Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures

Damián A. Mallón[1], Guillermo L. Taboada[2], Carlos Teijeiro[2], Juan Touriño[2], Basilio B. Fraguela[2], Andrés Gómez[1], Ramón Doallo[2], and J. Carlos Mouriño[1]

[1] Galicia Supercomputing Center (CESGA), Santiago de Compostela, Spain
[2] Computer Architecture Group, University of A Coruña, A Coruña, Spain
{dalvarez,agomez,jmourino}@cesga.es
{taboada,cteijeiro,juan,basilio,doallo}@udc.es

Abstract. The current trend to multicore architectures underscores the need of parallelism. While new languages and alternatives for supporting more efficiently these systems are proposed, MPI faces this new challenge. Therefore, up-to-date performance evaluations of current options for programming multicore systems are needed. This paper evaluates MPI performance against Unified Parallel C (UPC) and OpenMP on multicore architectures. From the analysis of the results, it can be concluded that MPI is generally the best choice on multicore systems with both shared and hybrid shared/distributed memory, as it takes the highest advantage of data locality, the key factor for performance in these systems. Regarding UPC, although it exploits efficiently the data layout in memory, it suffers from remote shared memory accesses, whereas OpenMP usually lacks efficient data locality support and is restricted to shared memory systems, which limits its scalability.

Key words: MPI, UPC, OpenMP, Multicore Architectures, Performance Evaluation, NAS Parallel Benchmarks (NPB)

1 Introduction

Currently, multicore clusters are the most popular option for the deployment of High Performance Computing (HPC) infrastructures, due to their scalability and performance/cost ratio. These systems are usually programmed using MPI [1] on distributed memory, OpenMP [2] on shared memory, and MPI+OpenMP on hybrid shared/distributed memory architectures. Additionally, the Partitioned Global Address Space (PGAS) languages, such as Unified Parallel C (UPC) [3], are an emerging alternative that allows shared memory-like programming on distributed memory systems, taking advantage of the data locality, being of special interest for hybrid architectures.

In the past other alternatives, such as High Performance Fortran, aimed to replace MPI as the primary choice for HPC programming. These alternatives have been dismissed and MPI is still the preferred platform for HPC developers due to its higher performance and portability. In order to evaluate the current

validity of this assessment, this work presents an up-to-date comparative performance evaluation of MPI, UPC and OpenMP on two multicore scenarios: a cluster composed of 16-core nodes interconnected via InfiniBand, and a 128-core shared memory system. This evaluation uses the standard parallel benchmarking suite, the NAS Parallel Benchmarks (NPB) [4]. A matrix multiplication kernel and the Sobel edge detection kernel, have also been used to assess the scalability of the evaluated parallel programming solutions.

This paper is structured as follows: Section 2 presents current trends in multicore parallel programming (MPI, OpenMP and PGAS). Section 3 introduces the benchmarks used for the comparative evaluation and related work. Benchmarking results obtained from two multicore systems are shown and analyzed in Section 4. Finally, Section 5 presents the main conclusions of this evaluation.

2 Parallel Programming for Multicore Architectures

This section presents three of the main options for parallel programming multicore architectures. These approaches are the message-passing paradigm, shared memory programming, and the PGAS programming model.

The message-passing is the most widely used parallel programming model as it is portable, scalable and provides good performance for a wide variety of computing platforms and codes. It is the preferred choice for parallel programming distributed memory systems, such as multicore clusters. Therefore, as the programmer has to manage explicitly data placement through point-to-point or collective operations, the programming of these architectures is difficult. MPI is the standard interface for message-passing libraries and there is a wide range of MPI implementations, both from HPC vendors and the free software community, optimized for high-speed networks, such as InfiniBand or Myrinet. MPI, although is oriented towards distributed memory environments, faces the raise of the number of cores per system with the development of efficient shared memory transfers and providing thread safety support.

The shared memory programming model allows a simpler programming of parallel applications, as here the control of the data location is not required. OpenMP is the most widely used solution for shared memory programming, as it allows an easy development of parallel applications through compiler directives. Moreover, it is becoming more important as the number of cores per system increases. However, as this model is limited to shared memory architectures, the performance is bound to the computational power of a single system. To avoid this limitation, hybrid systems, with both shared/distributed memory, such as multicore clusters, can be programmed using MPI+OpenMP. However, this hybrid model can make the parallelization more difficult and the performance gains could not compensate for the effort [5, 6].

The PGAS programming model combines the main features of the message-passing and the shared memory programming models. In PGAS languages, each thread has its own private memory space, as well as an associated shared memory region of the global address space that can be accessed by other threads,

although at a higher cost than a local access. Thus, PGAS languages allow shared memory-like programming on distributed memory systems. Moreover, as in message-passing, PGAS languages allow the exploitation of data locality as the shared memory is partitioned among the threads in regions, each one with affinity to the corresponding thread. This feature of the PGAS languages could makes them important in the near future, as modern multicore architectures became NUMA architectures with the inclusion of the memory controller in the CPU. UPC is the PGAS extension to the ISO C language, and has been used in this evaluation due to its important support by academia and industry, with compilers available from Univ. of Berkeley, Michigan Tech. Univ., Intrepid/GCC, IBM, Cray, and HP. UPC provides PGAS features to C, allowing a more productive code development [7]. However, as an emerging programming model, performance analysis are needed [8–10].

3 Parallel Benchmarks on Multicore Architectures

A comparative performance evaluation needs standard, unbiased benchmarks with implementations for the evaluated options. As UPC is an emerging option, the number of available benchmarks for UPC benchmarking is limited. The main source of UPC benchmarks is the Berkeley UPC distribution [11], which includes the codes selected for our evaluation: the UPC version of the NPB and the Sobel edge detection kernel coded in MPI, UPC and OpenMP. Additionally, we have implemented a blocking algorithm for matrix multiplication.

3.1 NAS Parallel Benchmarks Description

The NPB consist of a set of kernels and pseudo-applications, taken primarily from Computational Fluid Dynamics (CFD) applications. These benchmarks reflect different kinds of computation and communication patterns that are important across a wide range of applications, which makes them the de facto standard in parallel performance benchmarking. There are NPB implementations available for a wide range of parallel programming languages and libraries, such as MPI (from now on NPB-MPI), UPC (from now on NPB-UPC), OpenMP (from now on NPB-OMP), a hybrid MPI+OpenMP implementation (not used in this comparative evaluation as it implements benchmarks not available in NPB-UPC), HPF and Message-Passing Java [12], among others.

The NPB evaluated are: CG, EP, FT, IS and MG. NPB-MPI and NPB-OMP are implemented using Fortran, except for IS which is programmed in C. The fact that the NPB are programmed in Fortran has been considered as cause of a poorer performance of NPB-UPC [9], due to better backend compiler optimizations for Fortran than for C. The CG kernel is an iterative solver that tests regular communications in sparse matrix-vector multiplications. The EP kernel is an embarrassingly parallel code that assesses the floating point performance, without significant communication. The FT kernel performs series of 1-D FFTs on a 3-D mesh that tests aggregated communication performance. The IS kernel

is a large integer sort that evaluates both integer computation performance and the aggregated communication throughput. MG is a simplified multigrid kernel that performs both short and long distance communications. Moreover, each kernel has several workloads to scale from small systems to supercomputers.

Most of the NPB-UPC kernels have been manually optimized through techniques that mature UPC compilers should handle in the future: privatization, which casts local shared accesses to private memory accesses, avoiding the translation from global shared address to actual address in local memory, and prefetching, which copies non-local shared memory blocks into private memory. Although OpenMP 3.0 adds data locality support, the NPB-OMP codes do not take advantage of these features.

3.2 Related Work

Up to now only three works have analyzed the performance of MPI against UPC in computational kernels [9, 10, 13]. The first work [9] has compared the performance of NPB-MPI with NPB-UPC on a 16 processor Compaq AlphaServer SC cluster, using the class B workload. The second work [10] has used two SGI Origin NUMA machines, each one with 32 processors, using the class A workload for 3 NPB. The Sobel edge detector kernel has also been used in both works. The third work [13] compares MPI with UPC very briefly using 2 NPB kernels and class B, in a Cray X1 machine, with up to 64 processors. Other works have tackled the kernel and micro-benchmarking of low-level parameters on multicore architectures, especially memory hierarchy performance [14, 15], but they are limited to 8-core systems, and do not evaluate distributed memory programming solutions such as MPI or UPC.

This paper improves previous works by: (1) using a higher number of processor cores (128) with different memory configurations (shared and hybrid shared/distributed memory); (2) including OpenMP performance in the shared memory configuration; (3) using a larger workload, more representative of large scale applications (class C); and (4) providing an up-to-date performance snapshot of MPI performance versus UPC and OpenMP on multicore architectures.

Finally, this paper addresses performance issues that are present in a cluster with 16-core nodes, and on a 128-core system, but not in 4- and 8-core systems. Thus, it is possible to foresee the main drawbacks that may affect performance on the multicore architectures that will be commonly deployed in the next years.

4 Performance Evaluation

The testbed used in this work is the Finis Terrae supercomputer [16], composed of 142 HP Integrity rx7640 nodes, each one with 8 Montvale Itanium 2 dual-core processors (16 cores per node) at 1.6 GHz and 128 GB of memory, interconnected via InfiniBand. The InfiniBand HCA is a dual 4X IB port (16 Gbps of theoretical effective bandwidth). For the evaluation of the hybrid shared/distributed memory scenario, 8 nodes have been used (up to 128 cores). The number of cores

used per node in the performance evaluation is $\lceil n/8 \rceil$, being n the total number of cores used in the execution, with consecutive distribution. An HP Integrity Superdome system with 64 Montvale Itanium 2 dual-core processors (total 128 cores) at 1.6 GHz and 1 TB of memory has also been used for the shared memory evaluation. The nodes were used without other users processes running, and the process affinity was handled by the operating system scheduler.

The OS is SUSE Linux Enterprise Server 10, and the MPI library is the recommended by the hardware vendor, HP MPI 2.2.5.1 using InfiniBand Verbs (IBV) for internode communication, and shared memory transfers (HP MPI SHM driver) for intranode communication. The UPC compiler is Berkeley UPC 2.8, which uses the IBV driver for distributed memory communication, and pthreads within a node for shared memory transfers. The backend for both and OpenMP compiler is the Intel 11.0.069 (icc/ifort, with -O3 flag).

4.1 Matrix Multiplication and Sobel Kernels Performance

Figure 1 shows the results for the matrix multiplication and the Sobel kernel. The matrix multiplication uses matrices of 2400×2400 doubles, with a blocking factor of 100 elements, and the experimental results include the data distribution overhead. The Sobel kernel uses a 65536×65536 unsigned char matrix and does not take into account the data distribution overhead. The graphs show the speedups on the hybrid scenario (MPI and UPC) and in the shared memory system (UPC and OpenMP).

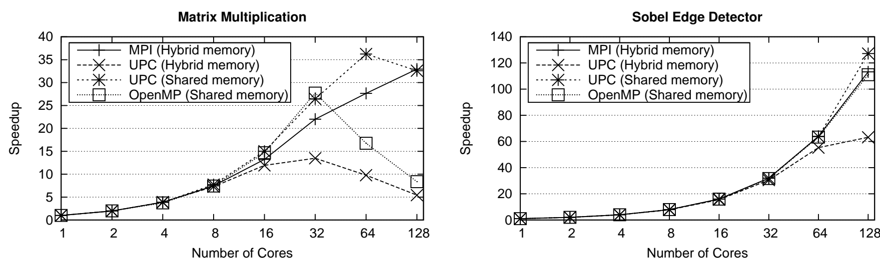

Fig. 1. Performance of matrix multiplication and Sobel kernels

The three options obtain similar speedups on up to 8 cores. MPI can take advantage of the use of up to 128 cores, whereas UPC (hybrid memory) presents poor scalability, as it lacks efficient collectives [17]. In shared memory, UPC and OpenMP show similar speedups up to 32 cores. However, on 128 cores UPC achieves the best performance, whereas OpenMP suffers an important performance penalty due to the sharing of one of the matrices, whereas in UPC this matrix is copied to private space, thus avoiding shared memory access contention. MPI shows better performance than OpenMP for this reason.

In the Sobel kernel results, because the data distribution overhead is not considered, the speedups are almost linear, except for UPC on the hybrid scenario, where several remote shared memory accesses limit seriously its scalability. Nevertheless, UPC on shared memory achieves the highest speedups as these remote accesses are intraprocess accesses (UPC uses pthreads in this scenario).

4.2 Performance of NPB Kernels on Hybrid Memory

Figure 2 shows NPB-MPI and NPB-UPC performance on the hybrid configuration (using both InfiniBand and shared memory communication). The left graphs show the kernels performance in MOPS (Million Operations Per Second), whereas the right graphs present their associated speedups.

Regarding the CG kernel, MPI performs slightly worse than UPC using up to 32 cores, due to the kernel implementation, whereas on 64, and especially on 128 cores MPI outperforms UPC. Although UPC uses pthreads within a node, its communication operations, most of them point-to-point transfers with a regular communication pattern, are less scalable than MPI primitives.

EP is an embarrassingly parallel kernel, and therefore shows almost linear scalability for both MPI and UPC. The results in MOPS are approximately 6 times lower for UPC than for MPI due to the poorer UPC compiler optimizations. EP is the only NPB-UPC kernel that has not been optimized through prefetching and/or privatization, and the workload distribution is done through a `upc_forall` function, preventing more aggressive optimizations.

The performance of FT depends on the efficiency of the exchange collective operations. Although the UPC implementation is optimized through privatization, it presents significantly lower performance than MPI. The UPC results, although significantly lower than MPI in terms of MOPS, it shows higher speedups than MPI. This is a communication-intensive code that benefits from UPC intranode shared memory communication, which is maximized on 64 and 128 cores.

IS is a quite communication-intensive code. Thus, both MPI and UPC obtain low speedups for this kernel (less than 25 on 128 cores). Although UPC IS has been optimized using privatization, the lower performance of its communications limits its scalability, which is slightly lower than MPI speedups.

Regarding MG, MPI outperforms UPC in terms of MOPS, whereas UPC shows higher speedup. The reason is the poor performance of UPC MG on 1 core, which allows it to obtain almost linear speedups on up to 16 cores.

4.3 Performance of NPB Kernels on Shared Memory

Figure 3 shows NPB performance on the Superdome system. As in the hybrid memory figures, the left graphs show the kernels performance in MOPS and the right graphs show the speedups. MPI requires copying data on shared memory, and therefore could be considered less efficient than the direct access to shared memory of UPC and OpenMP. The following results do not support this hypothesis.

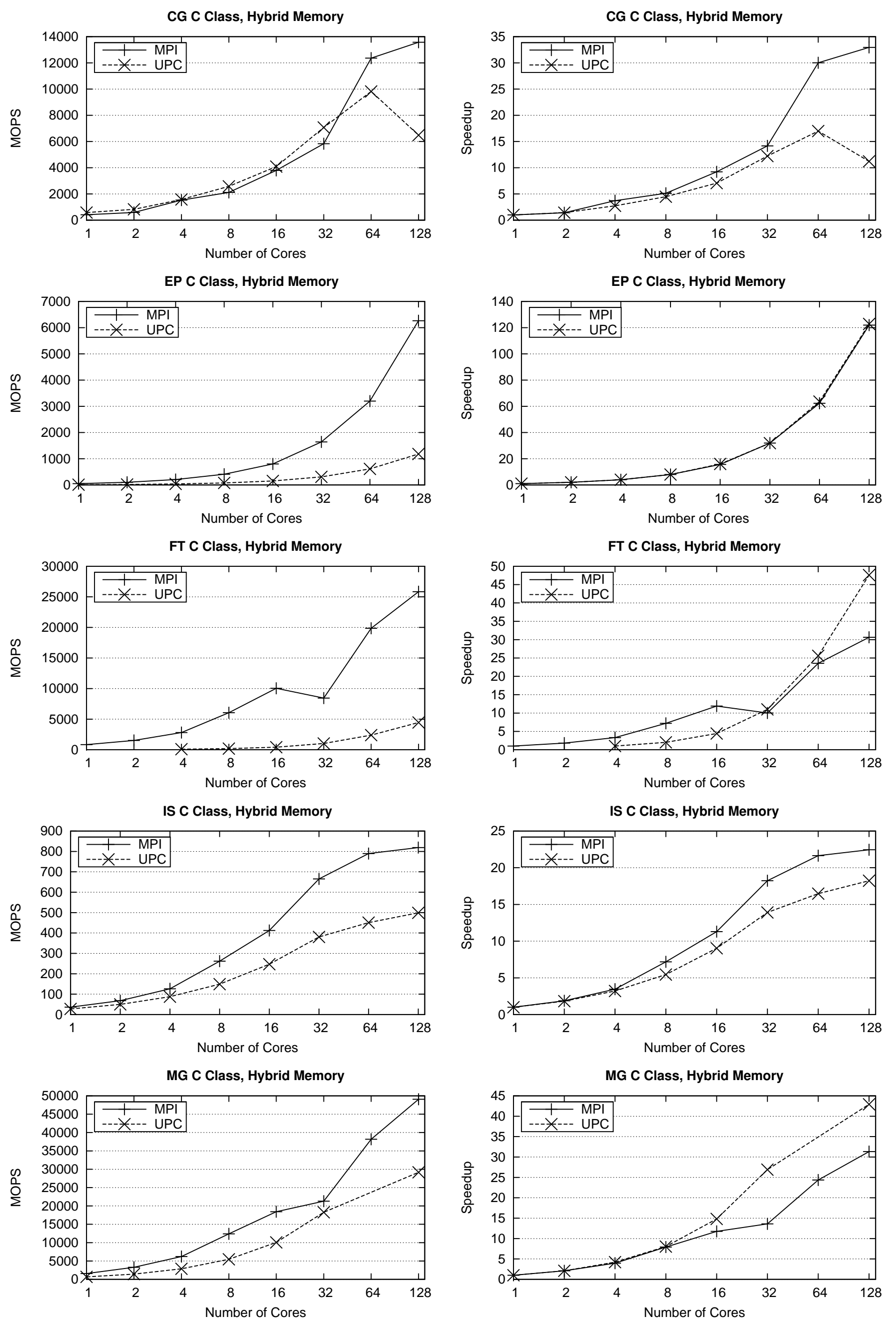

Fig. 2. Performance of NPB kernels on hybrid shared/distributed memory

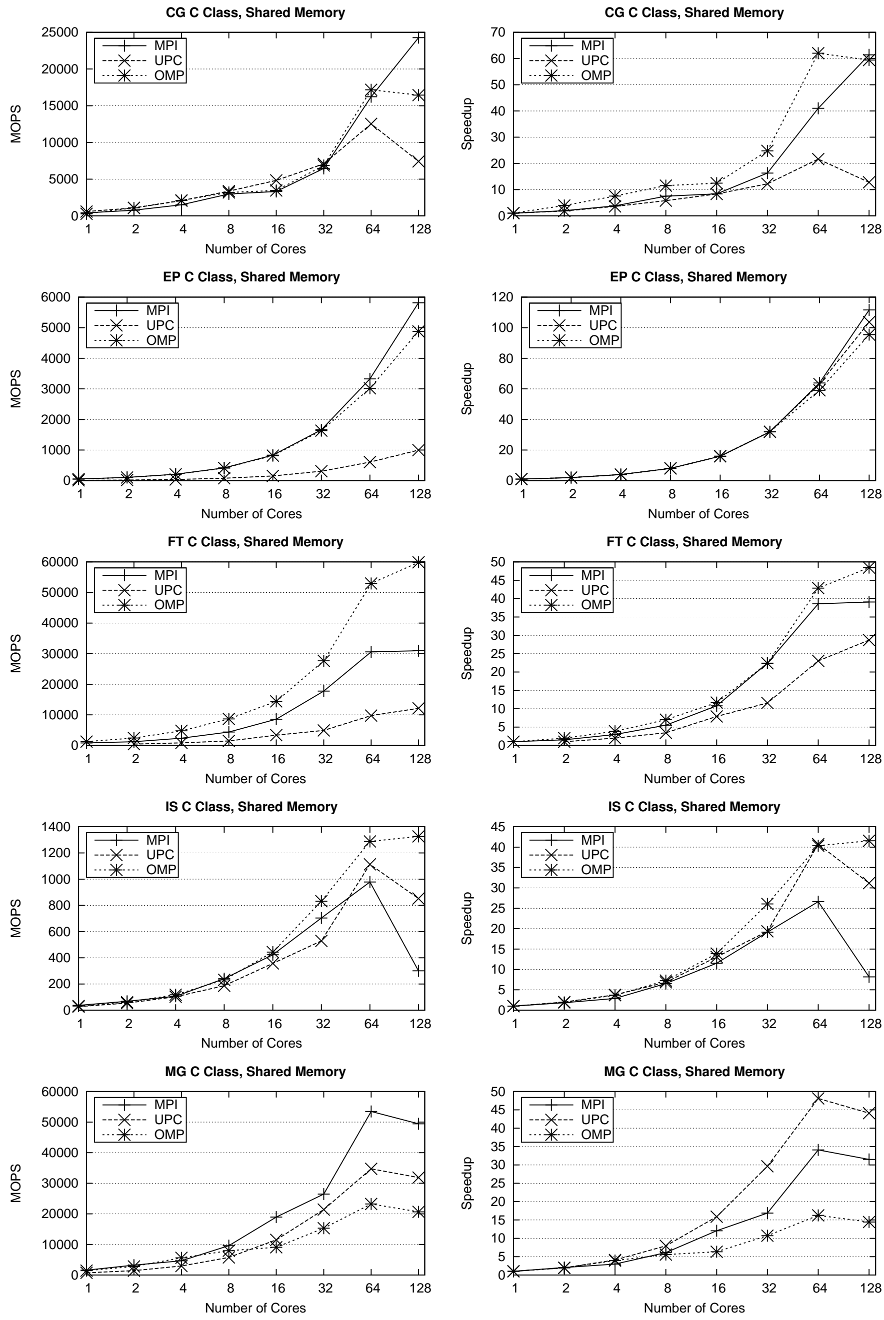

Fig. 3. Performance of NPB kernels on shared memory

Regarding CG, all options show similar performance on up to 32 cores. However, for 64 and 128 cores UPC scalability is poor, whereas MPI achieves the best MOPS results. The poor performance of OpenMP on 1 core leads OpenMP to present the highest speedups on up to 64 cores, being outperformed by MPI on 128 cores, due to the lack of data locality support of NPB-OMP.

As EP is an embarrassingly parallel code, the scalability shown is almost linear for MPI, UPC, and OpenMP, although MPI obtains slightly higher speedups, whereas OpenMP presents the lowest scalability. These results are explained by the efficiency in data locality exploitation of these three options. In terms of MOPS, UPC shows quite low performance, as discussed in subsection 4.2.

As FT is a communication-intensive code, its scalability depends on the performance of the communication operations. Therefore, OpenMP and MPI achieve high speedups, whereas UPC suffers from a less scalable exchange operation. The code structure of the OpenMP implementation allows more efficient optimizations and higher performance. Due to its good scalability OpenMP doubles MPI performance (in terms of MOPS) on 128 cores. UPC obtains the poorest performance.

IS is a communication-intensive code that shows similar performance for MPI, UPC and OpenMP on up to 32 cores, both in terms of MOPS and speedups, as the results on 1 core are quite similar among them. This fact can be partly explained by the fact that the IS kernels use the same backend compiler (icc). Regarding 64 and 128 cores results, OpenMP obtains the best performance and MPI the lowest, as the communications are the performance bottleneck of this kernel.

Regarding MG, MPI achieves better performance in terms of MOPS than UPC and OpenMP, whereas UPC obtains the highest speedups, due to the poor performance of this kernel on 1 core. OpenMP shows the lowest results, both in terms of MOPS and speedups.

5 Conclusions

This paper has presented an up-to-date performance evaluation of two well-established parallel programming models (MPI and OpenMP) and one emerging alternative (PGAS UPC) on two multicore scenarios. The analysis of the results obtained in the hybrid setup shows that MPI is the best performer thanks to its efficient handling of the data locality. However, UPC speedups are better for some benchmarks. Moreover, in some benchmarks the performance in the hybrid setup with 128 cores is not as high as expected, showing that for some workloads the network contention using a high number of cores may be a problem. This will be a bigger issue as the number of cores per node increases in the next years, where a higher network scalability will be required in order to confront this challenge.

Both MPI and UPC obtain better speedups in shared memory than in the hybrid setup up to 64 cores. However, for 128 cores all the options suffer from

remote memory accesses and poor bidirectional traffic performance in the cell controller.

MPI usually achieves good performance on shared memory, although UPC and OpenMP outperform it in some cases. OpenMP speedups are generally higher than those of MPI due to its direct shared memory access, which avoids memory copies as in MPI. UPC, despite its direct shared memory access and data locality support, suffers from its compiler technology and performs worse than the other two options. However, due to its expressiveness and ease of programming, is an alternative that has to be taken into account for productive development of parallel applications.

Acknowledgements. This work was funded by Hewlett-Packard and partially supported by the Spanish Government under Project TIN2007-67537-C03-02. We gratefully thank Jim Bovay and Brian Wibecan at HP for their valuable support, and CESGA for providing access to the Finis Terrae supercomputer.

References

1. "MPI Forum." `http://www.mpi-forum.org` [Last visited: June 2009].
2. "OpenMP." `http://openmp.org` [Last visited: June 2009].
3. "Unified Parallel C." `http://upc.gwu.edu` [Last visited: June 2009].
4. "NAS Parallel Benchmarks." `http://www.nas.nasa.gov/Resources/Software/npb.html` [Last visited: June 2009].
5. R. Rabenseifner, G. Hager, and G. Jost, "Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes", in *Proc. of the 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09)*, Weimar (Germany), pages 427–436, 2009.
6. R. Rabenseifner, G. Hager, G. Jost, and R. Keller, "Hybrid MPI and OpenMP Parallel Programming", in *Proc. of the 13th European PVM/MPI Users Group Meeting (EuroPVM/MPI'06)*, page 11, 2006.
7. T. A. El-Ghazawi, F. Cantonnet, Y. Yao, S. Annareddy, and A. S. Mohamed, "Productivity Analysis of the UPC Language", in *Proc. 3rd Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO'04)*, Santa Fe (NM), pages 1–7, 2004.
8. T. A. El-Ghazawi and C. Sébastien, "UPC Benchmarking Issues", in *Proc. 30th IEEE Intl. Conf. on Parallel Processing (ICPP'01)*, Valencia (Spain), pages 365–372, 2001.
9. T. A. El-Ghazawi and F. Cantonnet, "UPC Performance and Potential: a NPB Experimental Study", in *Proc. of the 15th ACM/IEEE Conf. on Supercomputing (SC'02)*, Baltimore (MD), pages 1–26, 2002.
10. F. Cantonnet, Y. Yao, S. Annareddy, A. Mohamed, and T. A. El-Ghazawi, "Performance Monitoring and Evaluation of a UPC Implementation on a NUMA Architecture", in *Proc. of the 2nd Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO'03)*, Nice (France), 274 (8 pages), 2003.
11. "Berkeley UPC." `http://upc.lbl.gov/` [Last visited: June 2009].

12. D. A. Mallón, G. L. Taboada, J. Touriño, and R. Doallo, "NPB-MPJ: NAS Parallel Benchmarks Implementation for Message Passing in Java", in *Proc. of the 17th Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing (PDP'09)*, Weimar (Germany), pages 181–190, 2009.
13. T. A. El-Ghazawi, F. Cantonnet, Y. Yao, and J. Vetter, "Evaluation of UPC on the Cray X1", in *Proc. of the 47th Cray User Group meeting (CUG'05)*, Albuquerque (NM), 10 pages, 2005.
14. A. Kayi, Y. Yao, T. A. El-Ghazawi, and G. Newby, "Experimental Evaluation of Emerging Multi-core Architectures", in *Proc. of the 6th Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO'07)*, Long Beach (CA), pages 1–6, 2007.
15. M. Curtis-Maury, X. Ding, C. D. Antonopoulos, and D. S. Nikolopoulos, "An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors", in *Proc. of the 1st Intl. Workshop on OpenMP (IWOMP'05)*, vol. 4315 of *LNCS*, Eugene (OR), pages 133–144, 2008.
16. "Finis Terrae Supercomputer." `http://www.top500.org/system/details/9500` [Last visited: June 2009].
17. G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, R. Doallo, J. C. Mouriño, D. A. Mallón, and A. Gómez, "Performance Evaluation of Unified Parallel C Collective Communications", in *Proc. of the 11th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC'09)*, Seoul (Korea), 10 pages, 2009.