Performance Comparison of MPI and three OpenMP Programming Styles on Shared Memory Multiprocessors

Géraud Krawezik
LRI, Bâtiment 490, Université de Paris Sud
91000 Orsay, France
gk@lri.fr

Franck Cappello
LRI, Bâtiment 490, Université de Paris Sud
91000 Orsay, France
fci@lri.fr

ABSTRACT

When using a shared memory multiprocessor, the programmer faces the selection of the portable programming model which will deliver the best performance. Even if he restricts his choice to the standard programming environments (MPI and OpenMP), he has a choice of a broad range of programming approaches.

To help the programmer in his selection, we compare MPI with three OpenMP programming styles (loop level, loop level with large parallel sections, SPMD) using a subset of the NAS benchmark (CG, MG, FT, LU), two dataset sizes (A and B) and two shared memory multiprocessors (IBM SP3 Night Hawk II, SGI Origin 3800). We also present a path from MPI to OpenMP SPMD guiding the programmers starting from an existing MPI code. We present the first SPMD OpenMP version of the NAS benchmark and compare it with other OpenMP versions from independent sources (PBN, SDSC and RWCP). Experimental results demonstrate that OpenMP provides competitive performance compared to MPI for a large set of experimental conditions. However the price of this performance is a strong programming effort on data set adaptation and inter-thread communications. MPI still provides the best performance under some conditions. We present breakdowns of the execution times and measurements of hardware performance counters to explain the performance differences.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*

General Terms

Performance Measurement Experimentation

Keywords

MPI, OpenMP, Performance evaluation, Multiprocessors, Shared memory

1. INTRODUCTION

In the past decades, high performance computers have been implemented using a large variety of architectures: vector, MPP, SMP, COMA, NUMA, Clusters. Many parallel machines of the early 90s had a distributed memory architecture, and a lot of applications were ported to the message passing paradigm. The trend of the late 90s, which still holds, was for hybrid architectures (cluster of SMP or NUMA). They require a programming effort for porting message passing applications to the hybrid memory model, and many experiments [1] [2] [3] [4] have shown the difficulties of getting higher performance with this model. The fundamental reasons are the strong maturity of the message passing paradigm and the availability of highly tuned message passing libraries for distributed as well as for shared memory architectures. A current trend is to enlarge the number of processors inside the multiprocessors (SMP or NUMA). Considering the disappointing experience of hybrid architecture programming and its fundamental reasons, the HPC programmer on a shared memory multiprocessor faces the following issues: 1) which paradigm to choose for programming new applications and 2) whether or not to adapt an existing MPI application to the shared memory paradigm.

To help the programmer, we present a performance comparison of the message passing and shared memory paradigms for shared memory multiprocessors considering the *de facto* standards MPI and OpenMP and the well established NAS Parallel Benchmark. This paper provides three contributions: 1) a path for porting existing MPI applications to OpenMP SPMD including a set of computation and communication optimizations and its application for the parallelization of the NAS benchmark. 2) The comparison of MPI and all the known OpenMP programming styles for the NAS Benchmark on three hardware platforms. 3) The explanations of the performance results.

The second section describes the programming models analyzed in this study while section 3 presents the experimental conditions. In section 4, the SPMD Open-MP style and the path for porting existing MPI codes to OpenMP SPMD are detailed. Section 5 presents the performance comparison results. Section 6 explains these results using timings and hardware performance counters. Section 7 discusses the previous research studies in this field. Section 8 concludes.

2. PROGRAMMING SHARED MEMORY MULTIPROCESSORS

In this paper, we consider the *de facto* standards for message passing and shared memory programming styles: the MPI and OpenMP environments.

MPI (Message Passing Interface) is a well known message passing environment mainly used for SPMD 'coarse grain' and master-worker programming styles. We will not present it further here, since we assume that the reader is already familiar with this programming style. We recommend the reader who needs more explanations about MPI to refer to [5] and [6].

'Naive' Loop Level OpenMP. This loop level programming style is the most popular one with OpenMP mainly because it is easy to use and enables incremental development. Parallelizing a code requires to 1) discover the parallel loop nests contributing significantly to the computation time (by profiling the execution of the serial version of the code and checking data dependencies), 2) add directives for starting/closing parallel regions, managing the parallel threads (workload distribution, synchronization) and managing data. The 'loop level' approach means that the parallelization is considered loop nest by loop nest using the "`OMP DO`" construct. 'Naive' means that there is no attempt to encompass several parallelized loop nests in the same parallel region. This is the model that has been investigated by most researchers, notably in [7].

'Improved' Loop Level OpenMP: Some authors state that the previous model suffers from weaknesses: 1) the multiplication of the `PARALLEL` constructs implies a high thread management cost and 2) there is no guarantee of affinity between the threads and processors for consecutive parallel regions. This may lead to a bad utilization of the memory hierarchy. One approach proposed to reduce these weaknesses is to use a coarser grain of parallelization in order to obtain only one parallel region. As a loop level approach, the workload distribution within the parallel region still relies on the "`OMP DO`" directive. Since the declared parallel regions may encompass critical sections and non parallel sections, the programmer must use synchronization directives (`!$OMP MASTER`, `CRITICAL`, `BARRIER`) to enforce the semantic of the original sequential program. The incremental parallelization still applies, which keeps the model attractive. This is the model adopted by the RWCP team to test their OpenMP compiler presented in [8].

SPMD programming with OpenMP: This model is slightly different from the previous ones. As for the message passing paradigm, the programmer has to express manually the data and work distribution among the threads. The working data sets (multidimensional arrays) are spread in array sections by the programmer. Even though these array sections are stored inside the same shared memory hardware, the programmer uses them as if they where stored in separate memory banks. The programmer has to distribute the workload among the threads according to this data distribution trying to limit the data exchanges between threads. The two main side effects of this programming style are: 1) the parallelization of the loop nests requires to compute explicitly the iterations distribution among the threads (no more use of the `!$OMP DO` construct) and 2) the programmer has to express the data exchange between threads through explicit communications. Usually, the bounds for each loop are calculated in a setup part of the program, based on the number of threads, and the identifier of each thread (by a call to `omp_get_thread_num()`). This programming style allows the programmer to carefully manage data locality. For example, the explicit distribution of datasets and loop nest iterations allows taking advantage of parallel loop nests optimizations. Some case studies have been previously presented, notably by the authors of [9] and [10].

About nested parallelism with OpenMP: Most vendor compilers simply serialize the inner parallel regions when the nesting of parallel directives is used. The Compaq-HP and Fujitsu Fortran compilers are exceptions, as they fully implement the nesting of parallel regions. While nested parallelism with standard directives may generate efficient code for non-dynamic loop boundaries (typically: dense linear algebra), tests on a Compaq-HP ES40 4-way SMP machine demonstrate for the conjuguate gradient case (featuring dynamic boundaries computations) performance lesser than the one obtained with the sequential code. The analyze of the generated assembly code shows that the compiler produces a much larger executable code than for the single-level parallelism, including heavy management cost due to dynamic loop boundaries computation. Some experimental compilers, such as Nanos [11] or Omni/ST [12], provide ways to express multiple-level parallelism. In some cases they do not enable the programmer to do it simply (Nanos) as they use non-standard directives and a complex parallelization scheme, while in other cases the speedup may remain low (20 for 60 processors with an FFT program for Omni/ST). In any case, to be competitive with the SPMD model, a compiler implementing nested parallelism should distribute the loop iterations among the threads in order to produce blocking of memory references in the same way that the user does it manually for the SPMD mode. Currently, vendors and experimental compilers exhibit significant portability or performance issues that preclude the use of nested parallelism with OpenMP as a standard parallelization approach.

3. EXPERIMENTAL CONDITIONS

3.1 Methodology

We focus our investigation on performance and scalability and use timings and hardware performance counters as the metrics for our measurements. All codes (MPI and OpenMP) implementations are instrumented to provide execution time breakdowns.

Understanding the performance and scalability of parallel programs (MPI and OpenMP) requires measuring the execution times of the computation and data exchange parts. Determining the exact separation between them is not obvious for shared memory programs since they are interlaced and their respective contribution to the global execution time is difficult to extract since some cache misses experienced during one part may be due to invalidations raised during the others. Computation and data exchange parts themselves encompass subparts such as actual computation and memory references for the computation part and synchronization and actual data exchanges for the data exchange part.

To keep the comparison relevant between MPI and Open-

MP, we provide a two level result analysis. The first one is a breakdown of the execution time in computation and data exchange parts. The second one investigates deeply the computation part by examining the L1 and L2 data cache misses.

We use the vendors compilers and libraries, the documented compilers options and the execution mode corresponding to a multi-users environment.

3.2 Platforms

The IBM SP3 Night Hawk II node features sixteen Power3+ processors at 375 MHz. They are connected to the main memory (up to 16 GB) by a crossbar switch, with a bandwidth up to 14.2 GB/s. The cache for each processor is 16KB of L1 and 8MB for the L2. The programs are compiled with XLF 7.1 using the options: `-O3 -bmaxdata:0x80000000 -qcache=auto -qarch=auto -qtune=auto -qmaxmem=-1 -qunroll -qnosave`.
The programs are executed via the LoadLeveler proprietary queuing system. We use the shared memory version of the MPI library in order to get the best performance.

The SGI Origin 3800 features up to 128 4-way symmetric multiprocessors interconnected by a NUMA switch. Each processor is a R14000 at 500 MHz with 128 KB L1 and 8 MB L2. The machine used in this experiment features 64 processors and 32 Gigabytes of memory. We use the vendor compiler with the options: `-O3 -r14000 -64 -mips4 -OPT:IEEE_arithmetic=3:roundoff=3:unroll_analysis=on -LNO:cs2=4m:fusio=2 -CG:ld_latency=4 -OPT:pad_common=ON -IPA:cprop=OFF`. Execution is made using the vendor MPI library in a shared environment, with access through the LSF batch queuing system using several CPUSETs. With this configuration, we repeated the experiments 10 times and observed small performance variations (standard deviation for 10 measures is less than 5%). We present the best of these 10 measures.

3.3 Benchmark and Implementations

For our experiments, we have chosen three kernels (CG, MG, FT), and a mini application (LU) among the NAS (NPB) benchmark suite developed at NASA Ames Research Center [13]. We selected them because 1) they are well recognized by the parallel programming community, 2) the MPI version is highly tuned and 3) they feature different communication patterns and computation/communication ratio during their executions. The programs are executed with two different data set sizes: class A and B.

We compare seven different implementations featuring identical timer locations. The MPI version will be compared with the naive loop level OpenMP version from SDSC, improved loop level OpenMP from RWCP, retranscripted in Fortran, optimized loop level OpenMP (PBN), and two SPMD OpenMP versions developed by ourselves.

The MPI Version is the NPB 2.3 [13], with only more timers than the 'standard' version. The same timers are used for all the other implementations. Several research studies have already presented detailed investigations of its scalability, data locality and communication properties [14] [1].
'Naive' Loop Level OpenMP: Several researchers have developed their own loop level OpenMP version of the NAS Benchmark, but all of them parallelize the same loop nests in the same way. Thus we have chosen the hybrid MPI + OpenMP that was developed by the San Diego Supercomputing Center (SDSC). All MPI calls are removed to get pure loop level OpenMP versions.
'Improved' Loop Level OpenMP: This version has been implemented from the serial version of the NPB by the RWCP. The original version is written in C. To ensure the fairness of the comparison, we have translated the C version into Fortran, applying the corresponding directives at the same locations and starting from the original Fortran code of the NPB-serial. The C version had been adapted to this language completely, thus the loop nests were reorganized considering the different array placement in memory between C and Fortran.
'Naive' SPMD OpenMP: Only two of the benchmark codes (MG and FT) have been tested using the naive implementation of the data exchanges. By measuring the performance of this version, our goal is to show the impact of communication tuning.
SPMD OpenMP: This is the 'real' SPMD version, which features the different modifications that will be explicited in section 4. Each benchmark has been carefully tuned. However the computation parts are kept exactly the same as the original MPI version.
'Optimized' Loop Level OpenMP: This last implementation, 'Programming Baseline for the NPB' (PBN), is rather different than the others. It has been developed from an optimized version of the NAS-serial. However, comparing it with all the other versions is relevant to evaluate the impact of the sequential code optimizations compared to the parallelization optimizations.

4. HIGH PERFORMANCE OPENMP SPMD

To allow a fair comparison with existing tuned MPI programs provided by external sources, we derive the SPMD OpenMP version from an existing MPI version of the program. Note that this may be the case for many programers since there is a lot of MPI programs and shared memory nodes are getting more popular. We follow a strategy consisting in 1) preserving the computation parts in the SPMD OpenMP version as close as possible to the ones of the original MPI version and 2) optimizing the data exchanges designing and comparing several shared memory based communication algorithms. Figure 1 presents the translation path to get a SPMD OpenMP program from a MPI version.

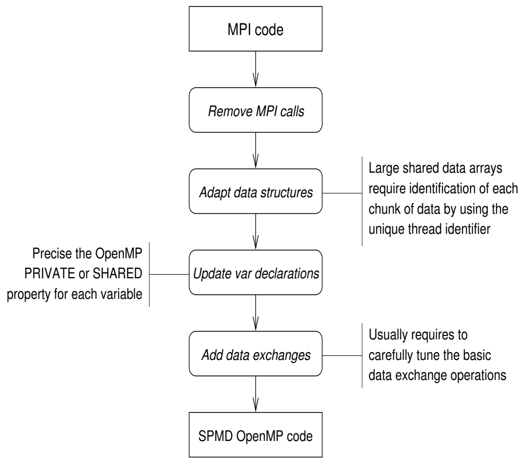
MPI code
Remove MPI calls
Adapt data structures
Large shared data arrays require identification of each chunk of data by using the unique thread identifier
Precise the OpenMP PRIVATE or SHARED property for each variable
Update var declarations
Add data exchanges
Usually requires to carefully tune the basic data exchange operations
SPMD OpenMP code

Figure 1: Translation path used to obtain high performance SPMD OpenMP programs from MPI ones

Preserving the computation parts of the original MPI programs can be considered as a first step in the programming effort. As we will show, reaching the performance of MPI on the computation part is not obvious and requires some dataset organisation rearrangement. In particular the programmer should compare the code generated by the compiler for the MPI and the equivalent SPMD OpenMP loop nests to understand the origin of the performance difference. In the rest of the paper, we will not go beyond this first step since the techniques to be used mainly concern loop nest optimizations which apply for the two programming models.

Replacing the MPI communications by shared memory data exchanges requires to implement and test different algorithms. We recommend the programmer to compare the different algorithms within his application instead of choosing the best one from synthetic benchmarks. Another criteria for selecting the communication algorithms is the simplicity of integration: for some programs we select an algorithm not because it is the best performer but because 1) of its relative simplicity of integration within the application according to the data set organization and 2) the consistency of data set organization regarding the other communication operation requirements.

4.1 Tuning the Computation Parts

The usual strategy for exploiting data locality in parallel programs consists in 1) distributing the working data sets among the processors to reduce communication needs and 2) expressing the memory references for each processor to limit cache misses. One way to express the data distribution among the threads is the declaration of smaller datasets using `!$OMP THREADPRIVATE`. However, two main reasons make us reject this approach: 1) efficient inter-thread communications require threads to read (or write) their neighbors datasets and 2) the way compilers handle threadprivate datasets may be inefficient. The first issue is presented in the next subsection.

Regarding the second issue, we noticed that some compilers do not handle properly large private arrays. For example, IBM XLF version 7.1 uses, for referencing every thread-private variable, a call to the internal function giving the unique thread identity number. These calls can be made at the innermost level of a loop nest leading to performance degradation.

To detect these problems, the programmer needs to control deeply the compiler code generation regarding OpenMP constructions. One way to circumvent the problem is to manage manually the distribution among the threads of the datasets and loop nest iterations. This implies 1) the use of a global shared array encompassing all the threads sub datasets and 2) the transformation of the references to the sub datasets by adding an index (the thread identity number). By removing sub dataset management from the innermost loop level, its overhead can be kept negligible. Figure 2 presents the organization of the memory.

However, using thread private variables for the scalar values such as loop bounds causes no problem. This is an excerpt of the typical declarations in a header file for a SPMP OpenMP program:

```
c-- Scalar variables are privatized --
      integer thread_id, bound
      common /private_scalars/ thread_id, bound
!$OMP THREADPRIVATE /private_scalars/
c-- Large working data arrays are shared
     double precision A(2000, num_threads)
```

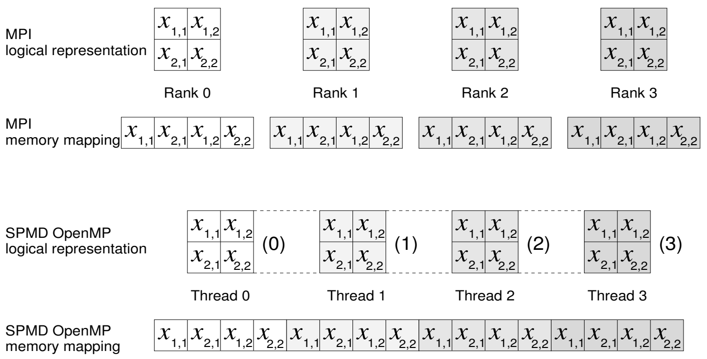
MPI logical representation
$x_{1,1}$ $x_{1,2}$ $x_{2,1}$ $x_{2,2}$
Rank 0
Rank 1
Rank 2
Rank 3
MPI memory mapping
$x_{1,1}$ $x_{2,1}$ $x_{1,2}$ $x_{2,2}$
SPMD OpenMP logical representation
(0)
(1)
(2)
(3)
Thread 0
Thread 1
Thread 2
Thread 3
SPMD OpenMP memory mapping

Figure 2: A private array of 2x2 numbers inside the shared memory, with 4 threads

```
      common /shared_arrays/ A
```

4.2 Tuning the Communication Parts

In this paper we focus on the data exchange operations required to implement the NAS benchmark considered in this study.

4.2.1 Point-to-Point Data Exchanges

This type of operation is basically an exchange of interfaces between data sets belonging to two different threads. Previous researches like [9] have proposed 'shadow arrays' to implement inter-thread communications and shown their advantages for some SPMD programs. For this approach, a shared variable is used as a communication buffer. To remove the need of buffering and implement a single copy data exchange operations, we use the shared property of the working data sets distributed among the threads. Communicating threads read/write directly from/into the working data sets of their communication peer. We ensure the data consistency by synchronizing the communicating threads. Figure 3 presents the time spent for the exchange of an hyperplane of 32x32 double precision numbers part of an hypercube of dimensions 32x32x32 on an IBM Night Hawk II multiprocessor (details about experimental conditions are presented in section 4). We consider MPI as the reference for this test and compare it with three OpenMP implementations: with buffer, single-copy with global synchronization (barriers), single-copy with synchronization between peers (locks).

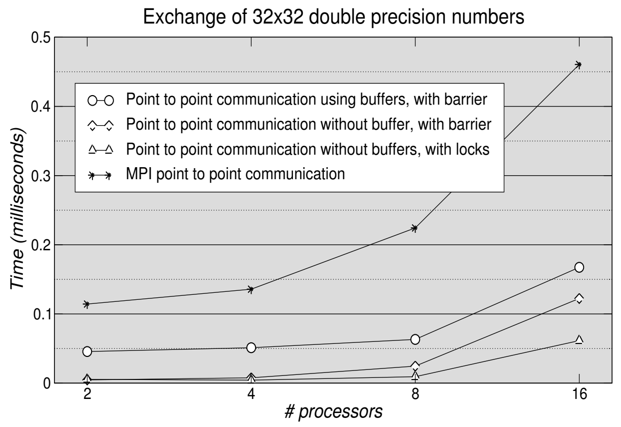

Figure 3: Point to point communication implementations on the IBM Night Hawk II

The best implementation in shared memory is the single-copy with locks. However, for the NAS benchmark we choose to use global synchronizations for two reasons: A) using locks in large programs can be quite difficult and B) the considered programs are regular applications with no load balancing management (all threads are supposed to exchange data after executing a similar computational job). In situation of nearly synchronous threads, figure 3 shows that the overhead of a global synchronization is quite small, up to eight processors.

Note that this model without buffers is usable only for translating synchronous send-recv operations. For the asynchronous communications, the programmer will need to use buffers like in MPI.

4.2.2 *Global Reduction*

Several algorithms of the global reduction operation have been tested: using the `REDUCTION` clause in a reduction loop, using the `ATOMIC` directive, computing a local reduction on each thread from shared arrays stored in the other threads and coordinated by barriers or locks. Figure 4 compares the tested implementations with `MPI_AllReduce(MPI_MAX)` for the reduction of an array of 32 double precision values.

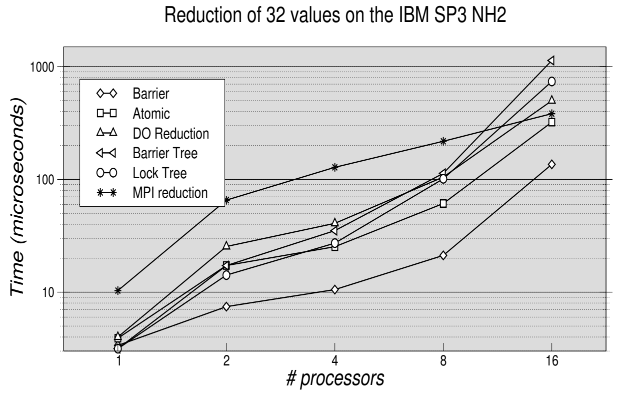

Figure 4: Comparison of the implementations of the reduction on the IBM NightHawk II

As seen in figure 4, the best OpenMP implementation uses the algorithm based on synchronization barriers, with a shared array of data as the input value, each thread making the reduction in a private variable. One of the worst one is based on the `REDUCTION` clause. The following code shows the best implementation, as equivalent to call MPI_AllReduce (BufferIn, BufferOut, 32, MPI_MAX, MPI_COMM_WORLD, ierr).

```
c-------- Reduction using a barrier -------
      double precision BufferIn(32,nthreads),
     >                 BufferOut(32)
      integer i
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(Buffer
     > Out, i)
      ...
!$OMP BARRIER
      do i = 1, 32
         BufferOut(i) = Max(BufferOut(i),
     > BufferIn(i))
      enddo
!$OMP END PARALLEL
c------------- End of program -------------
```

4.2.3 *All-to-All Operations*

The all-to-all is implemented using a global shared array and a main loop over the thread OpenMP identifiers. Each thread copies a chunk of data from its managed zone to the appropriate zones managed by the other threads. Figure 5 illustrates this implementation using the shared arrays as they have been described in section 4.1. Although simple, this operation gives good performance when tuned specifically for each program. Our experiments will consider two versions of the SPMD OpenMP programs (see Section 3.3): the first one uses the basic implementation (this is the 'naive' version of the algorithm), while the second one uses the tuned version.

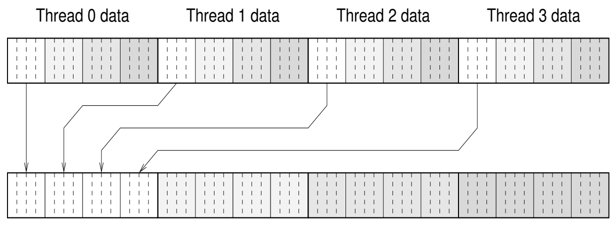

Figure 5: SPMD OpenMP implementation of the All-to-all operation, with 4 threads

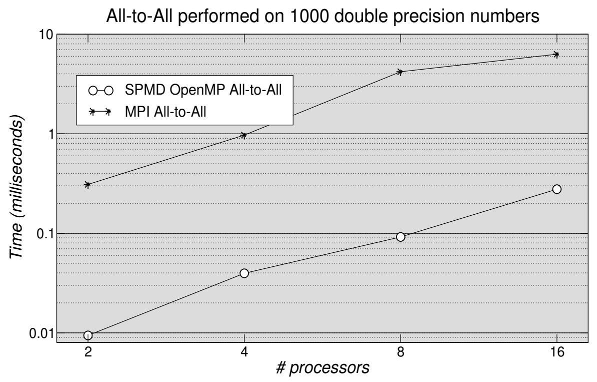

Figure 6: Performance of the MPI and SPMD OpenMP implementations of the All-to-All operation on the IBM Night Hawk II

Figure 6 presents the comparison of the performance of the all-to-all operation between the MPI implementation and the SPMD OpenMP one.

5. PERFORMANCE EVALUATION

5.1 Reference Measurements

The comparison will consider performance ratios between MPI and the OpenMP versions. Since this hides the basic scalability characteristics, we recall the parallel efficiency of the original NPB 2.3 on a single IBM NH2 node (7).

5.2 Raw Comparison Results

Figure 8 presents the OpenMP/MPI performance ratio, for the different OpenMP implementations and the 4 NAS programs on the IBM NightHawk II.

The first result concerns the uni-processor performance. Most of the 'non-optimized' implementations exhibit same performances. The 'optimized' version reaches significant performance improvement (from 10% to 30%) for MG, FT Class A and LU class A. For MG a buffer copy in the border communication has been removed giving about 20% improvement for both classes. 'Optimized' FT uses arrays with less dimensions, offering an improvement of 25% for class A

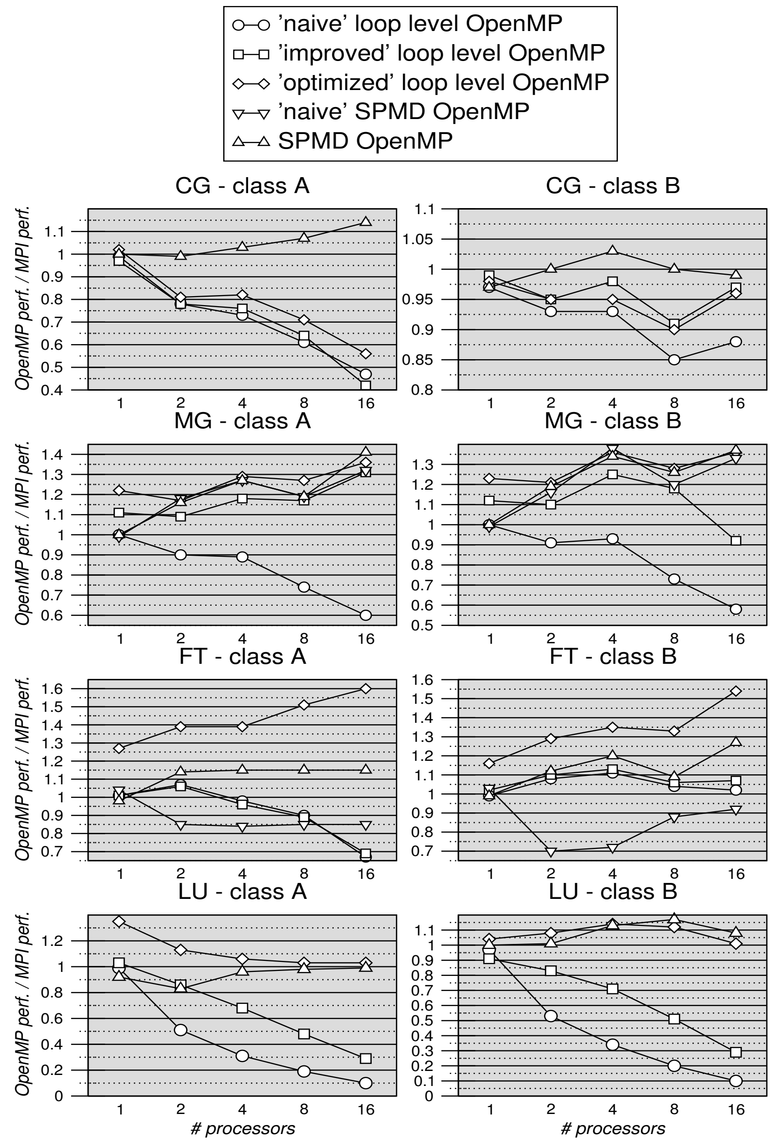

Figure 8: Performance of OpenMP implementations of the NPB on the IBM NH2 relatively to MPI

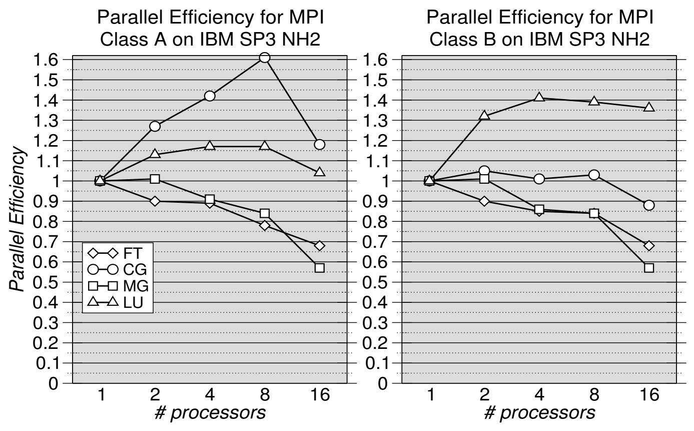

Figure 7: Parallel efficiency for the original MPI implementation of the NAS benchmark 2.3, class A and B, within a single IBM NH2 node.

and 15% for class B. LU uses a pipelined algorithm providing a 20% speedup for class A only. As section 6 will show, the performance advantage of the 'optimized' version on the computation part will remain effective for all multiprocessor configurations.

For CG class A, SPMD OpenMP provides a significant improvement over both MPI and other OpenMP versions (up to 15% over MPI, and 140% over 'naive' OpenMP). For CG class B, MPI and SPMD OpenMP provide similar performance. The gain over 'naive' OpenMP remains of about 15%.

For MG, the 'optimized' and 'improved' loop level versions provide good results (more than 30% better than MPI) for class A. For class B the 'improved' version is less efficient than the 'optimized' one. The SPMD OpenMP code is competitive with the others OpenMP versions for both classes.

For FT, the 'optimized' version outperforms the others including MPI. The improvement comes from a performance tuning of the code based on array shape modification. SPMD OpenMP provides a flat 15% improvement over MPI for class A and a non-uniform advantage (up to 25%) over MPI for class B. 'Naive' and 'improved' versions show a decreasing efficiency for class A and reach MPI performance for class B.

For LU, SPMD OpenMP provides only a little advantage (about 15%) over MPI for class B. The other OpenMP implementations show a high negative slope meaning that they do not scale.

Figure 9 presents the OpenMP/MPI performance ratio, for the various OpenMP implementations and for the 4 NAS programs on the SGI Origin 3800.

The same general trends apply on both machines up to 16 processors regardless of their architecture. Thus we discuss only the differences. For processor configurations larger than 16, almost all OpenMP implementations follow a negative slope (except SPMD OpenMP for LU). For FT, the 'optimized' version exhibits a parabolic curve leading to more performance reduction compared to MPI for processors larger than 16. 'Naive' and 'improved' OpenMP versions follow a decreasing curve from 4 processors on class B. For 32 and 64 processors for SPMD OpenMP, a platform-specific issue forces us to use a non-tuned version of the data exchanges.

The good performance of SPMD OpenMP compared to the others may seem obvious to obtain. However we insist that getting high performance with SPMD OpenMP requires significant programming efforts for the computation parts as well as for the communication parts of the programs.

6. DETAILED TECHNICAL RESULTS

In order to explain the performance of each implementation, we use timers to get the precise time spent in the computation and communication parts of the benchmarks. Furthermore, to precisely explain the computation numbers, we use the hardware performance counters of the test machines.

6.1 Breakdown of the Execution Time

As the main goal of our study is to make a comparison between MPI and OpenMP, all the results are represented relatively to MPI. Figure 10presents the time spent in the computation intensive parts of the code.

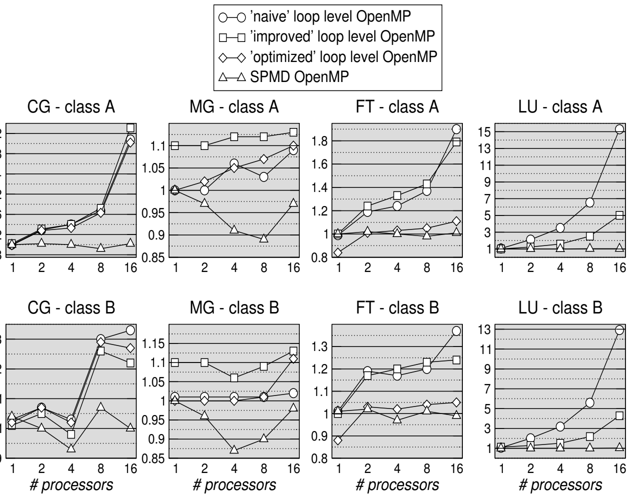

Figure 10: Execution time of OpenMP relatively to MPI for the computing parts on the IBM NH2

First, we notice from this figure (10) the poor performance and scalability on the computational part of all loop level OpenMP versions for most of the considered benchmarks. We will explain in the following part what are the underlying reasons for this. We can notice that the better performance of the PBN clearly comes from the optimization of the computational part and is not only due to the communication overhead.

The second point concerns the SPMD programs. As stated before (section 4), we try to keep the computation parts of the MPI and SPMD OpenMP implementations as close as possible. It appears that this is successful despite the modifications done for large shared data arrays. The computation times being very close, the performance differences come from the second factor: communication (or data exchange).

Figure 11 presents the communication/com-putation ratio for all the benchmarks on the IBM SP3. As for most of the versions, the loop level OpenMP programs do not use explicit communications, they are not represented. The only exception is MG which executes explicit data movements due to the torus structure of the problem.

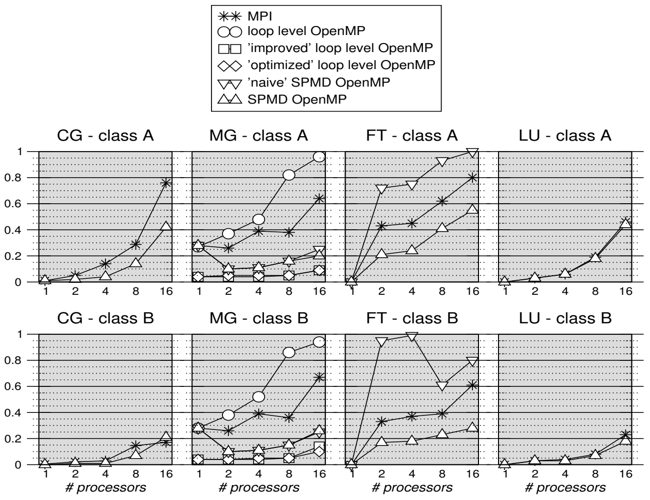

Figure 11: Communication/computation ratio of the 4 kernels on the IBM NightHawk II

By observing the communication/computation ratios (figure 11), we can understand the performance for each and every implementation.

For CG, MPI and SPMD OpenMP weaken their advantage over the loop level versions due to the increasing time spent for the data exchange. SPMD OpenMP overtakes MPI on class A thanks to the lower communication time. This advantage vanishes for CG class B from 16 processors. We have no explanation yet about the cause of this phenomenon, which can be put in relation with what is observed with our elementary reductions on figure 4.

MG is the only kernel where loop level OpenMP implementations perform explicit communication. Thus, the speedup efficiency of all versions decreases due to increasing communication cost. The communication time of the 'naive' loop level version explains its poor global performance. The global performance shift between the 'improved', 'optimized' versions and the others comes from a reduction of the communication complexity (figure 11). There is no difference in communication time between the OpenMP SPMD versions. The better communication performance combined with a similar computational part explains their advantage

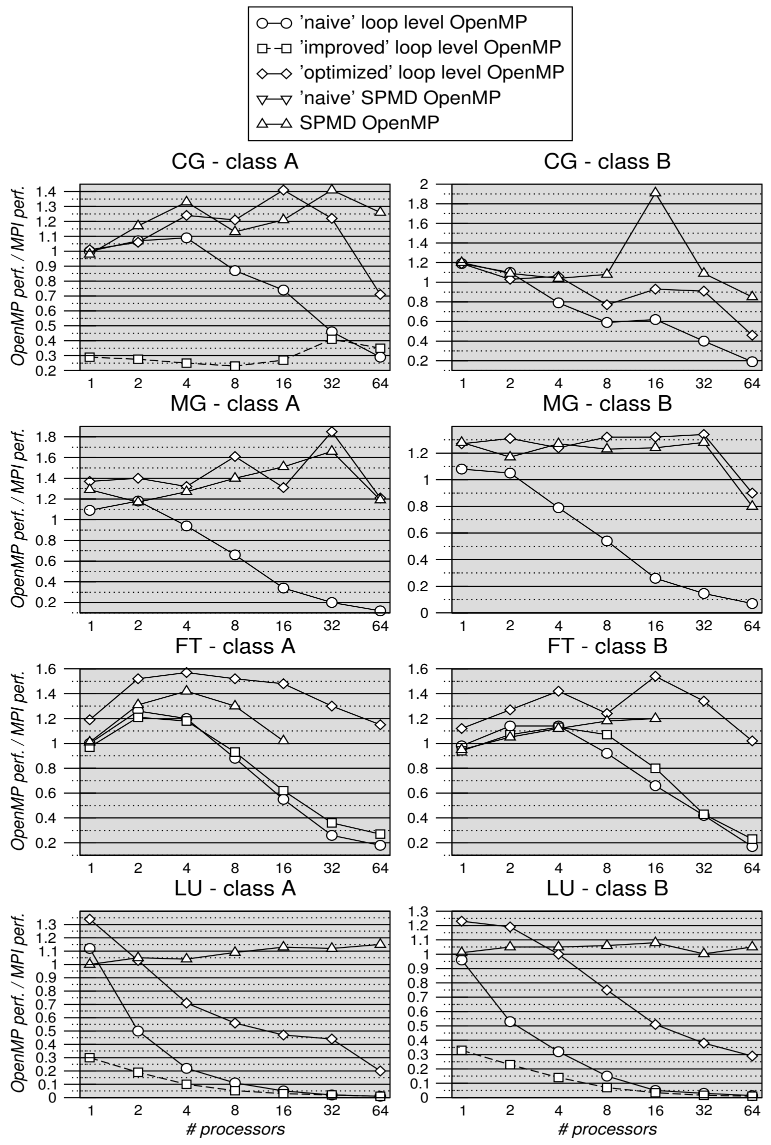

Figure 9: Performance of OpenMP implementations of the NPB on the SGI O3800 relatively to MPI

over MPI.

For FT, 'naive' and 'improved' OpenMP versions follow very similar trends in the computing part. Since there is no explicit data exchange for these implementations, their relative performance compared to MPI decreases as their relative execution time is increasing. The 'optimized' OpenMP implementation capitalizes on its 20% improvement over MPI and SPMD OpenMP for all number of nodes. Its computation time encompassing some 'implicit' data exchanges increases slightly with the number of nodes. The gain relies on the absence of explicit communications. In contrary to MG, FT exemplifies the impact on the communication time of the different data exchange implementations tested with SPMD OpenMP. The 'tuned' implementation outperforms the 'naive' one by 30%.

For LU, the time spent on the computing part is so high for 'naive' and 'improved' OpenMP implementations that it cannot compensate the increasing communication cost experienced by MPI and SPMD OpenMP. These two last versions reach similar performance on both the computing and communication parts.

For all the cases, we can notice that the communication/computation ratio is larger for MPI than for SPMD OpenMP. It is quite noticeable that the SPMD OpenMP version performs competitively compared to MPI even though 1) there is no attempt to optimize the virtual memory pages distribution and affinity with CPUs and 2) the SPMD version uses global arrays shared by all processors as for the loop level versions.

6.2 Hardware Performance Counters

In order to get further understanding of each version performance, we used the hardware performance counters of the two machines to collect L1 and L2 cache misses for the computation parts of the benchmarks.

Figure 12 gives the L1 and L2 data cache misses on the IBM NH2 for all experimental conditions. The result analysis of the SGI O3K is not presented here since it leads to the same conclusions. We still used the MPI values as references.

For CG, the parallelization of the outermost loop (row index) in all loop level versions leads every processor to ref-

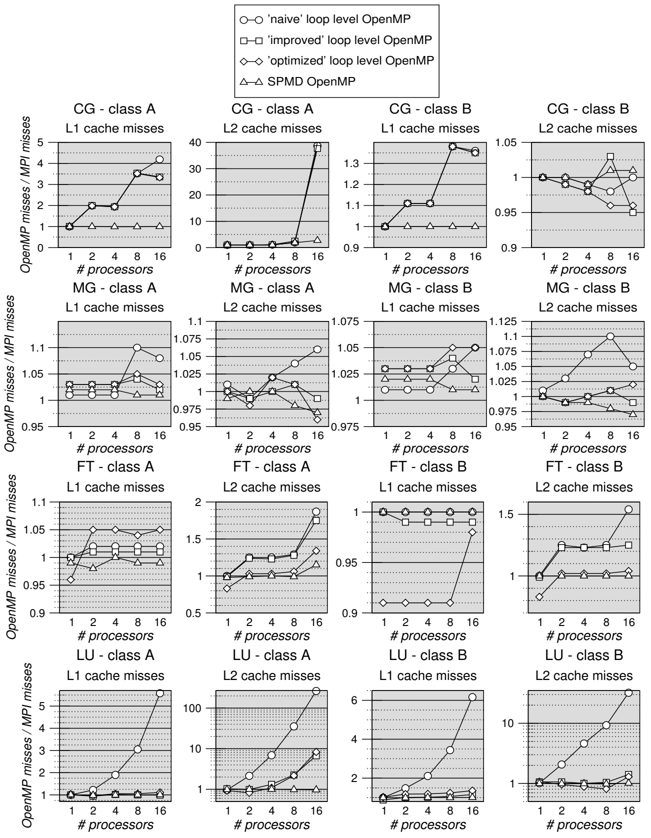

Figure 12: L1 and L2 cache misses for the computing parts of the 4 kernels on the NH2

erence the whole matrix and column index vector. SPMD (OpenMP and MPI) implementations distribute the matrix and column index vector (compressed row format) among the processors and assign more rows per processors. This distribution increases the spatial memory locality. Both SPMD approaches generate similar numbers of L1 and L2 cache misses.

For MG, the loop level versions distribute the 3 dimensional data sets among the processors as sets of 2 dimensional matrixes. In the SPMD versions, the data sets are distributed as 3 dimensional blocks. The cache misses results show a little variation between these two kinds of data distribution except for the 64 processors configuration. The differences between loop level and SPMD are even smaller for the IBM NH2. These results conform to the execution time on the computational part of the benchmark.

For FT the numbers of L1 and L2 cache misses relatively to MPI increase significantly for the loop level OpenMP version. The main computation is structured using four nested loops. The loop level OpenMP versions parallelize only the outermost one whereas the SPMD versions distribute the iterations of the two outermost ones, leading to a different memory reference pattern for all CPUs.

Loop level and SPMD codes for the LU application exhibit the same differences leading to a huge gap between them for L1 and L2 cache misses on the SGI Origin 3800 machine. For LU on the IBM NH2, we can notice that the cache misses numbers of the 'improved' version are close to the ones of the MPI version, while the computation time of the first is much larger. This is because the programmers have serialized two loops with dependencies, which degrades the computation time without increasing the number of cache misses.

If we put in relation the cache misses results and the execution times of the computation part we can conclude that one of the main flaws of loop level OpenMP is the bad utilization of the memory hierarchy. Since, as explained in 6.1, the SPMD and loop level OpenMP versions use the same shared array arrangement of the data sets, we can conclude that the poor performance of the loop level versions is not related only to the virtual memory pages distribution as observed by other studies. The poor performance is mainly due to the absence of parallel loop nest optimizations such as blocking (which can be exploited by MPI and SPMD OpenMP) in the loop level versions. Performance comparisons of the same benchmarks under several policies of virtual memory page placement on the SGI O3K confirm this conclusion.

7. RELATED WORK

Several researchers have studied the respective performance of MPI and OpenMP on shared memory machines. However none of them has gathered 1) a performance comparison of MPI and OpenMP considering several architectures, several dataset size, well accepted non trivial benchmarks and all the known OpenMP programming styles and 2) an indepth understanding of the performance differences.

The authors of [9] and [15] focus on NUMA computers. In [9], they mainly consider the specific issues of such platforms, like virtual memory pages allocation/migrations and resource sharing between the different processes running on the machine. The performance study shows the advantages of shadow arrays for small kernels using the SPMD OpenMP style. We extend the result demonstrating that OpenMP SPMD provides better performance than others OpenMP styles on well admitted benchmarks, several hardware platforms and a variety of dataset sizes.

In [15], the researchers are comparing a loop level version of the NAS benchmark (PBN) derived from a tuned sequential version against the NPB 2.3 MPI version and demonstrate similar performance on a SGI O2000. Our paper shows that the SPMD OpenMP version provides competitive performances. Moreover SPMD OpenMP, unlike the PBN, provides consistent outstanding performances across all benchmarks and platforms. In addition, we present an analysis of the performance results that enables further understanding of the respective merit of code optimization and OpenMP parallelization for the NPB.

In [16], MPI and OpenMP implementations of the SPECsies benchmark are compared. The experimental results show no performance difference between the two implementations. The authors conclude that this is due to the equivalence of efficiency of the way the two models exchange data between processors. Our experiments demonstrate that MPI an OpenMP implementations exhibit performance differences and that data exchange is an outstanding differentiating parameter.

In their study [7], the authors have compared the implementations of MPI and OpenMP for the Origin 2000. In their study, the authors compare the performance of the MPI and OpenMP implementations of a Finite Element Code. They conclude that the main advantage of the OpenMP version (i.e. the easiness of programming) is easily made a second argument when comparing with the performance of the MPI implementation. We have to pinpoint the fact that the authors used only a loop level OpenMP approach, and we have shown that it is in general the worst programming model as far as performance is concerned.

The authors of [10] compare two OpenMP implementations of a conjugate gradient solver on a Compaq ES40 computer (4-way Alpha multiprocessor). The first uses loop level parallelism while the second is based on the SPMD style. Results show a performance gain up to 17% for the SPMD version for 4 processors. For the same kind of solver (CG from the NPB) on 4 processors, we obtain a performance gain of 40% for the SPMD version over the loop level OpenMP version, at 16 processors the advantage is about 70%, and at 64 processors almost 85%.

An Ocean Model application is tested with the SPMD OpenMP and MPI models in [17]. This is interesting since MPI and SPMD OpenMP make use of the same parallelization scheme to produce high performance (domain decomposition). Results show that the SPMD OpenMP model is advantaged by the best way it handles communication, but also that none of the models performs better than the other one. We have shown that SPMD OpenMP provides better performance than MPI in many cases. We extend this result by examining 4 different applications, 2 architectures and 2 dataset sizes.

8. CONCLUSION

In this paper, we have presented a comparison between MPI and OpenMP on different multiprocessor machines and provided explanations for the relative performances of the different models. We have also presented a translation path from MPI to OpenMP SPMD and some optimizations for the computation and communication parts.

We have demonstrated that it is absolutely not obvious to get better performance with OpenMP codes compared to MPI ones on shared memory machines. The 'naive' loop level OpenMP version is simply not competitive compared to MPI and other OpenMP versions. The 'improved' loop level version which uses less parallel regions provides very limited performance improvement. The 'optimized' loop level OpenMP version provides good performance (but not consistent) and scalability on all the machines, but requires substential algorithmic and code optimizations. The nested parallelized version was not considered in this study due to its portability or performance issues. The analysis of all benchmark codes highlights that the performance problem of loop level OpenMP mainly comes from the loop nest parallelization. Tentatives for overhead reduction ('improved' loop level OpenMP) or full reprogramming ('optimized' loop level OpenMP) mostly fail to consistently improve the performance of standard loop level OpenMP, as they do not address this problem, or at the price of a significant algorithmic effort.

Currently, only SPMD programming with OpenMP can provide good performance consistently accross architectures, programs and dataset sizes. However, SPMD programming style is not enough because 1) the code generation of the compiler for handling thread private arrays may limit significantly the performance and 2) the performance of shared memory algorithms for interthread communications may be lower than the one of MPI. For reaching high performance, the programmer should carefully tune the data set organization (not only distribution) and interthread communications. The performance analysis (hardware performance counters and breakdown of the execution time) of our OpenMP SPMD version of the NAS benchmark demonstrates that we succeed in getting the same performance as the MPI version for the computing loops and better performance than the MPI version for the communication part.

9. REFERENCES

[1] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on IBM SP for the NAS Benchmarks. *Proc. of the international Conference on Supercomputing 2000 : High-Performance Networking and Computing (SC2000)*, 2000. http://www.sc2000.org/proceedings/techpapr/index.htm.

[2] P. Kloos amd F. Mathey and P. Blaise. OpenMP and MPI programming with a CG algorithm . In *Proceedings of the Second European Workshop on OpenMP (EWOMP 2000), http://www.epcc.ed.ac.uk/ewomp2000/proceedings.html*, 2000.

[3] A. Kneer. Industrial Mixed OpenMP/MPI CFD application for Practical Use in Free-surface Flow Calculations. In *International Workshop on OpenMP Applications and Tools, WOMPAT 2000, http://www.cs.uh.edu/wompat2000/Program.html*, 2000.

[4] L. Smith and M. Bull. Development of Mixed Mode MPI/ OpenMP Applications. In *International Workshop on OpenMP Applications and Tools, WOMPAT 2000, http://www.cs.uh.edu/wompat2000/Program.html*, 2000.

[5] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference.* Massachussets Institute of Technology Press, 1996.

[6] Jack Dongarra et al. *Message Passing Interface Forum.* www.mpi-forum.org/docs/docs.html, 1994.

[7] M.K. Bane et al. A Comparison of MPI and OpenMP Implementations of a Finite Analysis Code. In *Cray User Group, (CUG-2000) (Noordwijk, Netherlands, 22-26 May 2000)*, 2000.

[8] Kazuhiro Kusano, Shigehisa Satoh, and Mitsuhisa Sato. Performance Evaluation of the Omni OpenMP Compiler. *Lecture Notes in Computer Science*, 1940:403–414, 2000.

[9] B. Chapman, A. Patil, and A. Prabhakar. Performance Oriented Programming for NUMA Architectures. In Springer-Verlag Berlin Heidelberg, editor, *LNCS 2104, International Workshop on OpenMP Applications and Tools, WOMPAT 2001, West Lafayette, IN, USA*, 2001.

[10] P. Kloos amd F. Mathey and P. Blaise. OpenMP and MPI programming with a CG algorithm. In *Proceedings of the Second European Workshop on OpenMP (EWOMP 2000), http://www.epcc.ed.ac.uk/ewomp2000/proceedings.html*, 2000.

[11] E. Ayguade et al. NANOS: Effective Integration of Fine-grain Parallelism Exploitation and Multiprogramming, 1999.

[12] Yoshizumi Tanaka, Kenjiro Taura, Mitsuhisa Sato, and Akinori Yonezawa. Performance Evaluation of OpenMP Applications with Nested Parallelism. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 100–112, 2000.

[13] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Mail Stop T 27 A-1, Moffett Field, CA 94035-1000, USA, December 1995.

[14] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Proc. of international Conference on Supercomputing 1999 : High-Performance Networking and Computing (SC299)*, 1999.

[15] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance. In NASA Ames Research Center, editor, *Technical Report NAS-99-01*, 1999.

[16] B. Armstrong, S. Wook Kim, and R. Eigenmann. Quantifying Differences between OpenMP and MPI Using a Large-Scale Application Suite. In Springer-Verlag Berlin Heidelberg, editor, *LNCS 1940, ISHPC International Workshop on OpenMP: Experiences and Implementations (WOMPEI 2000)*, 2000.

[17] A. J. Wallcraft. SPMD OpenMP vs MPI for Ocean Models. In *First European Workshop on OpenMP - EWOMP'99, http://www.it.lth.se/ewomp99/programme.html*, 2000.