Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC

Patrick Carribault, Marc Pérache, and Hervé Jourdren

CEA, DAM, DIF, F-91297 Arpajon, France
{patrick.carribault,marc.perache,herve.jourdren}@cea.fr

Abstract. With the advent of multicore- and manycore-based supercomputers, parallel programming models like MPI and OpenMP become more widely used to express various levels of parallelism in applications. But even though combining multiple models is possible, the resulting performance may not reach expected results. This is mainly due to collaboration issues between the runtime implementations. In this paper, we introduce an extended taxonomy of hybrid MPI/OpenMP programming and a new module to the MPC framework handling a fully 2.5-compliant OpenMP runtime completely integrated to an MPI 1.3 implementation. The design and implementation guidelines enable two features: (i) built-in oversubscribing capabilities with performance comparable to state-of-the-art implementations on pure OpenMP benchmarks and programs, and (ii) the possibility to run hybrid MPI/OpenMP applications with a limited overhead due to the mix of two different programming models.

1 Introduction

The advent of multicore processors as basic-blocks of cc-NUMA (*cache-coherent Non-Uniform Memory Access*) architectures leads to the necessity to extract a large amount of parallelism. Parallel programming models have emerged to help programmers expressing several kinds of parallelism (task, data, etc.) at multiple granularity levels. One of the most commonly-used model is MPI [1], representing every task as process and using message-passing methods to communicate through explicit function calls (distributed memory paradigm). Another parallel programming model is OpenMP [2]: it enables multithreaded parallelization of any part of the program by adding directives to the sequential source code.

Hybrid programming with MPI and OpenMP (or *mixed-mode programming*) is a promising solution taking advantage of both models: for example, exploiting intra-node shared-memory with OpenMP and inter-node network with MPI. The MPI model allows to exploit a large number of cluster nodes. Furthermore, the intra-node communications are usually optimized thanks to shared memory. On a NUMA architecture, this programming model implicitly creates a well-balanced data locality because every task allocates its own set of data inside its private memory. On the other hand, OpenMP fully benefits from the shared memory available inside a node by avoiding any kind of communications: data are shared by default and the programmer has to guarantee the determinism of

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 1–14, 2010.

data races (with locks or any sort of synchronization mechanisms). This leads to a gain in memory and sometimes in performance but the main drawback is the data locality on a NUMA architecture: the OpenMP standard gives no rules about parallel memory allocation. Moreover, multithreading with OpenMP enables a lightweight load-balancing mechanism within a node. But even if these two models are relying on different paradigms, mixing them does not always lead to a performance gain. Side effects appear not only because of application semantics but because of the separate implementations resulting in some extra overhead. Several studies try to analyze their corresponding behavior but none of them proposed a unified representation of both models.

This paper presents an extension to the MPC framework [3] to create a unified representation of hybrid MPI/OpenMP parallelism. MPC is a thread-based framework unifying parallel programming models for HPC. It already provides its own thread-based MPI implementation [4]. To enable hybrid parallelism, we added an OpenMP runtime and optimized it to lower the overhead due to programming-model mixing. This paper makes the following contributions: (i) the implementation of a full OpenMP runtime with oversubscribing capabilities, (ii) an extended taxonomy about approaches to adopt hybrid parallelism and (iii) a unified runtime for low-overhead hybrid MPI/OpenMP parallelism.

This paper is organized as follows: Section 2 describes the related work for these parallel programming models. Section 3 exposes an extended taxonomy of the hybrid MPI/OpenMP programming model. It details the different ways to combine these two models and their issues. Section 4 depicts the implementation of hybrid parallelism in MPC, including the design and implementation of the OpenMP runtime. Finally, Section 5 proposes some experimental results on both OpenMP and hybrid benchmarks, before concluding in Section 6.

2 Related Work

Hybridization of parallel programming models has already been explored and tested to exploit current architectures. Some applications benefit from combining MPI and OpenMP programming models [5] while some work concludes that the *MPI everywhere* or *Pure MPI* mode is the best solution [6]. This is because several issues appear when dealing with these programming models. For example, the mandatory thread-safety of the underlying MPI implementation might lead to an additional overhead [7,8]. Furthermore, the way the application combines MPI and OpenMP directly influence the resulting performance. For example, the inter-node network bandwidth may not be fully used if only a small number of threads or tasks are dealing with message-passing communications [9]. To avoid this limitation, Viet *et al.* studied the use of thread-to-thread communications [10]. The OpenMP model can be as well extended to increase the support of other programming models. Jin *et al.* explored the notion of *subteams* with different ways of using OpenMP to increase hybrid performance [11].

To the best of our knowledge, one of the most advanced study on hybridization has been proposed by Hager *et al.* [12]. They designed a taxonomy on the ways

to combine MPI and OpenMP inside the same scientific application. But all these approaches and tests only deal with simple hybrid parallelism. Indeed, they hardly explored the potential benefit of using MPI and OpenMP with oversubscribing.

3 Hybrid Programming MPI/OpenMP

This section describes the multiple ways to mix these two programming models by extending the taxonomy introduced in [12] and details some issues related to the implementation of such models.

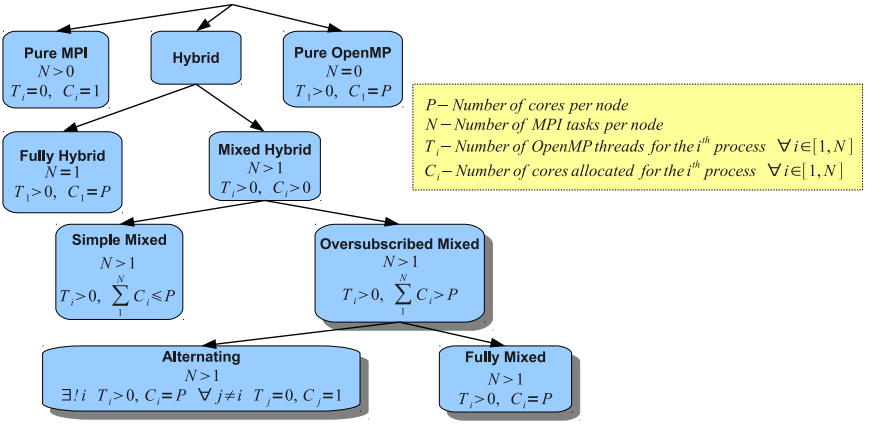


Fig. 1. Extended Taxonomy of Hybrid MPI/OpenMP Programming Models

3.1 Taxonomy of Hybrid Applications

Figure 1 proposes an extended taxonomy of hybrid parallelism [12]. We consider nodes with P cores. The classification is related to the number of MPI tasks N located on each node, the number of OpenMP threads T_i for the i^{th} process and the number of cores allocated for the i^{th} process C_i i.e., the cores on which the OpenMP threads will be spawned during the parallel regions of the i^{th} process.

Starting from the root, the first level of this tree illustrates the basic approaches: the *Pure MPI* mode uses the MPI programming model ($N > 0$) and creates no OpenMP thread at all ($T_i = 0$). On the other hand, the *Pure OpenMP* mode only spawns OpenMP threads to parallelize the application. This mode can be extended to multiple nodes [13]. The *Hybrid* subtree represents the combination of MPI and OpenMP inside the same application. The *Fully Hybrid* approach exploits the node with OpenMP threads ($T_i > 0$) and uses message-passing functions to communicate over the whole cluster ($N = 1$). The overall performance mainly relies on OpenMP parallelization on each node and might therefore be limited by Amdahl's law both because of the remaining sequential

parts (related to the fork-join model of OpenMP) and the serialized MPI communications. Deploying more than one MPI task on the node is called *Mixed Hybrid*. In this mode, it is possible to either exploit each core with one execution flow (*Simple Mixed* where the total number of allocated cores is at most P) or oversubscribe the cores to recover latencies (*Oversubscribed Mixed* subtree). The first solution is commonly used when dealing with hybrid parallelism. For example, each multicore processor can be exploited with one MPI task spawning OpenMP threads on it. The oversubscribed approaches are less explored: one advantage is the ability to recover different kinds of latencies (load-balancing, MPI communications, system calls, etc.) within the application.

This paper introduces the *Oversubscribed Mixed* subtree with two approaches: the *Alternating* mode where the application alternatively uses MPI and OpenMP parallelization, and the *Fully Mixed* mode where one MPI task is placed by core and all tasks might exploit the entire node when entering a parallel region. These approaches advocate for a solid and coherent implementation of MPI and OpenMP handling oversubscription.

3.2 Programming Model Interleaving

The taxonomy depicted in Figure 1 sorts several functional approaches to mix OpenMP threads and MPI tasks. But even if these models are not relying on the same paradigms, combining them in practice can be very tricky. First of all, the MPI library has to handle multiple concurrent control flows. For this purpose the MPI 2.0 standard introduces several levels of thread support: to fully use any of the approaches described in Figure 1, the underlying library has to support the `MPI_THREAD_MULTIPLE` level. Moreover, the task/thread placement is a real issue. Deciding where to pin such MPI task or such OpenMP thread mainly depends on the use of these models. If the application spends almost 100% of its consumed time in OpenMP parallel regions, the right solution is probably to span the OpenMP threads on each socket and use, for example, one MPI task per multicore processor (*Simple Mixed* mode). But this is not always the best approach to use hybrid programming. If one wants to use OpenMP threads to balance the load among the MPI tasks, spanning the threads everywhere and overlapping the OpenMP instances might be one approach to reach higher performance (*Oversubscribed Mixed* subtree in Figure 1). Because of this last point, the MPI and OpenMP runtimes must cooperate to establish some placement and oversubscribing policies. Indeed, load balancing can be obtained only if the corresponding runtimes allow the scheduling of one task and several threads on each core without a too large overhead. For example, if both runtime implementations use *busy waiting* to increase the reactivity and the performance of one programming model, the combination of several models might lead to an extremely large overhead, disabling the potential performance gain. All these issues are highlighted and evaluated on MPI/OpenMP implementations in Section 5.

4 MPC-OpenMP: MPC's OpenMP Runtime

To enable hybrid parallelism in MPC [3,4], we add a complete OpenMP runtime through a new module called MPC-OpenMP supporting the 2.5 standard.[1] This section details the design and implementation of the MPC-OpenMP module and its integration with the other programming models supported in MPC (Pthread, MPI).

4.1 Runtime Design and Implementation

MPC already provides a thread-based implementation of the MPI 1.3 standard [4]: it represents every MPI task/process with user-level threads (called *MPC tasks*). Thanks to *process virtualization*, every MPI task is a thread instead of being a UNIX process. The main guideline to support the OpenMP standard is to model the threads like MPI tasks. In such way, the main scheduler would see every thread and task the same way. Another constraint is the weight of OpenMP threads: it is possible to often enter and exit parallel regions with the OpenMP model. Therefore, each thread has to be very light and to be ready within a short period of time. For this purpose, we introduce in MPC a new kind of thread called *microVP* (micro Virtual Processor) scheduling its own *microthreads*. Microthreads can be seen as some sort of filaments [14] or lazy threads [15]: entering the parallel region, OpenMP threads are distributed as microthreads among microVPs. At the very beginning, microthreads do not have their own stack, they use the microVP's one when they are scheduled. This accelerates the context switches. Stacks are created on-the-fly if one thread encounters a scheduling point (e.g., any synchronization point like a barrier).

Figure 2 depicts an example of the MPC execution model when using the OpenMP programming model in the *Pure OpenMP* mode. In this example, the underlying architecture has 4 cores modeled in MPC through Virtual Processors. The main sequential control flow is managed by the MPC task on the left of the figure. This task is scheduled on the first core. When starting the application, the MPC-OpenMP module creates microVPs for each task. In this example, because there is one control flow (one task on the first core), 4 microVPs are created, one on each core. When entering an OpenMP parallel region, the MPC task distributes microthreads on the microVPs. The figure illustrates the execution model after entering a parallel region with 6 OpenMP threads. The first and second cores have to schedule 2 microthreads while the other ones only have 1 microthread to execute.

The MPC-OpenMP runtime contains optimizations related to oversubscription (i.e., more than on microthread per microVP). For example, every barrier among microthreads on the same microVP can be resolved without any synchronization mechanism.

[1] MPC 2.0 including the MPC-OpenMP module is available at `http://mpc.sourceforge.net`

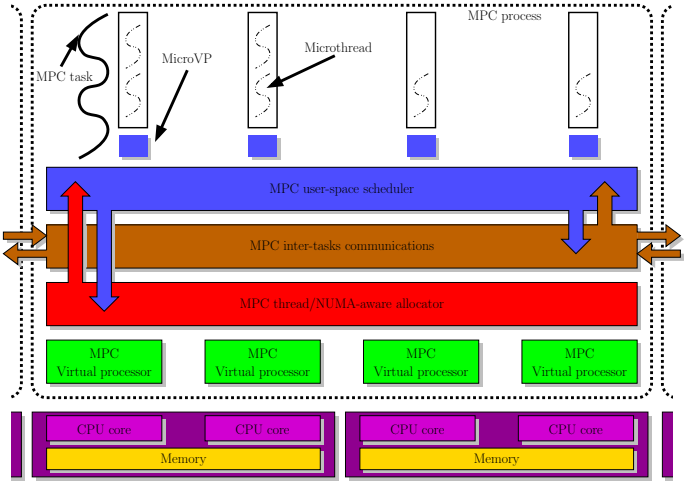


Fig. 2. MPC Execution Model of *Pure OpenMP* mode (6 OpenMP Threads on 4 Cores) adapted from [3]

4.2 Compiler Part

A compliant OpenMP implementation relies on a compiler to lower the source-code directives (e.g., `#pragma`). We modified the GCC compiler chain: instead of generating code for the GCC OpenMP library (called GOMP), the OpenMP directives are transformed into MPC internal calls (functions + Thread Local Storage). This patched GCC, called *MPC-GCC*, is available for versions 4.3.0 and 4.4.0. Thanks to these modifications, MPC supports the OpenMP 2.5 standard for C, C++ and Fortran languages.

4.3 Hybrid Execution Model

Hybrid programming is a combination of MPC tasks and threads. Each MPI task is modeled through an MPC task (user-level thread) and each instance of OpenMP is controlled by a set of microVPs scheduling microthreads when entering a parallel region. Each MPI task has its own set of microVPs to span the OpenMP threads on the node. The way these microVPs are initially created directly influences the mixing depth of the programming models. If every MPI task creates microVPs on the cores located on the same CPU socket, every OpenMP parallel region will span on the same multicore processor (*Simple Mixed* mode). This feature allows to limit the oversubscribing factor and maximizes locality. Figure 2 can be seen as an example of *Fully Hybrid* approach if the MPC task is actually an MPI task.

To enable hybrid parallelism with low overhead, some optimizations have been performed on the OpenMP and MPI parts of MPC. First of all, MPC has been

extended with `MPI_THREAD_MULTIPLE` features. The MPI standard requires to respect the message posting order. Nevertheless, once the matching phase is done, the message copies from the send buffer into the receive buffer can be performed concurrently. MPC-MPI has been optimized to parallelize the copy phase allowing multiple message copies in parallel of matching. Communications involving small messages were optimized using an additional copy into a temporary buffer to immediately release the sender. Initially, there was a unique buffer for each MPI task. This buffer has been duplicated for each microVP and the lock has been removed. MPC-OpenMP has also been optimized to deal with MPC-MPI. Busy waiting (e.g., when spawning parallel region) has been replaced by a polling method integrated in the MPC user-space scheduler leading to more reactivity without disturbing the other tasks.

5 Experimental Results

This sections describes the experiments to evaluate and validate the hybrid capabilities of MPC according to the taxonomy Figure 1. We first present the performance of the OpenMP runtime and the MPC-GCC compiler. Then, we detail the results of combining the MPI and OpenMP models on several representative benchmarks.

5.1 Experimental Environment

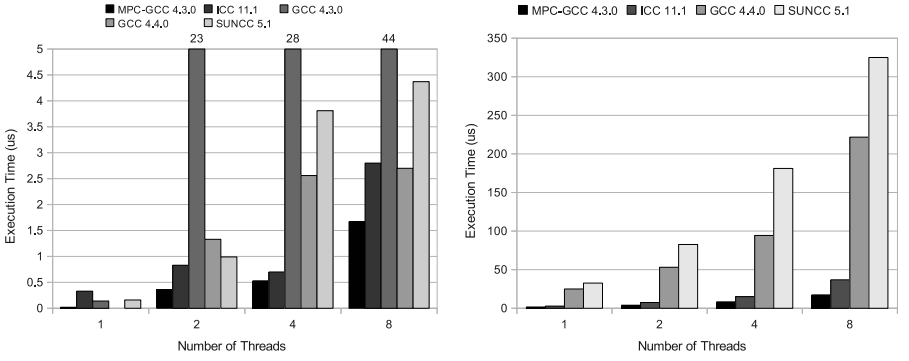
The experimental results detailed in this paper were obtained on a dual-socket quad-core Nehalem EP node[2] (8 cores) with 24GB of memory running under a Linux operating system. For lack of space, we only present the results on this architecture. We conducted the same experiments on Core2Duo and Core2Quad architectures and observed similar behaviors. For every benchmark, we bind the MPI tasks through the `schedaffinity` interface or other options (e.g., environment variables for IntelMPI). For example, IntelMPI provides several environment variables to place the tasks on the node and to reserve some cores for every task according to the topology of the underlying architecture.

5.2 Performance of MPC-OpenMP and MPC-GCC

This section describes the results of the OpenMP runtime implemented in MPC in *Pure OpenMP* mode. The compiler part is done through MPC-GCC version 4.3.0 and 4.4.0. Our implementation is measured against state-of-the-art compilers with their associated OpenMP runtime: Intel ICC version 11.1, GCC version 4.3.0 and 4.4.0, and Sun Compiler version 5.1.

EPCC Micro-Benchmarks: EPCC is a benchmark suite to measure the overhead of every OpenMP construct [16]. For each implementation, we tried the best

[2] Hyper-Threading and Turbo Boost are disabled.

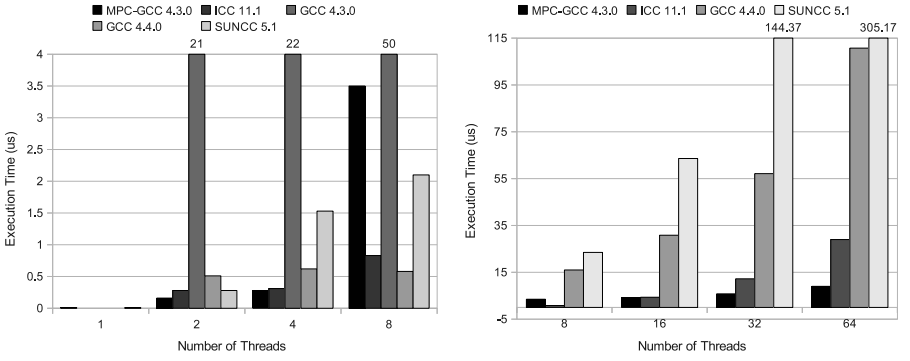


-a- From 1 to 8 threads -b- From 8 to 64 threads

Fig. 3. Overhead of OpenMP Parallel Construct on 8-core Nehalem Architecture

option combination (CPU affinity, waiting mode, ...). Figure 3 shows the overhead of opening and closing a parallel region (`#pragma omp parallel`) according to the total number of OpenMP threads. Figure 3-a depicts the overhead from 1 to 8 threads. Performance of MPC-GCC 4.3.0 and MPC-GCC 4.4.0 are similar: for this benchmark, only the OpenMP runtime shapes the performance, not the compiler. The performance of MPC are slightly better than the OpenMP implementation of ICC 11.1, GCC 4.4.0 and SunCC 5.1 though. GCC 4.3.0 gets some very surprising results: the overhead of almost every OpenMP construct is far behind the other implementations we tested. Thus, it does not appear on Figure 3-b. This figure describes the oversubscribing performance of the OpenMP runtimes. We tested up to an oversubscribing factor of 8 (i.e., 64 threads on 8 cores). The overhead of the parallel construct in the GCC 4.4.0 and SunCC implementations become very large compare to MPC, ICC. MPC implementation stays linear while ICC OpenMP results in an exponential overhead. Note these figures depict results with the best option combination. For example, we use the `SUNW_MP_THR_IDLE` option to specify the waiting mode of SunCC and the *passive mode* (`OMP_WAIT_POLICY` environment variable) for GCC 4.4.0 for the oversubscribing mode. This is the most suitable behavior for hybrid parallelism but this adaptation has a huge impact on the raw OpenMP performance: the overhead of entering and exiting parallel regions is 10 times larger in the passive mode without oversubscribing.

Figure 4 depicts the time consumed (in micro-seconds) by an explicit barrier (`#pragma omp barrier`) according to the number of threads. The same remarks can be made on the performance of GCC 4.3.0. The MPC-OpenMP runtime has a larger overhead on 8 cores than ICC, GCC 4.4 and SunCC. This is probably due to the NUMA aspect of the architecture. Even though the MPC runtime has NUMA-aware features, the barrier implementation does not fully exploit them. It might be interesting to implement a more efficient algorithm [17]. Nevertheless, the MPC-OpenMP oversubscribing performance depicted in Figure 4-b shows strong improvements compared to ICC, GCC and SunCC.



-a- From 1 to 8 threads -b- From 8 to 64 threads

Fig. 4. Overhead of OpenMP Barrier Construct on 8-core Nehalem Architecture

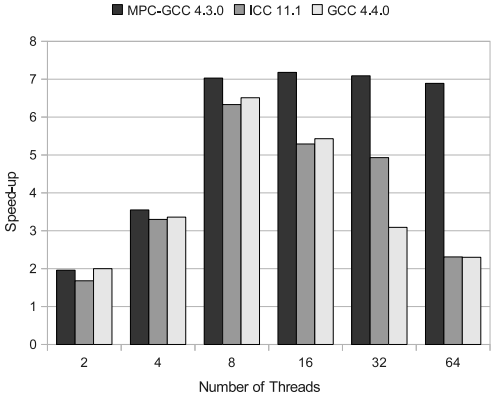


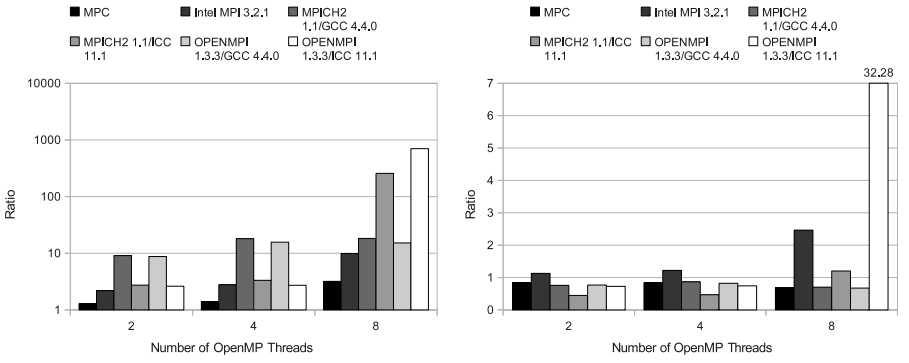
Fig. 5. BT Benchmark Results on 8-core Nehalem Architecture

NAS Benchmarks: Figure 5 shows the speed-ups obtained for the BT benchmark of the NAS 3.3 OpenMP suite [18]. Each acceleration is related to the corresponding single-threaded OpenMP version. MPC-GCC 4.4.0 obtains the same performance as MPC-GCC 4.3.0. The oversubscribing performance of MPC (with a number of threads larger than 8) are stable compared to other OpenMP implementations (ICC and GCC). Running this benchmark with 16 threads (i.e., 2 OpenMP threads per core) leads to slightly better performance than 8 with MPC. Indeed, the overall best speed-up is 7 reached by MPC with an oversubscribing factor of 2.

5.3 Overhead of Hybrid Parallelism

To evaluate the overhead of hybrid MPI/OpenMP parallelism, we designed a set of benchmarks related to the taxonomy depicted in Figure 1. Parts of these

benchmarks are directly extracted from [19]. We compare combinations of MPI libraries and OpenMP runtimes/compilers: IntelMPI, MPICH2 1.1 and OPENMPI 1.3.3 with ICC 11.1 and GCC 4.4.0. The target architecture is the one defined in Section 5.1.



-a- 1KB messages -b- 1MB messages

Fig. 6. Overhead of Funneled Hybrid Ping-Pong on Nehalem Architecture

Funneled Hybrid Ping-Pong: Figures 6-a and 6-b depict the overhead of hybrid MPI/OpenMP parallelism for an hybrid Ping-Pong benchmark [19] with MPI communications made by the master thread of the OpenMP region. The MPI library is required to be initialized with the `MPI_THREAD_FUNNELED` level. These plots illustrate the ratio of running a funneled ping-pong according to the performance of the single threaded version. OpenMP regions are used to read and write a buffer while the MPI communications are done by the master thread after an OpenMP barrier. Figure 6-a depicts the ratio (overhead) for messages of 1KB length. MPC has a low overhead compared to other MPI/OpenMP combinations: with 2 or 4 threads (*Simple Mixed* mode) the performance remains stable while it only increases a little with 8 threads (*Fully Mixed* mode).

Figure 6-b illustrates the same results for 1MB messages. Several implementations are able to reach better performance thanks to hybrid programming (ratio below 1). Runs with GCC OpenMP obtain decent speed-up but this is partly because it uses a passive waiting mode (for OpenMP) leading to low performance for OpenMP worksharing constructs. Nevertheless, MPC is the only hybrid implementation to reach speed-ups for all number of threads.

Multiple Hybrid Ping-Pong: The second benchmark is an hybrid ping-pong [19] with communications done by every thread inside the OpenMP parallel region. The MPI library is required to be initialized with the `MPI_THREAD_MULTIPLE` level. Figure 7-a depicts the ratio for 1KB messages compared to the single-threaded version. Note OPENMPI is not represented on this plot because we had some issues with its thread safety. The overhead of MPC is low compared

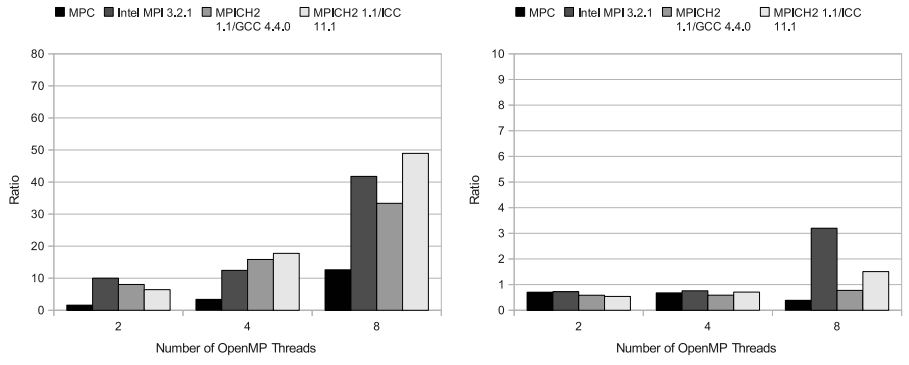
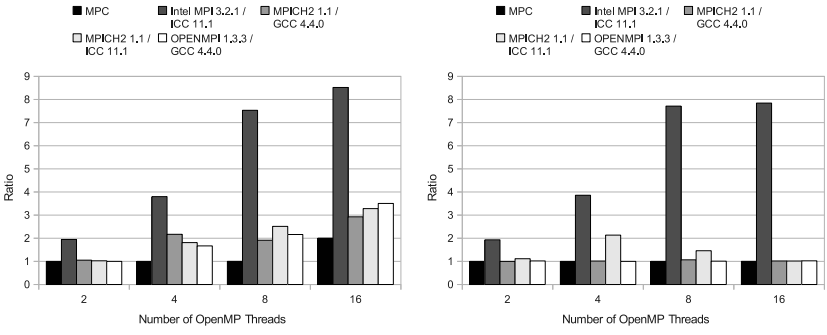


Fig. 7. Overhead of Multiple Hybrid Ping-Pong on Nehalem Architecture

to combinations including MPICH2 and IntelMPI. Indeed, it does not exceed 12 with 8 threads per MPI task while it reaches up to 50 for some implementations. Figure 7-b shows the overhead for 1MB messages. Most of the combinations benefit from hybrid parallelism to accelerate the ping-pong benchmark (ratio < 1). But MPC reaches the best performance with 8 threads per MPI task while MPICH2 and OPENMPI degrade their performance (*Oversubscribed Mixed* mode).

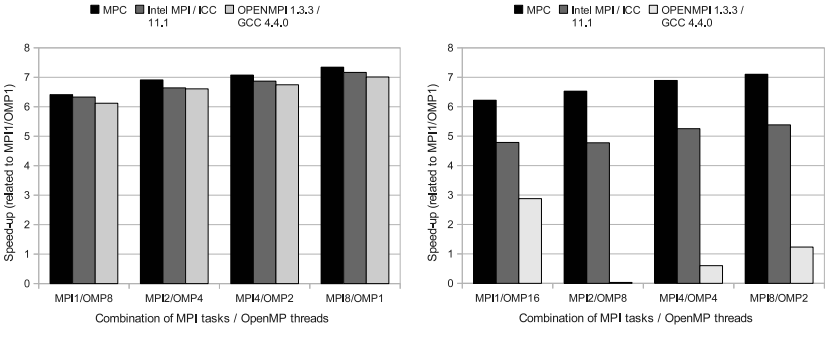
Alternating Benchmark: The last benchmark illustrates the *Alternating* mode depicted in the taxonomy tree Figure 1. Figure 8-a pictures the overhead ratio of one MPI task entering a parallel region (the threads performing independent computations) while the other 7 tasks are waiting on a barrier. While the MPC overhead is almost null up to 8 threads, other MPI/OpenMP combinations involve an overhead. Especially IntelMPI with the best performance options may degrade the execution time by a factor of 8, just because some MPI tasks are waiting on a barrier during the computational part of the single OpenMP parallel region. On the other hand, Figure 8-b shows the overhead of OpenMP computations done by the master thread of every MPI task at the same time. The overall ratio is smaller than for the previous *Alternating* benchmark except for the IntelMPI library (with ICC OpenMP) which reaches a slowdown of 8.

NAS-MZ Benchmark: Figure 9 depicts the results of running the hybrid version BT: BT-MZ version 3.2 with class B. These figures represent a total of 8 threads and a total of 16 threads (number of MPI tasks times number of OpenMP threads per task). Even though some versions reach speed-ups, the best acceleration is done thanks to 8 MPI tasks with one (*Pure MPI* mode) or 2 threads per task (*Oversubscribed Mixed* mode). Performance of MPC combining MPI and OpenMP is comparable to state-of-the-art implementations even on non-intensive hybrid benchmarks.



-a- MPI Tasks Waiting -b- OpenMP Threads Waiting

Fig. 8. Overhead of *Alternating* Benchmark on 8-core Nehalem Architecture



-a- Total of 8 threads -b- Total of 16 threads

Fig. 9. Speed-up of BT-MZ on Nehalem Architecture

6 Conclusion and Future Work

This article introduces an extension to the MPC framework to enable a unified representation of hybrid MPI/OpenMP parallelism. The new OpenMP implementation shows performance comparable to state-of-the-art implementations on current architecture (Nehalem) with efficient support of oversubscribing. This feature might be one key for future-generation hybrid applications. Moreover, we propose an extended taxonomy to explore new ways to hybridize applications and we test them on different benchmarks. The results show that MPC enables a low-overhead combination of MPI and OpenMP. MPC 2.0 including these hybrid capabilities is available at `http://mpc.sourceforge.net`

For future work, several optimizations can be done on the OpenMP runtime. For example, the barrier algorithm might be improved [17]. Other approaches can be tested inside the MPC framework like dynamic load-balancing of MPI applications with OpenMP [20] or subteam utilization [11]. OpenMP 3.0 is

becoming a new standard introducing the notion of OpenMP task and the ability to easily balance an application. Supporting OpenMP 3.0 is a future direction. Finally, hybrid performances have to be evaluated on large-scale applications.

References

1. MPI Forum: MPI: A message passing interface standard (March 1994)
2. OpenMP Architectural Board: OpenMP API (2.5 and 3.0) (May 2008)
3. Pérache, M., Jourdren, H., Namyst, R.: MPC: A unified parallel runtime for clusters of NUMA machines. In: Luque, E., Margalef, T., Benítez, D. (eds.) Euro-Par 2008. LNCS, vol. 5168, pp. 78–88. Springer, Heidelberg (2008)
4. Pérache, M., Carribault, P., Jourdren, H.: MPC-MPI: An MPI implementation reducing the overall memory consumption. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/PVI 2009. LNCS, vol. 5759, pp. 94–103. Springer, Heidelberg (2009)
5. Chen, L., Fujishiro, I.: Optimization strategies using hybrid MPI+OpenMP parallelization for large-scale data visualization on earth simulator. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 112–124. Springer, Heidelberg (2008)
6. Lusk, E.L., Chan, A.: Early experiments with the OpenMP/MPI hybrid programming model. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 36–47. Springer, Heidelberg (2008)
7. Thakur, R., Gropp, W.: Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE. In: Cappello, F., Herault, T., Dongarra, J. (eds.) PVM/MPI 2007. LNCS, vol. 4757, pp. 46–55. Springer, Heidelberg (2007)
8. Gropp, W.D., Thakur, R.: Issues in developing a thread-safe MPI implementation. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) PVM/MPI 2006. LNCS, vol. 4192, pp. 12–21. Springer, Heidelberg (2006)
9. Rabenseifner, R.: Hybrid parallel programming: Performance problems and chances. In: Proceedings of the 45th CUG (Cray User Group) Conference (2003)
10. Viet, T.Q., Yoshinaga, T., Sowa, M.: Optimization for hybrid MPI-OpenMP programs with thread-to-thread communication. Institute of Electronics, Information and Communication Engineers (IEICE) Technical Eeport, 19–24 (2004)
11. Jin, H., Chapman, B., Huang, L., an Mey, D., Reichstein, T.: Performance evaluation of a multi-zone application in different openmp approaches. Int. J. Parallel Program. 36(3), 312–325 (2008)
12. Hager, G., Jost, G., Rabenseifner, R.: Communication characteristics and hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proceedings of Cray User Group (2009)
13. Hoeflinger, J.: Extending openmp to clusters (2006)
14. Engler, D.R., Andrews, G.R., Lowenthal, D.K.: Filaments: Efficient support for fine-grain parallelism. Technical report (1994)
15. Goldstein, S.C., Schauser, K.E., Culler, D.E.: Lazy threads: implementing a fast parallel call. J. Parallel Distrib. Comput. 37(1), 5–20 (1996)
16. Bull, J.M., O'Neill, D.: A microbenchmark suite for OpenMP 2.0. SIGARCH Comput. Archit. News 29(5), 41–48 (2001)
17. Nanjegowda, R., Hernandez, O., Chapman, B.M., Jin, H.: Scalability evaluation of barrier algorithms for OpenMP. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 42–52. Springer, Heidelberg (2009)

18. Jin, H., Frumkin, M., Yan, J.: The OpenMP implementation of NAS parallel benchmarks and its performance. Technical Report: NAS-99-011 (1999)
19. Bull, J.M., Enright, J.P., Ameer, N.: A microbenchmark suite for mixed-mode OpenMP/MPI. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 118–131. Springer, Heidelberg (2009)
20. Corbalán, J., Duran, A., Labarta, J.: Dynamic load balancing of MPI+OpenMP applications. In: ICPP, pp. 195–202. IEEE Computer Society, Los Alamitos (2004)