables that have to be made sharable, in the cases where the compiler was unable to detect it. As will be shown in section 4, this can sometimes be a tedious task for application codes.

For the Fortran programming language several compiler options exist to make different kinds of variables sharable automatically, e.g. all module or common block variables. In addition to finding all variables that have to be declared sharable, dynamic memory management in an application deserves some attention. For all variable allocations from the heap (e.g. by `malloc`), it has to be determined whether the memory should be taken from the regular heap, thus being only accessible by the thread calling `malloc`, or from the DSM heap, thus being accessible by all threads. Intel Cluster OpenMP provides several routines to easily replace native heap memory management routines by DSM heap routines.

The task of keeping shared variables consistent across multiple nodes is handled by the Cluster OpenMP runtime library. Intel provides detailed information on how this process works in the product documentation and in a white paper [6]. In principle the mechanism relies on protecting memory pages via the `mprotect` system call; pages that are not fully up-to-date are protected against reading and writing. When a program reads from such a protected page, a segmentation fault occurs and after intercepting the corresponding signal the runtime library requests updates from all nodes, applies them to the page and then removes the protection. At the next access, the instruction finds the memory accessible and then the read will complete successfully. Still the page is protected against writing. In case of a

write operation, a so-called twin page is created for further reads and writes on the accessing node, after the protection has been removed. The twin page then becomes the thread's temporary view.

The higher the ratio of cheap memory accesses, that means to thread private memory or to twin pages, versus expensive memory accesses, the better the program will perform. At each synchronization construct, e.g. a barrier, nodes receive information about pages modified by other nodes and invalidate those. As a consequence, the next access will be expensive.

3 Micro-Benchmarks

In order to better understand the behavior of Intel Cluster OpenMP's DSM mechanism and to get an estimate of how expensive the DSM overhead is, we created a set of micro-benchmarks. In addition, we compared the well-known OpenMP micro-benchmarks [3, 9] with Intel Cluster OpenMP on two different network fabrics.

All measurements presented in this and the following sections were carried out on a cluster of eight Dell PowerEdge 1950 servers equipped with two Intel Xeon 5160 (dual-core, 3.0 GHz) CPUs. All nodes are running Scientific Linux 5.0 and are connected via Gigabit Ethernet (referred to as Eth) and 4x SDR InfiniBand (referred to as IB). The InfiniBand adapters are attached to the PCI-Express bus. We used the Intel 10.0.025 compiler suite for 64-bit systems.

Table 1 shows selected results of the OpenMP micro-benchmarks for traditional OpenMP and Intel Cluster OpenMP. The EPCC OpenMP micro-benchmarks measure the overhead of OpenMP constructs by comparing the time taken for a section of code executed sequentially, to the time taken for the same code executed in parallel enclosed in a given directive. We ported the EPCC micro-benchmarks to Intel Cluster OpenMP by adding sharable directives, where necessary.

It it obvious that there is a severe difference in overhead between OpenMP and Intel Cluster OpenMP, independent of the network fabric. Thus, the granularity of parallelism to be efficiently exploitable with Cluster OpenMP has to be much coarser. While for a run with a single thread only a small difference between the two network fabrics can be observed, the overhead increase with two and four threads is significantly lower on InfiniBand than on Ethernet. As will be seen in section 4, application codes resemble this behavior. We found that using a fast network like InfiniBand is crucial in order to exploit application scalability with Intel Cluster OpenMP.

We implemented a couple of own micro-benchmarks especially to test the DSM performance by employing

	OpenMP	ClOMP Eth	ClOMP IB
PARALLEL FOR			
1 thread	0.31	478.82	482.84
2 threads	1.00	1159.53	720.62
4 threads	1.12	1540.97	962.52
BARRIER			
1 thread	0.01	478.24	481.37
2 threads	0.43	738.38	589.95
4 threads	0.60	751.61	634.64
REDUCTION			
1 thread	0.35	479.44	481.34
2 threads	1.54	1888.25	1302.87
4 threads	2.32	3315.19	2660.42

Table 1. Selected results (overhead in microseconds [us]) of the EPCC OpenMP micro-benchmarks for OpenMP and Intel Cluster OpenMP, with one thread per node.

the same measurement approach as the EPCC micro-benchmarks, of which the following are of interest here:

- *testheap*: A number of pages is allocated via `kmp_aligned_sharable_malloc` (OpenMP: `valloc`), then they are written and then freed again. This process is repeated a couple of times and the average runtime is calculated.

- *read_f_other*: The time required to read a page allocated via the DSM by a different thread is measured. For the Cluster OpenMP runtime that requires transferring the page.

- *write_t_other*: Similar to read_f_other, but now the page allocated by a different thread is written. For the Cluster OpenMP runtime that requires creating a twin page.

The performance results for traditional OpenMP and Intel Cluster OpenMP are shown in table 2. The OpenMP measurements were run with two threads. Both ClOMP measurements were run with two Cluster OpenMP processes on one and two nodes, respectively.

We experienced noteworthy variations in the results on InfiniBand. This is due to thread creation by the Intel Cluster OpenMP runtime for communication handling.

It becomes obvious that allocating dynamic memory gets more expensive. It is considered good parallel programming practice to allocate as large chunks of memory as possible (thus as seldom as possible) in order to not stress the operating system's memory management. With Intel Cluster OpenMP, special care has to be taken in case of dynamic

	testheap	read_f_other	write_t_other
OpenMP	0.85	1.8	2.32
ClOMP, Eth			
1 node	3.81	2.74	2.44
2 nodes	10.75	255.56	251.82
ClOMP, IB			
1 node	3.80	1.76	4.26
2 nodes	26.33	101.34	104.54

Table 2. Selected results (two threads, overhead in microseconds [us]) of our Cluster OpenMP micro-benchmarks.

data structures which involve many allocations, maybe even hidden from the user via an abstract interface.

Although Cluster OpenMP allows the programmer to access memory on other nodes transparently, from a performance perspective this is not for free. Intel Cluster OpenMP can be started to use more than one thread per node, instead of multiple processes on one node. In that case, accessing memory from a different thread on the same node is significantly cheaper.

With Intel Cluster OpenMP it is even more important to respect and stick to the following OpenMP tuning advices:

- *Enlarge the parallel region*: Creating a team of threads at the entrance to a parallel region and putting it aside at the exit involves some overhead, although most current compilers do a good job in keeping it minimal. Fewer and shorter serial parts contribute to better scalability, thus parallel regions should be as large as possible in most cases. With Cluster OpenMP the overhead of creating or activating a team of threads is higher than for OpenMP, as all involved nodes have to communicate.

- *Work on data locally*: Keeping data local is very important on ccNUMA architectures. We found that tuning measures for ccNUMA also improve performance on Cluster OpenMP, for example respecting the first touch initialization strategy of the Linux operating system. If threads are accessing local memory, no page transportation or twin page has to be created.

- *Prevent false sharing*: Normally, false sharing occurs when threads write to different parts of the same cache line. Thus, false sharing does not result in a data race. Because of that, e.g. in the case of two threads running on two different cores that do not share a cache, only one core can hold the valid cache line, thus the other core has to wait and update later. This can affect the performance significantly. In the case of Cluster OpenMP, false sharing becomes an issue on a per page basis. If two or more threads write to different locations on the same page, the update process has to occur at the next synchronization point. This kind of problem can be resolved by inserting appropriate padding in many cases, although it is pretty hard to detect in complex applications.

4 Applications

In this section, we present our experiences of porting four different applications to Intel Cluster OpenMP. We used the same experiment setup as in the previous section.

4.1 Jacobi

We tried Intel Cluster OpenMP on the Jacobian solver available on the OpenMP website [2]. We measured the scalability using a matrix size of 6000×6000. We compared the Cluster OpenMP version on Ethernet and InfiniBand to traditional OpenMP and two MPI implementations, one with synchronous communication and one with asynchronous communication.

In all versions the domain decomposition approach for the parallelization is exactly the same. The main difference is that with MPI the data on the boundary has to be transferred explicitly, while the programmer does not have to reason about that in OpenMP. As the DSM system of Intel Cluster OpenMP works on a per page basis, in some cases depending on the total number of threads and the number of threads per node, some threads will have to access pages on other nodes for reading data at or near the boundaries.

The comparison between OpenMP and Cluster OpenMP is shown in figure 1. We found that by binding Cluster OpenMP threads to scattered cores with the `KMP_AFFINITY` environment variable the performance can be improved. Binding lead to speedup improvements of up to 10% and was especially effective for the runs with two threads.

It can be noticed that the scalability on one node is limited to two threads, as the Jacobi solver stresses the memory bandwidth. Thus, running with two Cluster OpenMP threads on two nodes shows a better scalability (1.95 over 1.67) than the traditional OpenMP version on a single node, as the memory bandwidth available to the application is virtually doubled by running on two nodes. Of course, using more than two threads per node does not improve scalability with Cluster OpenMP for the same reason as with the traditional OpenMP version. The maximum speedup is obtained with eight nodes: 9.92 with InfiniBand and two threads per node and 7.50 with Ethernet and four threads per node.

The scaling of the MPI version with synchronous communication is shown in figure 2, the version with asynchronous communication is shown in figure 3.

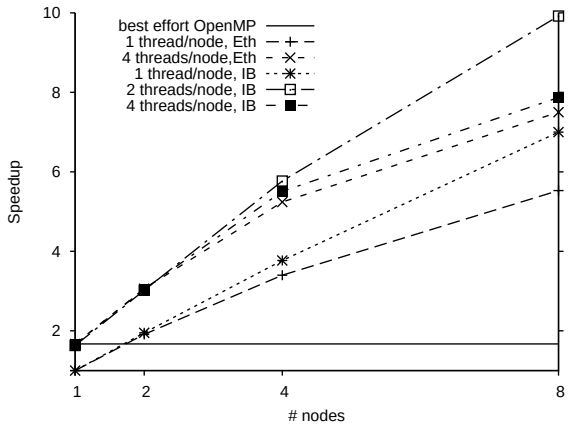

Figure 1. Speedup of the Cluster OpenMP version of Jacobi.

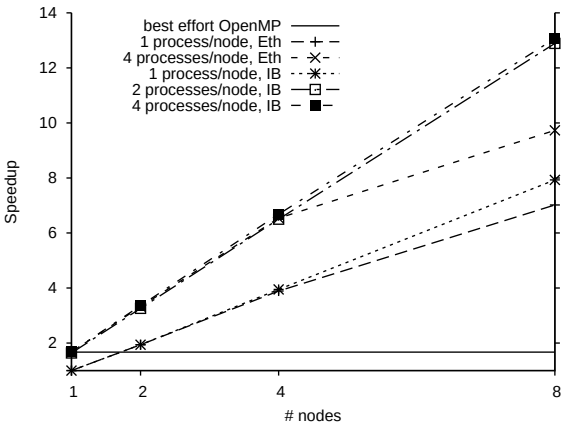

Figure 2. Speedup of the synchronous MPI version of Jacobi.

Overlapping communication and computation with asynchronous MPI is particularly beneficial when employing the slower Gigabit Ethernet network fabric, whereas for InfiniBand it does not make a big difference. Likewise the Cluster OpenMP version profits from the faster network, because communication and computation cannot be overlapped. In all cases MPI clearly outperforms Cluster OpenMP.

Both MPI versions deliver a speedup of slightly more than 13 with eight nodes and four processes when using the fast InfiniBand network fabric, whereas the speedup of Cluster OpenMP is limited to 9.92 employing 2 threads per node at best.

There are two places in the program where communication is involved: In updating data on the boundaries of the subdomains and in the reduction operation to calculate the error estimation. In order to improve the Cluster OpenMP version, we implemented prefetching with an additional Posix-thread, that would be similar to overlapping computation and communication.

Unfortunately, we were unable to achieve any significant performance improvement by prefetching the boundary data (we used the `segvprof.pl` tool provided by Intel to make sure the prefetching worked as expected). To understand this disappointing result, we extrapolated the runtime for eight threads on eight nodes from the serial runtime assuming perfect scalability. On the other hand, we predicted the runtime on the basis of our previous micro-benchmark measurements for page transfers and reduction operations on eight nodes. As both estimations only differ within 1.5 percent, we concluded that with prefetching there is only little to gain. This result corresponds to the observation that the asynchronous and synchronous MPI versions perform similarly on the fast InfiniBand network. When applying the prefetch strategy in combination with the slower GE network, we observed a slight speedup improvement of about four percent.

We took a closer look at the MPI version as well. Using the Intel Trace Analyzer tool, we observed a communication overhead of 3.3 percent of the total runtime in MPI_Recv and about 22.4 percent in MPI_Allreduce for a run of 32 processes. That approves that the collective reduction operation is much more expensive than the point-to-point sends and receives and this gap will further grow with increasing the number of processes.

As our micro-benchmark experiments revealed that an MPI reduction operation performs significantly better than a reduction operation in Cluster OpenMP, we linked the Intel Cluster OpenMP program with the Intel MPI library and called the MPI reduction operation from within the Cluster OpenMP program; this combination is probably not officially supported by Intel. Unluckily, the current Intel MPI version does not support full multi-threading, so we had to implement expensive locking. By replacing Cluster OpenMP's reduction operation with the MPI reductions, we got an increase in speedup of 1.5% on the presented dataset. On a different dataset where more iterations are required and thus more reduction operations are called, we achieved a speedup of up to 12% with the MPI reduction operation. We concluded that there is still room for improvement in the Intel Cluster OpenMP implementation, as the reduction can be implemented with less locking than in our experiments.

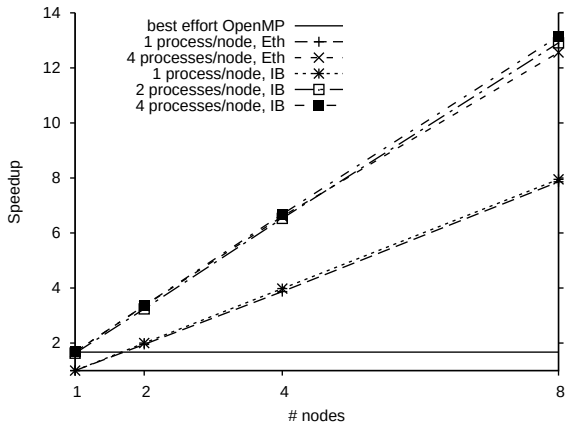

Figure 3. Speedup of the asynchronous MPI version of Jacobi.

4.2 Sparse Matrix-Vector-Multiplication

A sparse matrix-vector-multiplication (SMXV) typically is the most time consuming part in iterative solvers. In order to estimate whether Intel Cluster OpenMP is suited for this class of applications, we examined the SMXV benchmark kernel of DROPS, a 3D CFD package for simulating two-phase flows with a matrix of some 300 MB and about 19,600,000 nonzeros.

The performance of the SMXV benchmark is shown in table 3. In addition to the Woodcrest-based systems (UMA), we evaluated the performance on a Sun Fire V40z server system, equipped with four AMD Opteron 848 single-core 2.2 GHz CPUs (ccNUMA), which provides a ccNUMA architecture. We compared two parallelization strategies: In the *rows*-strategy the parallel loop runs over the number of rows and a dynamic loop schedule is used for load balancing, while in the *nonzeros*-strategy the number of nonzeros is statically partitioned into blocks of approximately equal size, one block for each thread.

The *nonzeros*-strategy outperforms the *rows*-strategy on the ccNUMA architecture and on Intel Cluster OpenMP as well, when carefully initializing all data respecting the operating system's first touch policy. While the dynamic loop scheduling in the *rows*-strategy successfully provides good load balance, the memory locality is not optimal. The *nonzeros*-strategy shows a neglectible load imbalance for the given dataset, but its advantage is that each thread works on local data. Employing the locality of the *nonzeros*-strategy, we observed a nearly linear speedup for the case of one thread per node. There is only little difference between Gigabit Ethernet and InfiniBand, as there is only little communication involved. In short, Cluster OpenMP behaves like a distinct ccNUMA architecture.

	rows		nonzeros	
	1 thread p. node	4 threads p. node	1 thread p. node	4 threads p. node
OpenMP UMA	561.9	960	561.5	978.1
OpenMP ccNUMA	326.3	793.9	324.5	1147.6
ClOMP Eth, 1 node	548.0	887.2	551.8	939.4
ClOMP Eth, 2 nodes	113.0	540.1	1058.7	1382.4
ClOMP Eth, 4 nodes	14.5	136.8	2037.9	2435.6
ClOMP IB, 1 node	547.9	817.9	551.9	940.4
ClOMP IB, 2 nodes	904.4	1208.4	1072.0	1415.3
ClOMP IB, 4 nodes	1328.3	1845.4	2075.0	2536.6

Table 3. Performance [MFLOP/s] of SMXV.

4.3 Fire

The Flexible Image Retrieval Engine (FIRE) [4] has been developed at the Human Language Technology and Pattern Recognition Group of the RWTH Aachen University. The benchmark version which we examined consists of more than 35,000 lines of C++ code. The current version of FIRE is available for download in the Internet.

Given a query image and the goal to find k images from a database that are similar to the query image, a score is calculated for each image from the database and the k database images with the highest score are returned. In [11] two layers have been parallelized with OpenMP and displayed nearly linear scalability. Shared-memory parallelization is obviously more suitable than distributed-memory parallelization for the image retrieval task, as the image database can then be accessed by all threads and does not need to be distributed. Because of that, we expected FIRE to be a perfect candidate for Intel Cluster OpenMP as searching through the database involves very little synchronization and only neglectible writing to shared memory.

To make variables of the C++ STL sharable, instances of such variables have to use the `kmp_sharable` allocator. In order to achieve this, that allocator has to be specified at the variable declaration. On one hand this solution is elegant and does not require much code changes at the declaration point, but on the other hand the type signature of the variable is changed. This implies that if such a variable is

passed as a parameter to a function, the function declaration has to be changed to reflect the type change.

The FIRE code makes extensive usage of the STL, many of FIRE's object data types use STL data types as members or even are derived from STL data types. Variables are passed down the call stack to all functions requiring access to them. In order to make FIRE work with Intel Cluster OpenMP, virtually the whole code base would have to be touched and nearly every class would have to be changed. This is not feasible in a limited amount of time and in contrast to the findings in [11] that with OpenMP only very little code changes were necessary. Providing a STL which allocates all STL variables as sharable might be a solution for this and similar codes.

4.4 PANTA

PANTA is a 3D solver that is used in the modeling of turbomachinery [12]. The package used in our experiments consists of about 50,000 lines of Fortran 90 code. Several approaches to parallelize this code have been described, e.g. [7]. In order to achieve the best possible speedup with Cluster OpenMP, we have chosen the highest level parallelization currently exploited with OpenMP, that is a loop over 80 inversion zones.

We had to manually compute the distribution of loop iterations onto threads, as the OpenMP DO work-sharing construct was not applicable in this case because of the code structure. As the number of loop iterations is relatively small and as at the end of each loop iteration there is a critical region in which some global arrays are updated in a reduction-type manner, we cannot expect good scaling from this code.

Creating a Cluster OpenMP version of the PANTA code parallelized with OpenMP was straight forward: We enabled the compiler's autodetection and propagation of sharable variables and asked the compiler to make all argument expressions, all common block variables, all module variables and all save variables sharable by default. The performance of the resulting program is shown in figure 4.

We are aware of the fact that making all these variable types sharable by default puts more variables under the control of the DSM than necessary and that this will probably cause a performance penalty. Nevertheless, the scalability of the Cluster OpenMP version on a single node is similar to the OpenMP version, thus the penalty is acceptable in the case where as many Cluster OpenMP threads (not processes) are used per node as possible.

Better scalability with traditional OpenMP on a single node is prohibited because the available memory bandwidth is saturated. Using Intel Cluster OpenMP, we can use more than one node and thus effectively increase the available memory bandwidth. Using two nodes, the best effort

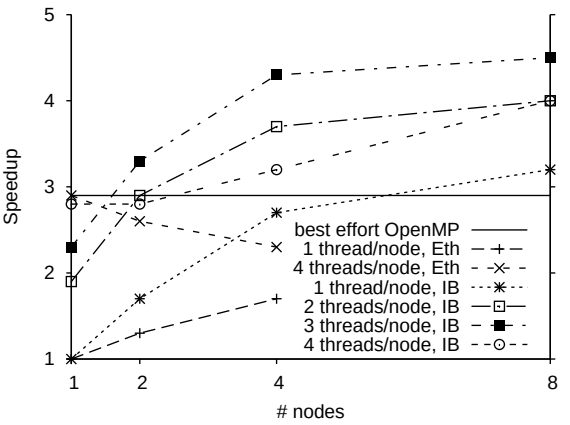

Figure 4. Speedup of Panta.

speedup can be increased from 2.9 with traditional OpenMP to 3.3, using four nodes to 4.3. Adding more nodes will only lead to slight improvements.

Using four threads per node performs worse than only three threads per node. According to Intel, one possible explanation might be that the Cluster OpenMP management thread taking care of the DSM system infers with the computational threads.

As already seen with the micro-benchmarks and the Jacobian solver, using InfiniBand improves the performance of Cluster OpenMP significantly. For PANTA, Gigabit Ethernet performs worse than traditional OpenMP in all cases. Improvements in the latency and bandwidth of recent InfiniBand products might increase the scalability of this application further.

Unluckily, the current version of Intel Cluster OpenMP does not support Nested OpenMP. For the Panta code, there is an additional OpenMP parallelization at the loop level available, namely at the linear equation solver [7]. We suspect that employing this level with two or even four threads per node would increase the total scalability of the program.

5 Tool support

The DSM-mechanism used by Intel Cluster OpenMP uses segmentation fault signals to activate the page movement and synchronization mechanism. That makes debugging a Cluster OpenMP program very hard, if not impossible, if the debugger cannot be taught to ignore the segfaults and to not step into the Cluster OpenMP library's handler routine. In doing so we successfully used the Intel command line debugger and the TotalView GUI-based debugger with Cluster OpenMP programs. Nevertheless, using traditional debuggers is not very helpful in finding errors related

to Intel Cluster OpenMP. The typical problem is that a variable has erroneously not been made sharable. In this case some threads will run into segmentation faults when accessing that memory location, but the runtime system is unable to deliver the page and thus terminates the program in most cases.

In order to find the places in which accesses to variables that are not sharable occur, one can use the command line tool `addr2line` on a core dump. We found it easy to use and in most cases it was no problem to figure out which variable has caused the problem. Intel has announced that future versions of the Intel Thread Checker tool will also find variables that should be made sharable.

In addition, Intel delivers a command line tool named `segvprof.pl` that provides means to count the number of segmentation faults on the function level. This can be handy in locating parts of the program that are not performing well, as e.g. too many accesses to remote pages occur.

Again, this tool is very basic in it's current form and for complex codes like PANTA, the provided functionality is too limited to find and understand performance problems related to Cluster OpenMP. Intel has announced that future versions of the Intel Trace Collector and Analyzer will support such an analysis.

6 Conclusions and Future Work

Intel Cluster OpenMP allows shared-memory OpenMP programs to be executed on a cluster. It takes advantage of the relaxed consistency memory model of OpenMP. Nevertheless, OpenMP primitives get two to four orders of magnitudes more expensive.

Intel Cluster OpenMP proved to be successful for several small applications and while preserving the easier and more comfortable parallelization paradigm of OpenMP and shared-memory, a cluster of SMP nodes could be exploited. But for more complex applications like PANTA, scalability does not come for free and further tuning has to be invested.

We ran into problems with C++ programs employing the STL, which still have to be resolved. We suspect that there is room for improvement concerning Intel's current implementation of reductions and on the tool support.

Future work will be to evaluate more programs of the scientific domain with Intel Cluster OpenMP. We will apply tuning measures to Cluster OpenMP programs: Porting codes like PANTA was straight forward because of the compiler features provided, still the full performance potential has not yet been achieved. We are interested in combining Cluster OpenMP with other parallelization paradigms to enable multi-level parallelism.

Acknowledgements

We sincerely thank Jay Hoeflinger and Larry Meadows from Intel for providing hints on potential performance improvements.

References

[1] MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, May 1994.

[2] ARB. OpenMP Application Program Interface, May 2005.

[3] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. In *European Workshop on OpenMP (EWOMP)*, Lund, Sweden, September 1999.

[4] T. Deselaers, D. Keysers, and H. Ney. Features for Image Retrieval - a quantitative comparison. In *26th DAGM Symposium, Pattern Recognition (DAGM 2004)*, number 3175 in Lecture Notes in Computer Science, pages 228 – 236, Tübingen, Germany, 2004.

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2006.

[6] J. P. Hoeflinger. Extending OpenMP to Clusters. 2006.

[7] Y. Lin, C. Terboven, D. an Mey, and N. Copty. Automatic Scoping of Variables in Parallel Regions of an OpenMP Program. In *Workshop on OpenMP Applications and Tools (WOMPAT 2004)*, Houston, USA, May 2004.

[8] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on Network of Workstations. 1998.

[9] F. J. L. Reid and J. M. Bull. OpenMP Microbenchmarks Version 2.0. In *6th European Workshop on OpenMP (EWOMP 2004)*, pages 63 – 68, Stockholm, Sweden, October 2004.

[10] M. Sato, H. Harada, A. Hasegawa, and Y. Ishikawa. Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. *Scientific Programming*, 9(2,3):123–130, 2001.

[11] C. Terboven, T. Deselaers, C. Bischof, and H. Ney. Shared-Memory Parallelization for Content-based Image Retrieval. In *ECCV 2006 Workshop on Computation Intensive Methods for Computer Vision (CIMCV)*, Graz, Austria, May 2006.

[12] T. Volmar, B. Brouillet, H. E. Gallus, and H. Benetschik. Time Accurate 3D Navier-Stokes Analysis of a 1.5 Stage Axial Flow Turbine, 1998.