A Microbenchmark Suite for Mixed-Mode OpenMP/MPI

J. Mark Bull, James P. Enright, and Nadia Ameer

EPCC, The King's Buildings, The University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland, U.K.
m.bull@epcc.ed.ac.uk

Abstract. With the current prevalence of multi-core processors in HPC architectures, mixed-mode programming, using both MPI and OpenMP in the same application, is becoming increasingly important. However, no low-level synthetic benchmarks exist to test the performance of this programming model. We have designed and implemented a set of microbenchmarks for mixed-mode programming, including both point-to-point and collective communication patterns. These microbenchmarks have been run on a number of current HPC architectures: the results show some interesting performance differences between the architectures and highlight some possible inefficiencies in the implementation of MPI on multi-core systems.

1 Introduction

With the advent of multi-core processors, and the associated diminishing rate of increase in processor clock speed, almost all current high performance computing systems now contain nodes which consist of shared memory multiprocessors. Large numbers of such nodes can be connected together with a high-bandwidth, low-latency network to form a scalable distributed memory system.

To program such systems, by far the most popular programming model is message-passing, using the MPI [4] library. MPI programs can execute on machines with shared memory nodes in a straightforward way by running one MPI process per core on each node. In this case, message-passing between processes on a node is normally implemented via shared memory, but this is not visible to the programmer. However, it is also possible to run fewer MPI processes than cores on each node, and make use of the additional cores by using a multi-threaded programming model. This is most frequently done using the OpenMP [5] API, but can also be accomplished via a lower-level thread library interface such as Posix threads. This programming style is termed *mixed-mode* or *hybrid* (we prefer the former term as the latter is somewhat overloaded in the HPC literature).

Several studies (for example [7],[10]) have shown that, in certain circumstances, mixed-mode programs can perform better than (or consume less memory than) the equivalent program using MPI only. Such advantages may outweigh the potential additional software complexity and possible loss of portability of

M.S. Müller et al. (Eds.): IWOMP 2009, LNCS 5568, pp. 118–131, 2009.
© Springer-Verlag Berlin Heidelberg 2009

the mixed-mode version. A number of commonly used HPC applications, such as CPMD [2] and ECMWF's Integrated Forecast System (IFS) [9] successfully exploit mixed-mode programming. With the principal performance gains in HPC architectures likely to come, in the near future at least, primarily from increasing the number of cores per chip, mixed-mode programming seems likely to assume a more important role, as it may allow applications to scale better on such systems than pure MPI.

Microbenchmarks (low-level synthetic benchmarks testing the performance of basic operations) exist for both MPI [3], [8] and OpenMP [1]. However, these microbenchmarks cannot on their own give sufficient information about the performance of mixed-mode programs, as there will, in general, be interactions between the MPI and OpenMP layers. The Sphinx benchmark suite from LLNL [11] contains a small number of OpenMP/MPI microbenchmarks, which measure the performance of mixed-mode barriers and reductions, and assess the ability to overlap threaded computation with MPI non-blocking communication.

To fill this gap, we have designed and implemented a suite of microbenchmarks for mixed-mode OpenMP/MPI programming. The utility of such a suite is demonstrated by the results presented in [6], which demonstrate how the available communication bandwidth between nodes can depend on the mix of MPI processes and OpenMP threads employed.

In Section 2, we describe the contents of the suite and the rationale for its construction. In Section 3, we present selected results from running the microbenchmarks on a number of current HPC architectures, and demonstrate the interesting features thus illuminated. Finally, Section 4 presents our conclusions and possibilities for future work.

2 Benchmark Design and Implementation

The basic design concept of the mixed-mode microbenchmarks is to provide mixed-mode analogues for (a subset of) the typical operations found in MPI microbenchmark suites, for both point-to-point and collective communications. There are two main considerations which have driven the design of the benchmark suite. Firstly, we wish to adequately capture the cost of the inter-thread communication and synchronisation which may occur in mixed-mode programs if not all threads participate in the inter-node (MPI) communication. To do this, we measure not only the cost of the MPI library calls themselves, but also the (possibly multi-threaded) writing of send buffers, and reading of receive buffers. The second consideration is that we wish to be able to directly and easily compare the performance of the same communication patterns when we hold the total number of cores constant, but vary the number of MPI processes and OpenMP threads (such that the product of these two values equals the number of cores). This is achieved by the appropriate choices of data buffer sizes and MPI message lengths.

The benchmarks are implemented in Fortran90. We may produce a C version in the future, but we expect that there would be little dependence on the base language, as most of the performance characteristics are dictated by the hardware and by the MPI and OpenMP libraries.

2.1 Point-to-Point Operations

In the mixed-mode microbenchmark suite we measure the performance of three point-to-point communication operations: PingPong, PingPing and HaloExchange. Each of these operations is implemented in each of three different ways:

1. **Master-only:** MPI communication takes place in the master thread, outside of parallel regions.
2. **Funnelled:** MPI communication takes place in the master thread, inside parallel regions.
3. **Multiple:** MPI communication takes place concurrently in all threads inside parallel regions.

To illustrate this, Figures 1–3 show pseudocode representations of these three forms of the PingPong benchmark. The PingPing benchmark differs from PingPong in that messages are exchanged in both directions between the two processes concurrently. For both the PingPong and PingPing benchmarks, the user can specify the two MPI ranks which participate: this is intended to permit the measurement of both intra-node and inter-node MPI communication, by specifying two MPI ranks which will execute either on the same, or on different nodes. The benchmark reports which of these was the case by comparing the results of `MPI_GET_PROCESSOR_NAME` on the two participating process. For the HaloExchange benchmark all MPI processes participate. The processes are arranged in a ring and each process exchanges messages with its two neighbouring processes. In the point-to-point benchmarks, the data sizes specified by the user correspond to the total number of 4-byte words sent between pairs of MPI processes.

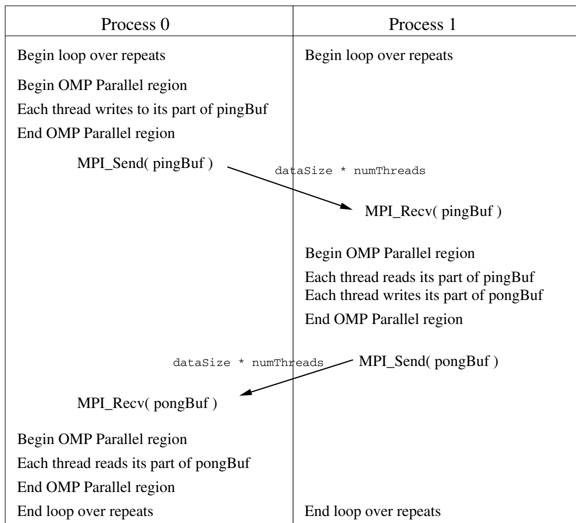

Fig. 1. Pseudocode for Master-only PingPong benchmark

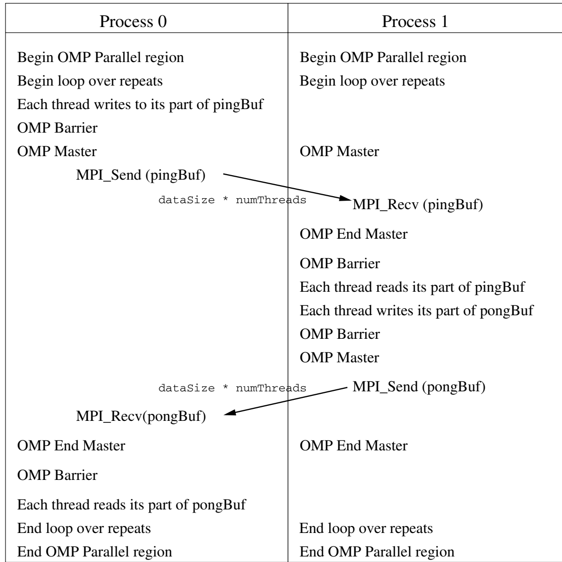

Fig. 2. Pseudocode for Funnelled PingPong benchmark

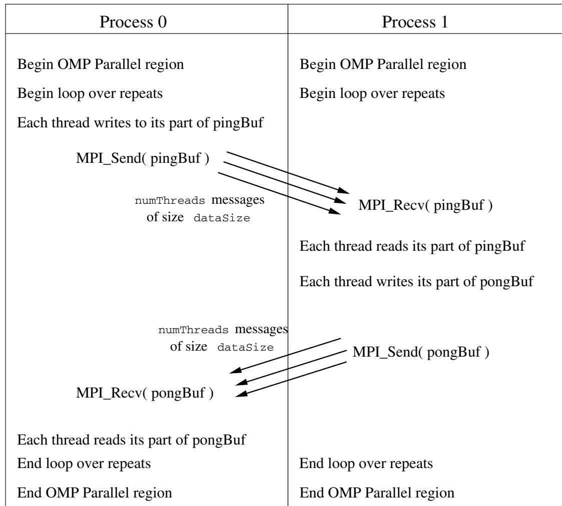

Fig. 3. Pseudocode for Multiple PingPong benchmark

2.2 Collective Operations

The microbenchmark suite contains measurements for mixed-mode analogues of the following operations: Barrier, Reduce, AllReduce, Gather, Scatter and

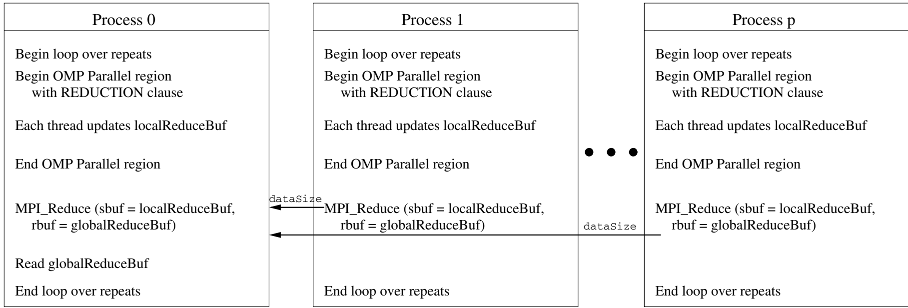

Fig. 4. Pseudocode for Reduce benchmark

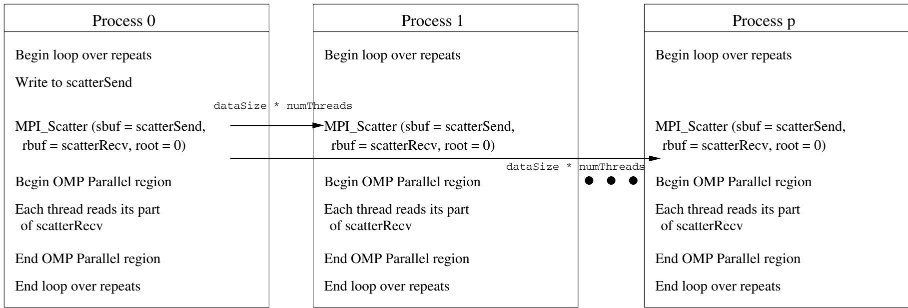

Fig. 5. Pseudocode for Scatter benchmark

AlltoAll. Figures 4 and 5 show pseudocode representations of the Reduce and Scatter benchmarks respectively.

The other collective benchmarks are constructed in an analogous fashion. The total amount of data involved is proportional to both the number of OpenMP threads and the number of MPI processes. This means that experiments can easily be conducted where the total amount of compute resource, and the product of the number of threads and the number of processes is fixed, but the number of processes and number of threads per process is varied. The benchmarks are constructed so that when this is done, the patterns and quantity of data movement are preserved. (Note that for the Barrier benchmark, no data is involved).

2.3 Benchmark Control

The user is able to set some control parameters for the benchmark suite:

- A list of the benchmarks to be run.
- The minimum and maximum data sizes to be run. The data size starts at the minimum size and is successively doubled until the maximum is reached.
- A target execution time. If the execution time for a given data size is less than this value, it is rejected, and the test is re-run with twice the number

of repetitions. If the execution time is more than twice the target time, it is accepted, and the initial number of repetitions for the subsequent data size is set to half its current value. This process is intended to keep the execution time for each test approximately constant, regardless of the benchmark or the data size used. Before each timed test, two repetitions are run as a warm-up.
– The MPI process IDs to be used for the PingPong and PingPing benchmarks. Negative values are permitted: in this case the value is added to the total number of processes to give a valid ID.

At present, the number of MPI processes and OpenMP threads are controlled by the way the benchmark is executed (i.e. by the `mpirun` command or equivalent, and the value of the `OMP_NUM_THREADS` environment variable) and they are fixed for that run. We considered trying to vary the number of process and threads within a run, but the complexity of the programming and the possible lack of control over idle threads mean we have not yet done so.

2.4 Other Issues

Each benchmark has a validation test, which is run on the warm-up repetitions. For each benchmark and data size a Pass or Fail is reported.

The benchmark reports the value returned by `MPI_INIT_THREAD`, and issues a warning if the level of support is not adequate for the benchmark. The Multiple versions of point-to-point benchmarks require `MPI_THREAD_MULTIPLE`, while all other benchmarks require `MPI_THREAD_FUNNELED`. We have found that the value returned is a poor indicator of whether the validation test will succeed. We have encountered one implementation of MPI which returns `MPI_THREAD_SINGLE` but runs the benchmarks requiring `MPI_THREAD_FUNNELED` successfully, and another implementation which returns `MPI_THREAD_MULTIPLE`, but fails to run the benchmarks requiring this value.

3 Benchmark Results

3.1 Hardware

We have run the benchmark suite on four different platforms:

– **IBM eServer 575 Power5 cluster**. Each node contains 8 1.6GHz dual-core processors and 32GB of memory, and the nodes are connected with IBM's High Performance Switch (HPS) with a total of four links from each node to the network. The system was running Version 10.1 of the IBM xlf90 Fortran compiler and Version 4.3 of IBM Parallel Operating Environment. Our experiments used 4 nodes (64 cores).
– **IBM eServer 575 Power6 cluster**. Each node contains 16 4.7 GHz dual-core processors and 128 or 256GB of memory. The nodes are connected through an Infiniband network with four links from each node to the network.

The system was running Version 12.1 of the IBM xlf90 Fortran compiler and Version 4.3 of IBM Parallel Operating Environment. Our experiments used 4 nodes (128 cores).

– **IBM BlueGene/P**. Each node has four 850MHz Power450 cores and 2GB of memory. There are three networks connecting the compute nodes of the BlueGene/P, a 3D torus network and two tree networks (one used for collective communication, the other for barrier synchronisation). The system was running Version 11.1 of the IBM xlf90 Fortran compiler and BlueGene Driver Version 1.0 Release 3.0. Our experiments used 16 nodes (64 cores).
– **Cray XT4**. Each node contains a quad-core 2.3 GHz AMD Opteron processor and 8 or 16GB of main memory. The network is a Cray SeaStar 3D torus. The system was running Version 7.2.4 of the PGI pgf90 compiler and Version 3.0.2 of the Cray Message Passing Toolkit. Our experiments used 16 nodes (64 cores).

In all cases we fully populated the nodes, so the product of the number of MPI process per node and the number of OpenMP threads per process always equals the number of cores per node.

3.2 Results

We do not have space here to show the results of all the benchmarks on all the platforms, so we have selected some of the more interesting results for presentation.

Figures 6 and 7 show the results of running the Master-only version of the PingPong benchmark on the IBM Power 5 cluster and BlueGene/P system respectively. The execution times are normalised to the execution time with

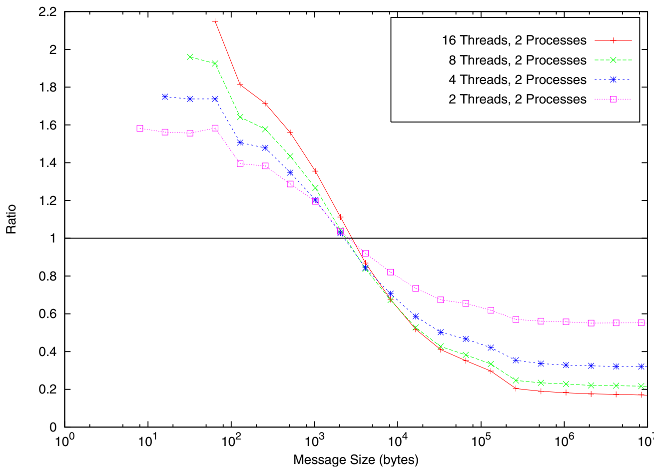

Fig. 6. IBM Power 5: Ratio of time in Master-only PingPong benchmark to one thread per process

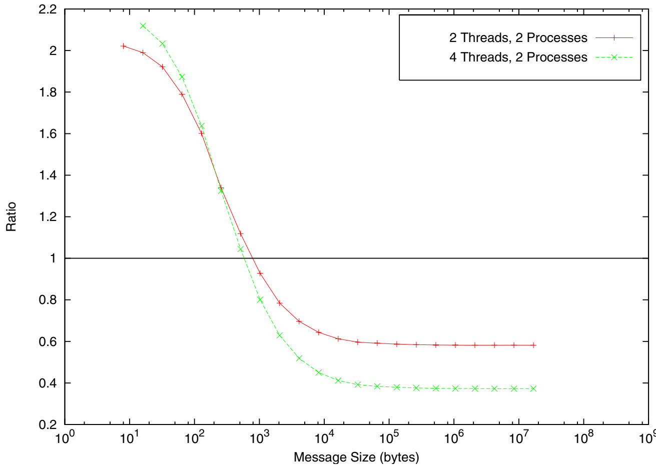

Fig. 7. IBM BlueGene/P: Ratio of time in Master-only PingPong benchmark to one thread per process

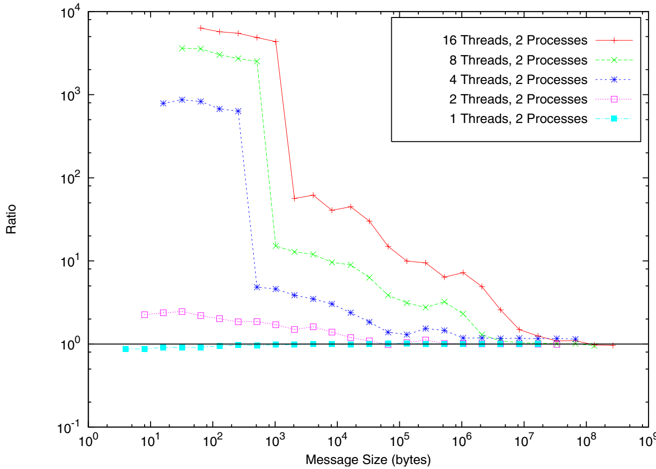

Fig. 8. IBM Power5: Ratio of Multiple to Master-only PingPong execution times

one OpenMP thread per MPI process. The MPI ranks participating in the benchmark are chosen to lie on different nodes. On both systems, for small data sizes, the execution time is least for one thread per MPI process, and increases with the number of threads, whereas for large data sizes, the execution time is greatest for one thread per MPI process, and decreases with the number of threads. The crossover between these regimes occurs between 10^3 and 10^4 bytes. Recall that the send and receive buffers are being written/read by multiple threads. For small data sizes, the overhead of parallelisation is not

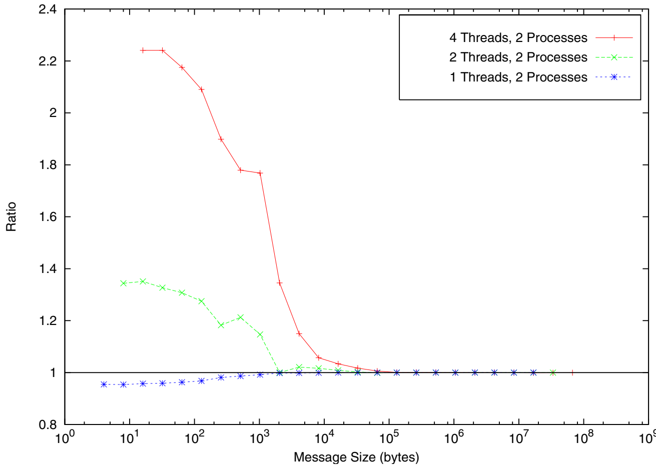

Fig. 9. IBM BlueGene/P: Ratio of Multiple to Master-only PingPong execution times

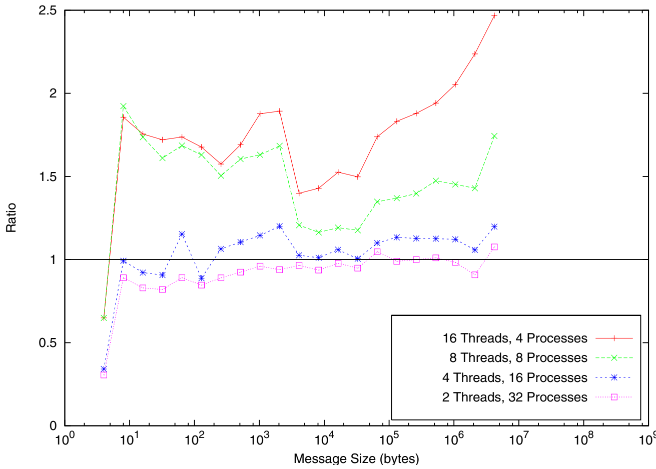

Fig. 10. IBM Power 5: Ratio of execution time for Reduce benchmark to one thread per process

worthwhile, but above the crossover, significant benefit is gained from having multiple threads employed. The other hardware platforms display similar behaviour (not shown here).

Figures 8 and 9 show the results of running the Multiple version of the PingPong benchmark on the IBM Power 5 cluster and BlueGene/P system respectively. In this case the execution times are normalised by the time for the Master-only PingPong benchmark running on the same number of processes and threads. For the Power5 system, we observe very poor performance for the

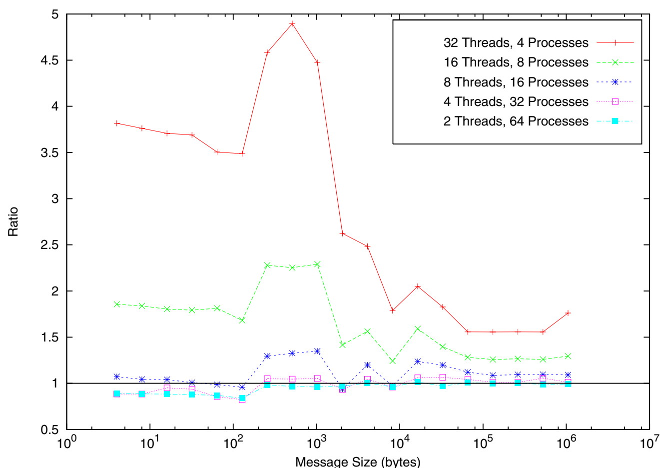

Fig. 11. IBM Power 6: Ratio of execution time for Reduce benchmark to one thread per process

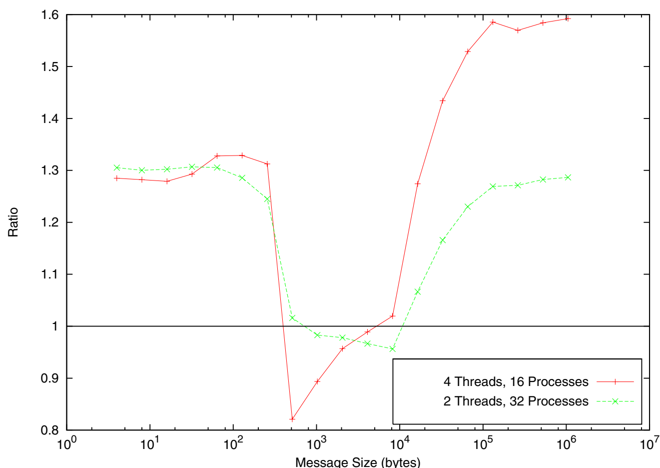

Fig. 12. IBM BlueGene/P: Ratio of execution time for Reduce benchmark to one thread per process

Multiple version on small data sizes: in some cases it is over 3 orders of magnitude slower (note the log scale on the vertical axis in Figure 8. The Power 6 system displays similar behaviour to the Power 5: contention for locks inside the MPI library is a possible cause of this. In contrast, the Multiple version on the BlueGene/P system is a little over two times slower using four threads per process than using one. On neither system is there any benefit gained from calling MPI from multiple threads for large data sizes. This suggests that a single large message is able to utilise all the off-node bandwidth.

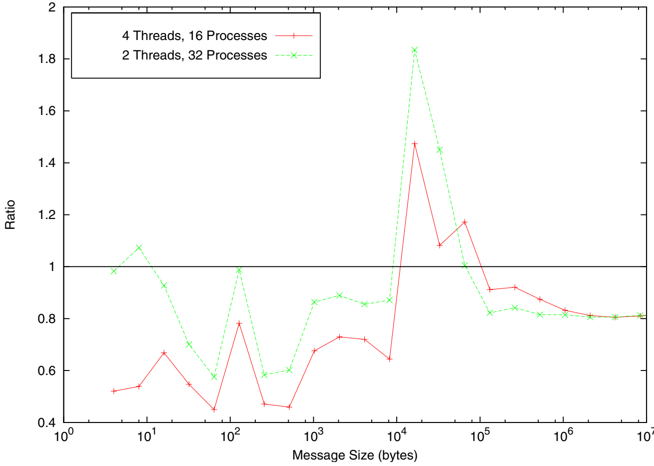

Fig. 13. Cray XT4: Ratio of execution time for Reduce benchmark to one thread per process

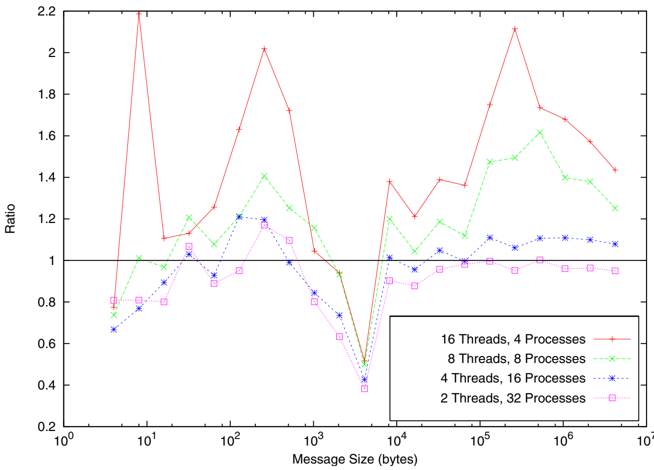

Fig. 14. IBM Power 5: Ratio of execution time for AlltoAll benchmark to one thread per process

Figures 10 to 13 show the results of running the Reduce benchmark on all four platforms. The execution times are normalised to the execution time with one OpenMP thread per MPI process.

On the IBM Power 5 and Power 6 systems, we observe that the mixed-mode version of Reduce is generally slower than the pure MPI (one thread per process), though there are some modest gains to be had by using two threads per process for small data sizes. On the BlueGene/P system, the mixed-mode version is

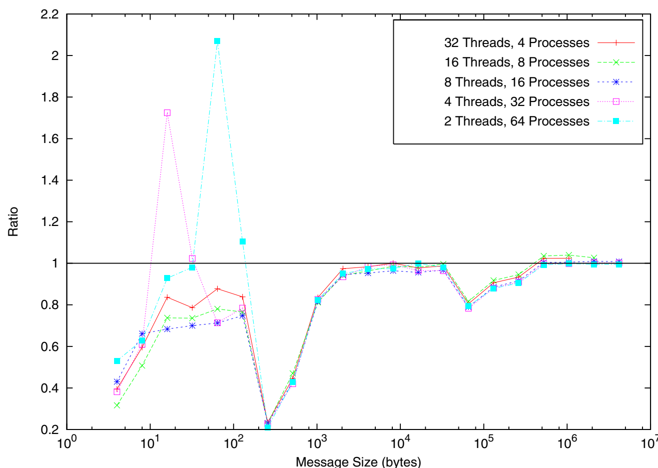

Fig. 15. IBM Power 6: Ratio of execution time for AlltoAll benchmark to one thread per process

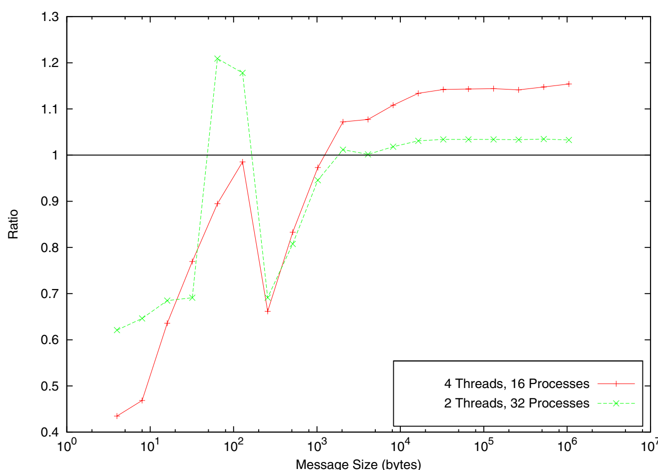

Fig. 16. IBM BlueGene/P: Ratio of execution time for AlltoAll benchmark to one thread per process

also slower, except for a window of data sizes between 10^3 and 10^4 bytes. This suggests that on these platforms, the MPI Reduce is well optimised for shared memory nodes. It is also possible that the implementation of OpenMP array reductions is not very efficiently implemented. On the Cray XT4, however, the mixed mode version is generally faster, except for data sizes between 10^4 and 10^5 bytes. This system is known to suffer from contention between cores on the same node for access to the network: having fewer, larger, messages entering the network seems to be beneficial.

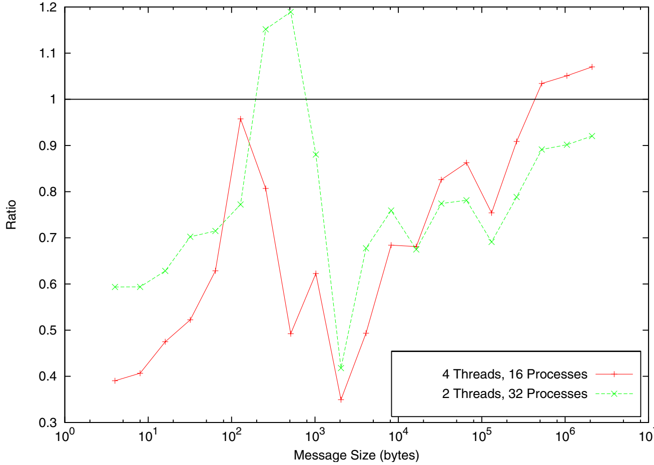

Fig. 17. Cray XT4: Ratio of execution time for AlltoAll benchmark to one thread per process

Figures 14 to 17 show the results of running the AlltoAll benchmark on all four platforms. The execution times are normalised to the execution time with one OpenMP thread per MPI process. The four systems show different behaviours for this benchmark. On the IBM Power 5, the mixed mode version is significantly faster for data sizes in the range 10^3 to 10^4 bytes, and the optimal number of threads per process is usually two. For the IBM Power 6, having multiple threads per process is beneficial on small data sizes, but increasing the number of threads per process beyond two makes little difference. On the BlueGene/P, mixed-mode is worthwhile for small data sizes, but not large ones, and on the Cray XT4 it is worthwhile for almost all data sizes and is up to three times faster in some cases.

4 Conclusions and Future Work

We have described the design and implementation of a set of microbenchmarks for mixed-mode OpenMP/MPI programming. These cover both point-to-pont and collective communication patterns. We have run these benchmarks on four current HPC architectures: the results show some interesting performance differences between the architectures and highlight some possible inefficiencies in the implementation of MPI on these systems.

In the future, we intend to run the benchmarks on other systems, for example on Intel- and Opteron-based clusters (where there may be multiple combinations of MPI library and OpenMP compiler available) and on vector systems such the NEC SX-9 and the Cray X2. We can also consider additions to the benchmark suite: for example multi-PingPong (where every core on a node communicates with a corresponding core on another node).

References

1. Bull, J.M., O'Neill, D.: A Microbenchmark Suite for OpenMP 2.0. In: Proceedings of the Third European Workshop on OpenMP (EWOMP 2001), Barcelona, Spain (September 2001)
2. Hutter, J., Curioni, A.: Dual-level Parallelism for Ab Initio Molecular Dynamics: Reaching Teraflop Performance with the CPMD Code. Parallel Computing 31(1), 1–17 (2005)
3. Intel. MPI Benchmarks, `http://www.intel.com/cd/software/products/asmo-na/eng/cluster/mpi/219847.htm`
4. MPI Forum, MPI: A Message-Passing Interface Standard Version 2.1 (2008)
5. OpenMP ARB, OpenMP Application Programming Interface Version 3.0 (2008)
6. Rabenseifner, R.: Hybrid Parallel Programming on HPC Platforms. In: Proceedings of the Fifth European Workshop on OpenMP, EWOMP 2003, Aachen, Germany, September 22-26, pp. 185–194 (2003)
7. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2009 (2009) (to appear)
8. Reussner, R., Sanders, P., Traeff, J.L.: SKaMPi: a Comprehensive Benchmark for Public Benchmarking of MPI. Scientific Programming 10(1), 55–65 (2002)
9. Salmond, D., Saarinen, S.: Early Experiences with the New IBM p690+ at ECMWF. In: Proceedings of the Eleventh ECMWF Workshop Reading, UK, pp. 1–12. World Scientific, Singapore (2005)
10. Smith, L., Bull, M.: Development of Mixed Mode MPI/OpenMP Applications. Scientific Programming 9(2-3), 83–98 (2001)
11. The Sphinx Parallel Microbenchmark Suite, `http://www.llnl.gov/CASC/sphinx/sphinx.html`