2009 11th IEEE International Conference on High Performance Computing and Communications

Performance Evaluation of Unified Parallel C Collective Communications

Guillermo L. Taboada, Carlos Teijeiro, Juan Touriño,
Basilio B. Fraguela, Ramón Doallo
Computer Architecture Group, University of A Coruña (Spain)
Email: {taboada,cteijeiro,juan,basilio,doallo}@udc.es

José Carlos Mouriño, Damián A. Mallón, Andrés Gómez
Supercomputing Center of Galicia (CESGA)
Santiago de Compostela (Spain)
Email:{jmourino,dalvarez,agomez}@cesga.es

Abstract

Unified Parallel C (UPC) is an extension of ANSI C designed for parallel programming. UPC collective primitives, which are part of the UPC standard, increase programming productivity while reducing the communication overhead. This paper presents an up-to-date performance evaluation of two publicly available UPC collective implementations on three scenarios: shared, distributed, and hybrid shared/distributed memory architectures. The characterization of the throughput of collective primitives is useful for increasing performance through the runtime selection of the appropriate primitive implementation, which depends on the message size and the memory architecture, as well as to detect inefficient implementations. In fact, based on the analysis of the UPC collectives performance, we proposed some optimizations for the current UPC collective libraries. We have also compared the performance of the UPC collective primitives and their MPI counterparts, showing that there is room for improvement. Finally, this paper concludes with an analysis of the influence of the performance of the UPC collectives on a representative communication-intensive application, showing that their optimization is highly important for UPC scalability.

1. Introduction

As constellations and multi-core systems are increasing their popularity, there is a need for new programming models that provide an efficient support to application development on these architectures. Among the proposed approaches, UPC [1] has shown to be a good alternative to more traditional parallel programming models (e.g., message-passing, data parallel and shared-memory models).

UPC is an extension of ANSI C designed for parallel programming, which is especially suitable for hybrid shared/distributed memory architectures (e.g., multi-core clusters) because of its flexible memory model, the Partitioned Global Address Space (PGAS). The PGAS model presents a global memory address space logically partitioned among several threads. In the PGAS model each shared-memory portion has affinity with a particular thread, thereby providing a productive programming model while allowing the exploitation of locality. Although the PGAS model allows the transparent access to data affine to a remote thread, this usually introduces a significant overhead. In order to reduce the number of these inefficient operations, the UPC collectives specification [2], which is part of the standard UPC specification [3], defines a set of data movements and computational operations (primitives) commonly used in parallel applications. The implementation of these primitives, in a UPC collectives library, provides improvements in the programmability, as well as the locality (and hence the performance) in the data access.

Regarding UPC compilers, the most relevant open-source implementations are Berkeley UPC [4] (BUPC) and GCC UPC [5] (GCCUPC). BUPC includes a UPC-to-C translator and a runtime environment that uses a high-performance communication layer called GASNet [6] to provide support to PGAS on high-speed networks, such as Myrinet or InfiniBand. The BUPC collectives library (from now on denoted as "BCOL", Berkeley COLlectives library) is based on the GASNet collective implementation. GCCUPC is an extension of GNU GCC for UPC code, but it does not include its own implementation of collective primitives. Thus, GCCUPC has to resort to an external collective library, such as the reference implementation of the UPC collectives specification (from now on REF library) [7]. Moreover, GCCUPC can only be used in shared-memory systems by default, as it does not support distributed communications without a runtime environment. GCCUPC has not been used in this paper because of these constraints and also for showing usually poorer performance than BUPC [8]. Besides these two options, there are also commercial UPC compilers: HP UPC [9] and IBM UPC.

This paper presents up-to-date performance results of UPC primitives, and identifies the inefficiencies in different test environments in order to provide a better implementation for UPC collective-based operations. The rest of this paper is organized as follows. Section 2 discusses the related work on UPC performance evaluations. Section 3 presents the current implementations of UPC primitives and the different extensions and optimizations that have been proposed. Section 4 contains the results of our evaluation conducted on a multi-core InfiniBand supercomputer using different configurations. Finally, Section 5 concludes the paper.

978-0-7695-3738-2/09 $25.00 © 2009 IEEE
DOI 10.1109/HPCC.2009.88

2. Related Work

Previous UPC performance evaluations [8], [10], [11], [12] have focused either on basic data movement primitives or whole applications. Data movement primitives include `upc_memget`, `upc_memput` and `upc_memcpy`, which process all data types as byte arrays (raw data). Regarding the evaluated applications, the most relevant ones are general problem-solving algorithms, such as matrix multiplication, Sobel edge detection, N-Queens, and the UPC version of the NAS Parallel Benchmarks (NPB) [13], developed at George Washington University [1]. The application benchmarks allowed to measure overall results of UPC performance, whereas the microbenchmarking of memory primitives showed communication latencies and bandwidths.

A shared outcome from these studies is that UPC shows better performance when data locality and the use of private memory are maximized [14]. Collective primitives have not played an important role in these studies: research on this topic has been mostly restricted to different proposals for optimizing or extending the standard collectives specification. Recently, the introduction of MPI-like optimizations on UPC collectives has been proposed and experimentally evaluated in [15]. Nevertheless, these works only present relative comparisons between UPC collective libraries, measuring percentages of improvement, without characterizing their real performance, so the comparison with other implementations is not possible. Moreover, these three studies are restricted to the use of a small number of threads (up to 16) on a cluster of dual processor nodes, without evaluating this hybrid shared/distributed memory architecture.

The analysis of the throughput of collective primitives is useful for selecting the implementation that obtains the highest performance on a given scenario, to detect inefficient implementations, and eventually to propose new algorithms in order to increase their performance. As previous studies lack this analysis, this paper evaluates current UPC collectives performance in order to provide UPC developers with this valuable information.

3. Implementations of UPC Collectives

The UPC collectives specification includes several primitives that implement common communication patterns, such as broadcast, scatter, gather, exchange or reduce. These primitives operate in the shared-memory space, which implies that the source and destination arguments of these primitives are pointers to shared-memory locations. In order to improve the functionality of the standard collective specification, different extensions and optimizations have been proposed. The most relevant ones are Value-based Collectives [16], One-sided Collectives and Variable-sized Data Blocks Collectives [17]. Value-based Collectives optimize communications of single-valued variables, which can be either on private or shared memory. The One-sided approach defines communications in a single direction, with an active and a passive peer for each communication, thus simplifying synchronizations in data transfers. Variable-sized Data Blocks Collectives provide a more flexible set of collectives that define custom message sizes and source/destination pointers for each communication peer, allowing non-contiguous data movements and the transmission of data from private memory.

An additional project on UPC collectives is the definition of sets of threads called "teams", which allow a collective primitive to be called by a subset of all available threads [18]. Another active line is the research on extensions of the UPC memory copy library [19], that aims for an efficient implementation of non-blocking and non-contiguous data transfers.

The basis of UPC collective primitives are data transfers between threads, which can be implemented either with bulk data transfers, using UPC functions such as `upc_memcpy`, or relying on the collective implementation provided by an underlying communication library. Regarding the two UPC collective libraries evaluated in this study, the BUPC collectives library (BCOL) [4] is an example of the latter approach, as it relies on a low-level communication library (GASNet), whereas the UPC reference implementation (REF) [7] is implemented with `upc_memcpy` operations.

BCOL is based on the low-level GASNet communication library [6], which is implemented using Active Messages [20]. From version 2.6.0 of the BUPC compiler, the former linear flat-tree implementation of collectives has been replaced by a binomial tree communication pattern, which organizes data transfers in a logarithmic number of steps, thus reducing memory and network contention.

Regarding REF, from Michigan Tech. University, the implementation of its collective primitives is based on `upc_memcpy` data transfers. Its communications use a fully parallel flat-tree algorithm, so that they are all performed in only one step. Two different approaches can be used in REF collective primitives: pull and push. Both techniques are based on `upc_memcpy`, and their distinguishing feature is the active side in the communications. In the pull approach, each destination thread copies its corresponding data from the source thread in parallel, while in the push approach each source thread copies its data to all the destination threads. The selection of a pull or a push approach has to provide a fair distribution of the communication overhead for each collective primitive. Thus, in broadcast and scatter a pull implementation is better because it makes the destination threads copy the data in parallel from the source thread, whereas the push approach maximizes parallelism in the gather collective. In this study, the most efficient approach has been selected for each REF collective primitive evaluated.

4. Performance Evaluation of UPC Collectives

4.1. Experimental Configuration

The performance evaluation has been conducted on the Finis Terrae supercomputer, ranked #427 in the November 2008 TOP500 list (14 TFlops) [21]. It consists of 142 nodes, each of them with 8 Montvale Itanium2 (IA64) dual-core processors at 1.6 GHz (16 cores per node), 128 GB of memory and InfiniBand [22] as interconnection network. Additionally, the Finis Terrae has an HP Superdome with 64 Montvale Itanium2 processors (128 cores) at 1.6 GHz and 1 TB of shared memory.

The software configuration consists of a SuSE Linux Enterprise Server 10 (for IA-64) OS, the Intel C compiler icc 10.1 with OpenMP support and BUPC 2.6.0 as UPC compiler/runtime. BUPC uses HP-MPI v2.2.5.1 for distributed-memory communications, relying on its OpenFabrics InfiniBand Verbs (IBV) communication device for internode transfers, and on a special low-level messaging protocol (SHM) which avoids the use of the InfiniBand network device for intranode communications. Two UPC collective libraries have been selected for this work, BCOL and REF, which have been also used in some evaluations and subsequent optimizations of UPC collective libraries [15].

This supercomputer allows the evaluation of UPC collective primitives on three scenarios: shared memory, distributed memory, and hybrid shared/distributed memory. In the shared-memory configuration (from now on denoted as “SMP”) the benchmarks are run on the shared-memory machine, the Finis Terrae Superdome, obtaining results up to 128 threads. In the distributed-memory scenario (from now on “DMP”) up to 8 nodes are used and the communication among threads is done exclusively using the InfiniBand Verbs (IBV) support provided by GASNet. The evaluation on DMP has been done using all the available combinations of number of nodes (up to 8) and number of threads per node (up to 16), obtaining results up to 128 threads. The third scenario is the hybrid shared/distributed memory configuration (denoted as “Hybrid”), where up to 8 nodes are used and internode communications are also done over InfiniBand using the GASNet IBV device. Nevertheless, intranode communications are shared-memory accesses. The results in the Hybrid scenario are also obtained for up to 128 threads using all the combinations of number of nodes (up to 8) and number of threads per node (up to 16). Among these three scenarios, the hybrid-memory layout is of particular interest in the PGAS model, as it increases the scalability of the shared-memory model, allowing the aggregation of computing resources with distributed memory, and it avoids the overhead incurred when the distributed-memory applications are used within shared-memory machines. Nevertheless, no evaluation of UPC collectives performance has been done up to now on a hybrid-memory architecture despite being the most commonly deployed (e.g., multi-core clusters) nowadays.

4.2. UPC Collectives Microbenchmark Suite

Due to the lack of suitable benchmarks for our purposes, we have implemented our own UPC collectives suite, which is similar to the Intel MPI collectives benchmarks (previously known as Pallas MPI) [23]. This suite has been designed to measure the performance of every collective primitive through a single call to a generic benchmarking function, which tests the performance of the primitive in a range of message sizes. Although this suite can be used to characterize the performance of all collective primitives present in the UPC specification, only five have been selected for our evaluation: broadcast, scatter, gather, exchange and reduce.

In order to avoid the issues that might arise when microbenchmarking communications performance, the benchmarkmarking suite has been designed following most of the guidelines presented in [24]. For example, the UPC_{IN,OUT}_ALLSYNC (strict) synchronization mode has been used in all collective calls (in UPC the synchronization mode is passed to each collective primitive call as a function parameter). The main goal of this approach is to characterize the maximum overhead that the synchronization can impose in a collective primitive operation. Furthermore, the results have been obtained using cache invalidation in order to avoid the influence of cache reuse. This technique has been implemented using new dynamically allocated buffers for each primitive call, without reuse. The design of the tests implicitly forces the obtention of correct results, but simple sanity checks are also performed here. The performance results of UPC collectives obtained with our microbenchmark suite are discussed in the next section.

4.3. UPC Collective Primitives Performance

Figures 1-4 and Table 1 show aggregated bandwidths of UPC collectives (latencies in the case of reduce) for BUPC using two UPC collective implementations, the BUPC default library (BCOL) and REF, on the three available configurations on Finis Terrae (SMP, DMP and Hybrid). The main difference between the collective implementations is that REF uses flat-tree communication algorithms, whereas BCOL resorts to binomial trees. The data size shown in the figures is the size of the data used in the collective operation at the root thread, or in any thread in a non-rooted operation. Thus, in a 1 MB scatter primitive (rooted operation) (1 MB)/$THREADS$ of data is sent to each thread ($THREADS$ is the number of UPC threads involved in the collective operation). The aggregated bandwidth metric includes the minimum number of bytes that need to be transferred in the collective primitive operation, thus allowing to

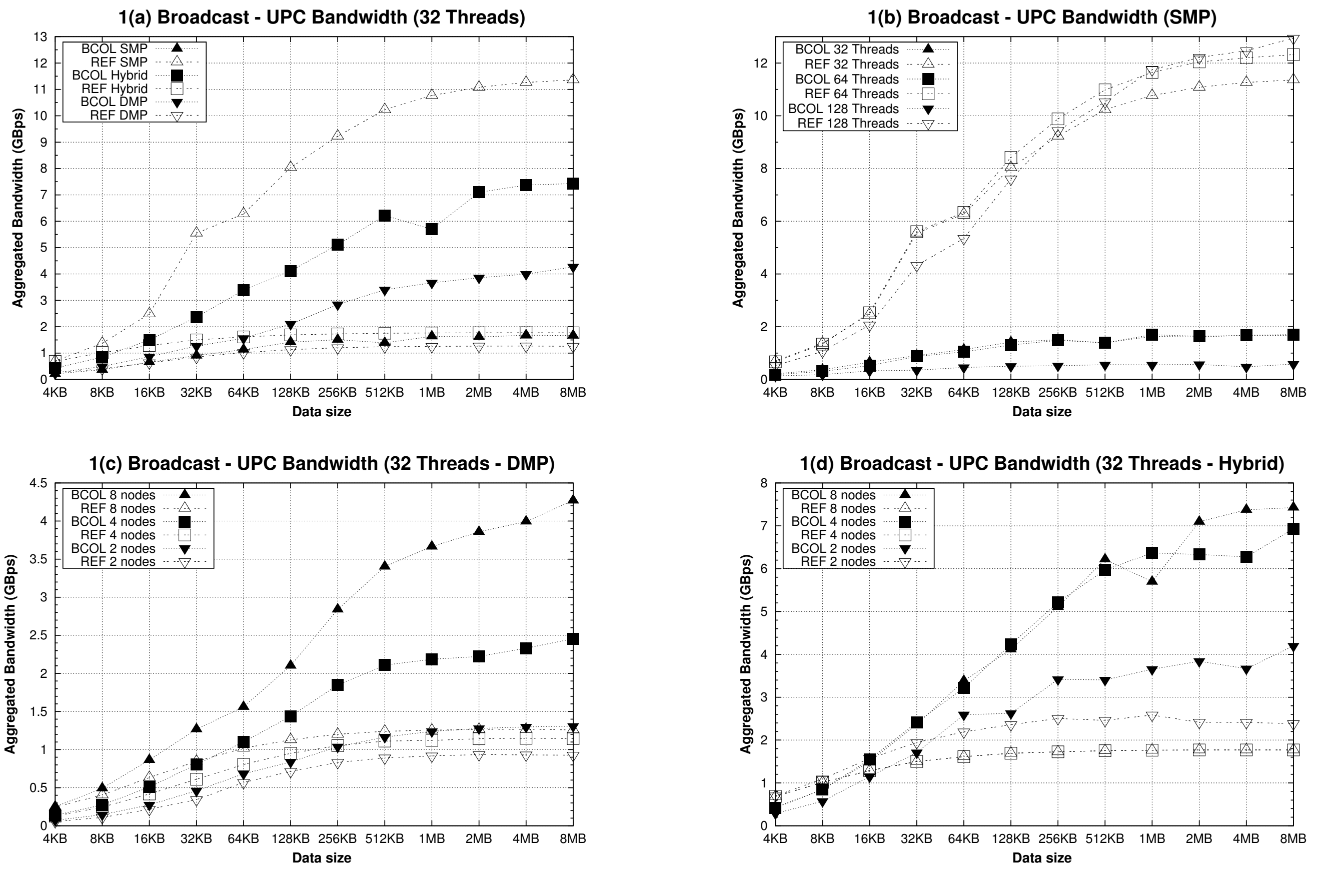

Figure 1. UPC broadcast performance

compare the performance obtained using different numbers of threads (performance scalability). It has been calculated as $data\_size/latency$ for the scatter and gather measures, and as $THREADS * data\_size/latency$ for the broadcast and exchange results.

Each figure (from Figures 1-4) consists of four graphs. The first one (a), shows the performance of the two collective implementations (BCOL and REF) using 32 threads on the three configurations, thus allowing the comparison among them. The Hybrid and DMP results have been obtained using 8 nodes and 4 threads per node. The second graph (b) presents the aggregated bandwidths for 32, 64 and 128 threads on SMP, allowing the analysis of the performance scalability. Graph (c) depicts the collective aggregated bandwidth with 32 threads on the DMP scenario varying the number of nodes used (2, 4 or 8 nodes, which means using 16, 8 or 4 UPC threads per node, respectively). A similar analysis (varying the number of nodes used) is also done for the Hybrid configuration in the last graph (d). Table 1 presents a summary of the results that would appear in a (a)-like graph for the reduce collective, but showing latencies instead of bandwidths.

Figure 1 shows the performance of the broadcast collective. Regarding the Graph 1(a), the best results have been obtained with REF SMP. The reason is the parallel access of the destination threads to the root thread data, without any additional synchronization. In fact, BCOL SMP performs the binomial tree in five steps ($log_2 32$), obtaining quite poorer performance than REF on SMP due to the synchronization overhead involved in a five step operation compared to the single step required for REF. The second best performance is achieved by BCOL Hybrid which, taking advantage of the shared-memory transfers, almost doubles the performance of BCOL DMP. The REF Hybrid and DMP results are poor, as they involve several internode transfers, which are an important performance bottleneck for a flat-tree algorithm. In fact, BCOL Hybrid and DMP outperform their REF counterpart, emphasizing the fact that the minimization of internode transfers improves the performance of the collectives. Graph 1(b) shows that REF significantly outperforms BCOL on SMP, which obtains comparatively quite poor performance, especially for 128 threads. Moreover, the scalability of both implementations is quite small. Regarding Graph 1(c), the best performance is obtained by BCOL using 8 nodes (4 threads per node). In this scenario, REF obtains poorer performance than BCOL as in a flat-tree implemen-

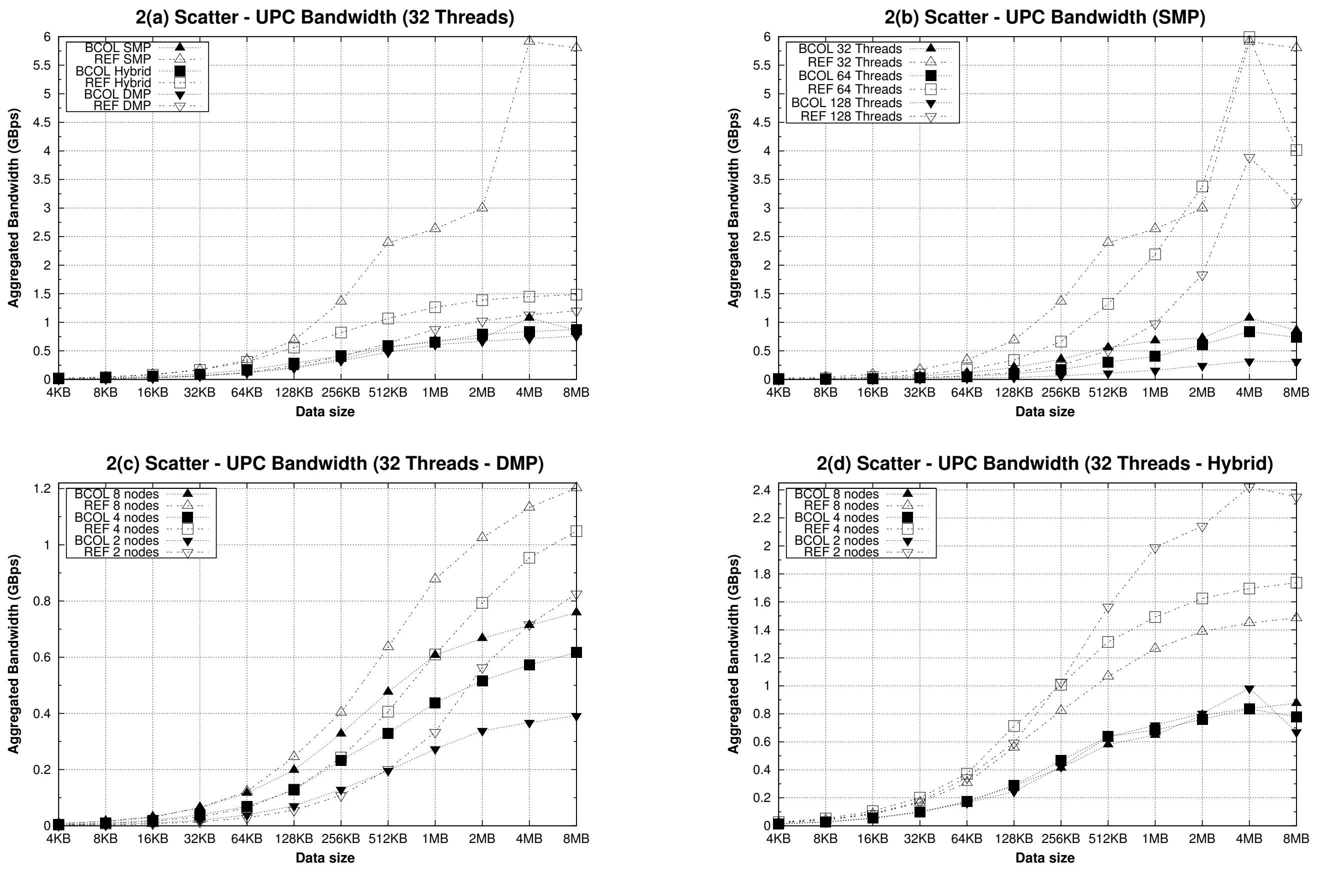

Figure 2. UPC scatter performance

tation the use of distributed-memory communications is an important performance bottleneck. From the analysis of the results of Graph 1(d) it can be derived that there is not much difference between using 8 nodes with 4 threads per node, and using 4 nodes with 8 threads per node. The best performance in this Hybrid scenario has been obtained by BCOL, although REF increases the throughput shown on the DMP configuration, especially on 2 nodes. In fact, the use of 2 nodes, and hence 16 threads per node, maximizes the number of intranode transfers, which benefits from the flat-tree algorithm of REF, whereas it harms BCOL performance.

Figure 2 presents the results of UPC scatter. Graph 2(a) shows that the best performance has been obtained by REF on SMP thanks to its parallel access to the source thread, which avoids synchronization steps and data buffering in intermediate threads. Regarding BCOL, its best results are obtained in the Hybrid configuration. Graph 2(b) shows that both implementations have poor scalability, obtaining the best results with 32 threads. In this scenario REF significantly outperforms BCOL. In Graph 2(c) REF almost doubles BCOL results, achieving its best performance using 8 nodes. This is the opposite behavior to the broadcast, where BCOL obtains better results than REF. In this case the BCOL scatter (binomial tree) has to transfer additional data for all the leaves of a node (intermediate buffering). For example, in a 1 MB scatter to 8 threads using a binomial tree it is required that the source thread transfers 512 KB in the first step, the two threads with data (the source and the one that got the data in the first step) will transfer 256 KB in the second step, and finally, four threads will copy 128 KB to the leaves of the binomial tree. Thus, BCOL requires the movement of 1536 KB whereas REF only 1024 KB, which means an overhead in terms of extra data transferred of 50% of the data size considered in the primitive. Regarding a 32-thread operation, the additional data overhead is 153% of the data size considered in the primitive. Thus, for a 1 MB scatter, 1.53 MB of additional data are transferred. Therefore, REF scatter obtains higher throughput as it transmits the minimum amount of data without synchronization overheads. This efficient implementation allows REF to also outperform BCOL in the Hybrid configuration (Graph 2(d)), although in this case the best results have been obtained using 2 nodes, and hence 16 threads per node, where the number of shared-memory transfers is maximized.

Figure 3 depicts the results of UPC gather. Similarly to scatter, REF usually outperforms BCOL on all configura-

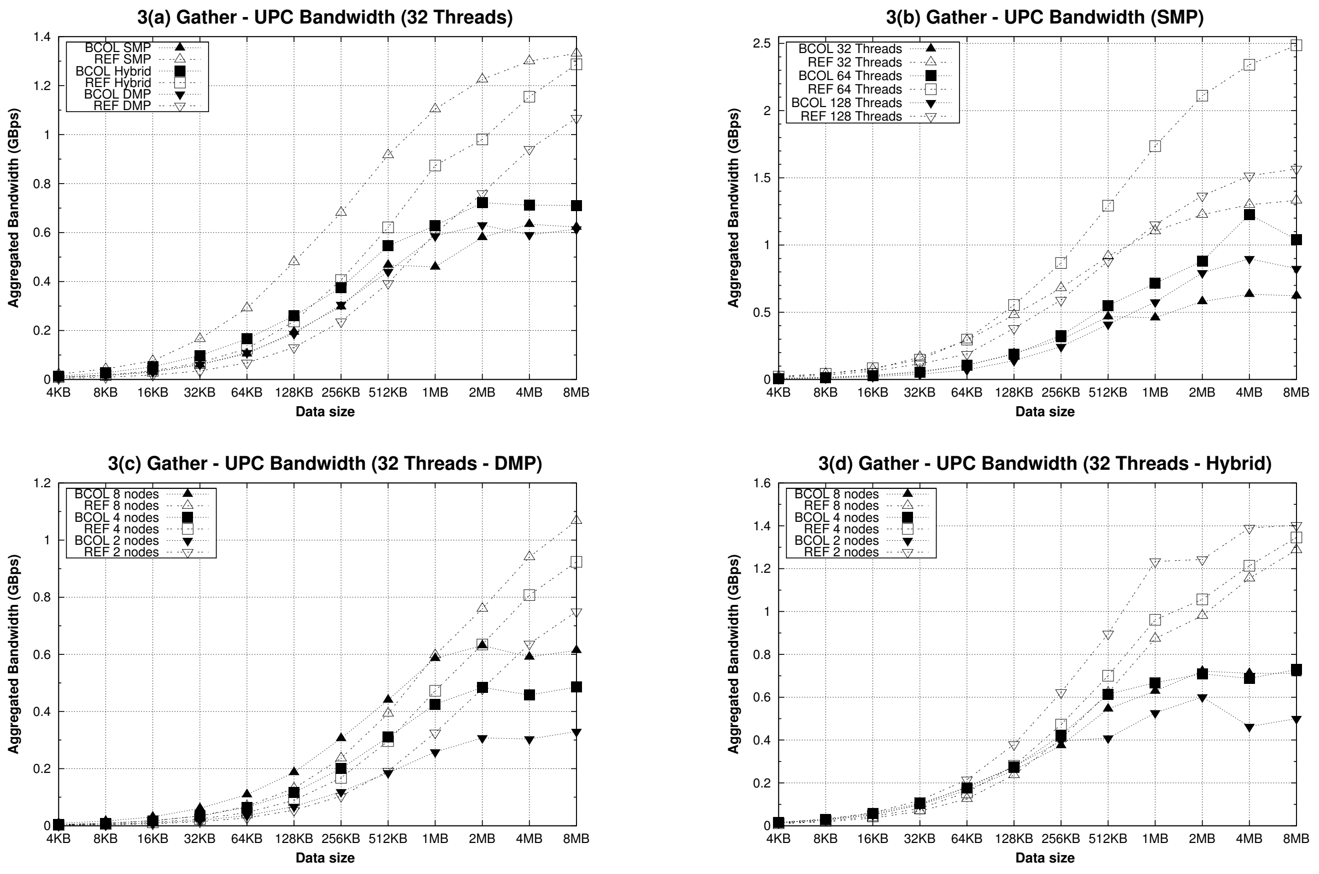

Figure 3. UPC allgather performance

tions. In fact, the evaluation of all the graphs of gather shows analogous conclusions to the previous primitive. However, the performance of REF gather is lower that that of scatter due to memory access performance. In Graph 3(a) the best performance has been also obtained by REF SMP, although the aggregated bandwidth values are significantly lower than for the scatter. Graph 3(b) shows that the highest throughput is obtained using 64 threads, both for REF and BCOL. In Graph 3(c) REF gather (like REF scatter) almost doubles BCOL, achieving the best performance using 8 nodes. Finally, REF gather achieves its highest performance on the Hybrid scenario (Graph 3(d)) with 2 nodes (maximizing the number of shared-memory transfers), whereas the best BCOL gather results have been obtained with 4 and 8 nodes (maximizing the number of internode transfers).

Figure 4 shows the exchange results. This primitive involves a more complex communication pattern than the preceding ones. Thus, its performance is highly influenced by the start-up communication latency, and thus the synchronization, showing REF on SMP much better performance than the other configurations, taking advantage of the shared-memory access and its flat-tree implementation. Furthermore, the aggregated bandwidth is usually higher than that of the preceding collectives as it is a non-rooted collective where all threads are actively communicating during the collective operation. The analysis of the performance of this primitive on SMP (Graph 4(b)) shows an important scalability for REF, especially for long messages, increasing the aggregated bandwidth almost linearly with the number of threads. In this scenario, the high number of transfers involved in this operation benefits from the flat-tree implementation of REF collectives. However, BCOL has an inefficient implementation on SMP, showing quite poor performance. In fact, BCOL exchange obtains worse performance on SMP than on DMP and Hybrid. Regarding DMP results (Graph 4(c)), BCOL slightly outperforms REF, whereas for the Hybrid configuration (Graph 4(d)) the performance gap is wider for BCOL on 4 and 8 nodes, but REF outperforms BCOL on 2 nodes. Both BCOL and REF implementations benefit from the use of 8 nodes on DMP and Hybrid, compared to the use of 4 and 2, as for this non-rooted operation all threads are actively communicating, thus taking advantage of the highest number of nodes.

Table 1 shows the latencies (in microseconds) of UPC reduce. The latency has been selected as performance measure instead of the bandwidth as the UPC reduce only

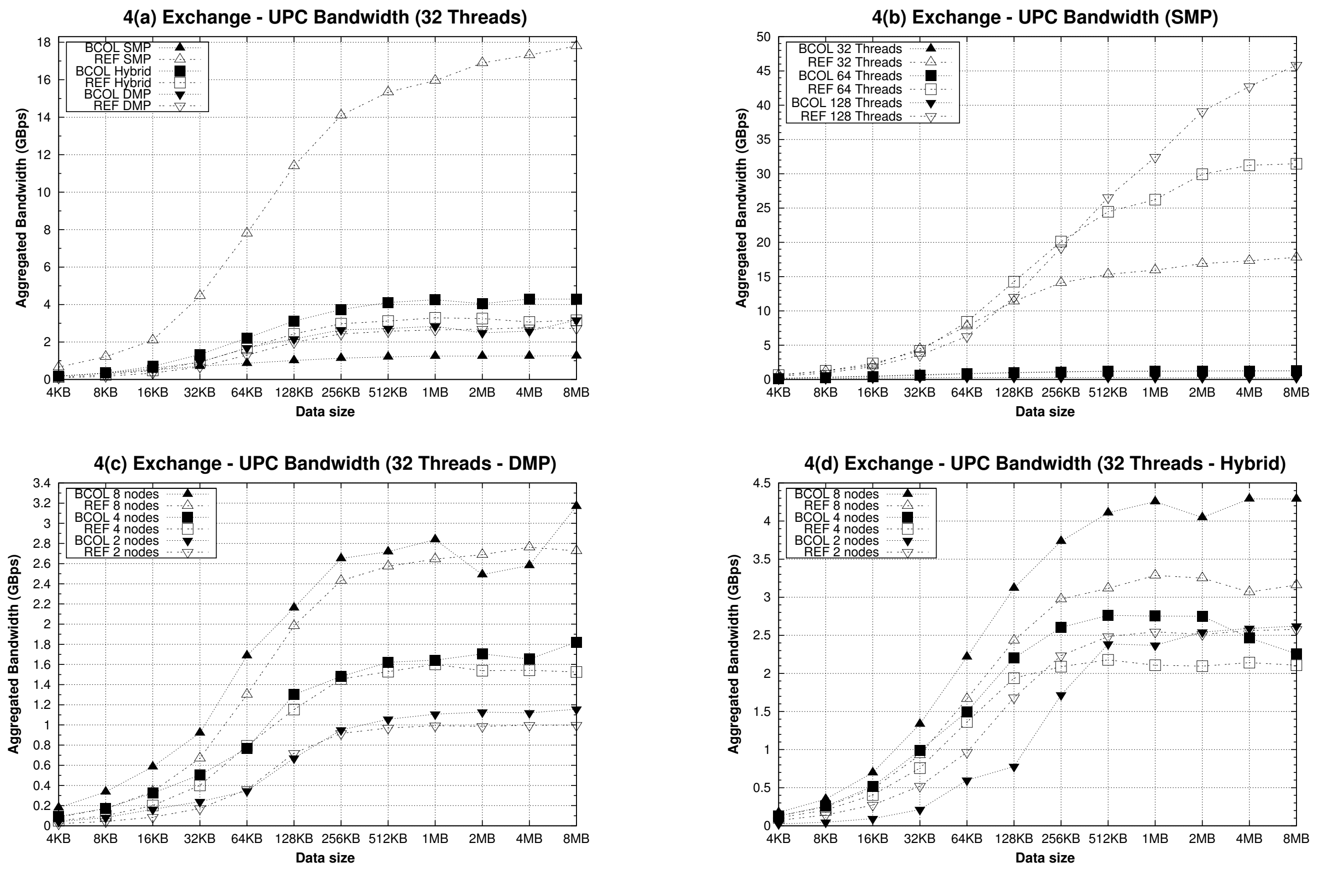

Figure 4. UPC exchange performance

		Data size	
		1KB	1MB
BCOL	SMP	622	624
	Hybrid	213	354
	DMP	531	641
REF	SMP	653	5191
	Hybrid	287	3599
	DMP	716	3980

Table 1. UPC reduce latencies (μs) for 32 threads

involves the transfer of a primitive data type value per thread, independently of the number of elements being processed. The operation of the UPC reduce differs from the MPI reduction, which communicates all the local data. Thus, the reduction of an array of 10 elements per thread/process returns a scalar result in UPC and a 10 element result array for MPI. The operation used in the microbenchmarking is the floating point addition of double precision data (doubles). Unlike the previous data movement collectives, reduce is a computational one, and therefore its UPC implementation is more intensive in computations than in communications. Thus, in this scenario, it can be concluded that the computation associated to a reduce call happens to be implemented more efficiently in BCOL reduce than in REF, because BCOL clearly outperforms REF especially for long messages. Regarding reduce 1 KB performance, BCOL outperforms REF on DMP and Hybrid, whereas it shows similar results to REF on SMP.

From the analysis of the performance results presented in Figures 1-4 and Table 1 it can be concluded that: (1) there are significant performance differences between BCOL and REF, up to 2000% (exchange SMP); therefore, it is possible to increase UPC throughput by selecting the best collective library at runtime for each configuration and message size; (2) UPC can take important advantage of the Hybrid configuration, increasing significantly the performance shown in the equivalent configuration in the DMP scenario (up to 100%); (3) the best UPC collectives performance is usually obtained by REF on SMP; (4) REF reduce, BCOL broadcast on SMP, and BCOL exchange on SMP are examples of collective primitives implemented inefficiently; and (5) it is possible to optimize collective operations minimizing the number of internode communications and using a flat-tree algorithm for shared-memory transfers (SMP or Hybrid scenarios) on intranode communication.

	Broadcast		Scatter		Gather		Exchange/Alltoall	
Library \ Message size	1 KB	1 MB	1 KB	1 MB	1 KB	1 MB	1 KB	1 MB
MPI (GBps)	0.0992	4.4013	0.0088	1.5360	0.0183	1.5627	0.0066	0.0971
BCOL SMP	3%	22%	23%	44%	23%	31%	21%	40%
REF SMP	8%	**145%**	61%	**171%**	61%	74%	82%	**514%**
BCOL DMP	2%	30%	13%	28%	13%	28%	11%	53%
REF DMP	1%	15%	11%	40%	9%	32%	6%	52%
BCOL Hybrid	5%	86%	43%	44%	45%	45%	16%	89%
REF Hybrid	9%	24%	72%	97%	3%	64%	13%	68%

Table 2. UPC vs. MPI collectives performance (32 threads/processes, 4 nodes, MPI = 100%)

4.4. UPC vs. MPI Collective Primitives Performance Analysis

This subsection presents a comparative analysis of the performance of the UPC collectives and their MPI counterparts. As MPI collectives have been thoroughly optimized for years, the gap between MPI and UPC collectives performance can be considered a good estimate of the quality of a UPC implementation. However, UPC will not always lag behind MPI, as it is expected that UPC collectives outperform MPI when shared-memory transfers are involved. Table 2 shows the relative performance of UPC compared to MPI (HP-MPI v2.2.5.1), where UPC collectives throughput is shown as a percentage of the MPI performance. Thus, UPC outperforms MPI when the percentage is higher than 100%, which only happens for some primitives with 1 MB messages on REF SMP. The reduce comparison is not shown, as the UPC reduce primitive has no equivalent operation in MPI (MPI reduce transfers an array instead of a single variable). These results have been obtained for two representative message sizes, 1 KB and 1 MB. An analysis of the results shows that the UPC performance in the Hybrid configuration, although significantly better than for DMP, is lower than MPI results. Furthermore, UPC suffers from higher start-up latencies than MPI, which means poor performance for 1 KB messages, especially for the broadcast. This comparative analysis of MPI and UPC collectives performance serves to assess that there is room for improvement in the implementation of the UPC collectives, as UPC could outperform MPI on the SMP and Hybrid scenarios, whereas it could rival MPI on DMP.

4.5. Performance Evaluation of UPC Applications with Collective Primitives

This subsection analyzes the performance shown by an application written in UPC using collective operations. The selected code is the FT benchmark from the UPC implementation of the NAS Parallel Benchmarks (NPB) [13], distributed with BUPC [4], as it is a communication-intensive application that performs an important number of calls to the exchange primitive. A preliminary performance result obtained with BCOL using 128 threads on the SMP scenario showed that the exchange operation overhead was 96% of the overall runtime of FT with workload B, confirming that FT is a suitable code for analyzing UPC collective implementations.

In order to analyze the potential performance increase in UPC collectives of the optimizations presented in this paper, we have developed a proposal for a more efficient UPC collective library (from now on denoted as "PROP"). PROP is based on REF and hence relies on `upc_memcpy` calls. It is intended to take advantage of the memory configuration and the data locality to reduce memory and data movements contention. However, PROP only uses the flat-tree algorithms for intranode transfers, whereas the binomial-tree communications are used for internode communications. Moreover, in SMP it uses `sched_setaffinity` libc system calls in order to improve performance maximizing the distance among the physical cores being used. PROP implements this technique setting the affinity of UPC threads (identified by $MYTHREAD$, which goes from 0 up to $THREADS$-1) to physically non-contiguous cores (e.g., when using 32 threads on a 128 core machine the affinity of the UPC thread $MYTHREAD$ is set to the core 4*$MYTHREAD$). Thus, the communication bandwidth is maximized as the main performance bottleneck in shared memory systems is the memory access performance, which achieves the highest performance with this approach.

Figure 5 shows the performance (measured in MOPS, millions of operations) of OpenMP, MPI, and UPC with BCOL/REF/PROP versions for FT on the SMP scenario (SMP has been selected because it allows a comparative evaluation against OpenMP). FT using BCOL obtains quite poor performance. However, the throughput obtained with REF exchange is significantly higher, and much more scalable. This analysis agrees with the evaluation of the exchange primitive presented in Section 4.3 (see Graph 4(b)). Additionally, FT with our PROP exchange implementation has been also evaluated, obtaining the best results of the UPC collectives. Regarding PROP results, the highest throughput increases have been obtained on 16, 32 and 64 cores (up

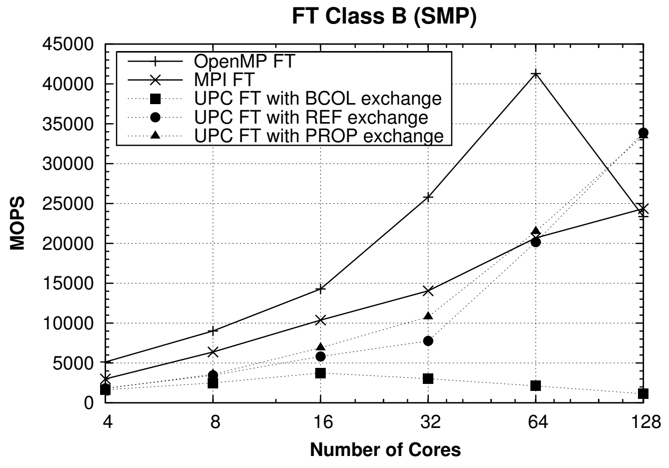

Figure 5. Performance comparison of OpenMP, MPI and UPC versions of NPB FT

to a 30% increase) due to a a more efficient mapping of UPC threads to cores through `sched_setaffinity`, maximizing the distance among threads in order to improve the throughput in the access to memory. However, its results on 128 cores are similar to REF exchange as the use of all the cores of the system (Finis Terrae Superdome) reduces the effectiveness of PROP. Nevertheless, although UPC implementations obtain the lowest performance using up to 32 cores, on 128 cores REF and PROP eventually outperform both OpenMP and MPI.

Regarding OpenMP, it shows quite high performance using up to 64 cores (in fact, it obtains the maximum throughput, above 40.000 MOPS), although using 128 cores it has the worst result. Nevertheless, MPI has the opposite behavior. Thus, although it presents less performance than OpenMP up to 64 cores, it takes advantage of the use of 128 cores. In this latter case, OpenMP presents lower performance because of its poor data locality support. In this case MPI is run on an SMP scenario, but HP-MPI on intranode resorts to an efficient low-level messaging protocol (HP-MPI SHM), which obtains similar performance to MPI on the DMP configuration (with HP-MPI IBV).

These observations are in tune with the expected results. Thus, OpenMP presents high performance but relatively low scalability as it does not take into account the data locality. MPI usually presents a scalable performance on most of the current architectures as it works on private data. However, its results are usually lower than those of OpenMP. Finally, UPC takes advantage of shared-memory communications while considering data locality, especially using PROP, which allows for higher speedups. Nevertheless, current UPC compilers/runtimes and collective libraries are not mature enough to significantly outperform OpenMP and MPI.

5. Conclusions

This paper has presented an up-to-date performance evaluation of two UPC collective libraries on three configurations: shared, distributed and hybrid shared/distributed memory. The main conclusions of this work are: (1) there is a lack of collective primitives benchmarks, so we have implemented our own UPC collectives microbenchmark suite, which is similar to a widely spread suite for MPI collectives (Intel MPI microbenchmarks); (2) the two collective implementations evaluated present significant differences in performance, which depend on the memory architecture, the message size and the communication pattern of the primitive. However, as a general rule, the collectives reference implementation (REF) achieves the best performance on shared memory, whereas BUPC collectives implementation (BCOL) usually presents the best results for reduce on all configurations, and for broadcast and exchange on the distributed-memory and hybrid scenarios; (3) it is possible to achieve important performance increases by automatically selecting the best collective primitive implementation at runtime; (4) UPC collective primitives take advantage of the use of hybrid shared/distributed memory configurations, currently the most commonly deployed ones (e.g., multicore clusters); (5) inefficient implementations of collective primitives have been detected, such as REF reduce, and BCOL broadcast and exchange on shared memory; (6) UPC obtains quite poor collective performance compared to MPI, although REF outperforms MPI on shared memory; moreover, the best comparative results are obtained for long messages, as UPC suffers from high start-up communication latencies. This comparative evaluation shows that there is room for performance improvement in UPC collectives libraries. And (7) an analysis of the influence of the performance of the UPC collectives on a representative communication-intensive application has shown that UPC codes can significantly benefit from the optimization of the collective primitives, even outperforming the scalability of OpenMP and MPI on a shared memory scenario.

Finally, it can be concluded that UPC codes can take full advantage of the use of efficient and scalable collective primitives. Thus, the characterization of their performance is highly important. Furthermore, the higher programmability provided by collective primitives, together with their locality exploitation, improves the productive development of efficient parallel applications in UPC.

As future work we intend to develop a more efficient UPC collective library that would take into account the locality and the communication overhead among all threads. Thus, in a hybrid shared/distributed memory architecture this library would minimize the number of remote memory operations. Moreover, the shared-memory (local) accesses can be improved taking advantage of the affinity of UPC threads in order to improve the memory throughput.

Our prototype of collective library, PROP, has achieved significant improvements in the performance of the evaluated UPC application (up to a 30% performance increase).

Acknowledgments

This work was funded by Hewlett-Packard and partially supported by the Ministry of Science and Innovation of Spain under Project TIN2007-67537-C03-02 and by the Galician Government (Xunta de Galicia, Spain) under the Consolidation Program of Competitive Research Groups (Ref. 3/2006 DOGA 13/12/2006). 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] George Washington University, "Unified Parallel C at GWU," http://upc.gwu.edu [Last visited: April 2009].

[2] E. Wiebel, D. Greenberg and S. R. Seidel, "UPC Collective Operations Specifications v1.0," http://www.gwu.edu/upc/docs/UPC_Coll_Spec_V1.0.pdf [Last visited: April 2009].

[3] UPC Consortium, "UPC Language Specifications v1.2. (May 31, 2005)." http://upc.lbl.gov/docs/user/upc_spec_1.2.pdf [Last visited: April 2009].

[4] UC Berkeley / LBNL, "Berkeley Unified Parallel C (UPC) Project," http://upc.lbl.gov [Last visited: April 2009].

[5] Intrepid Technology Inc, "GCC Unified Parallel C," http://www.intrepid.com/upc.html [Last visited: April 2009].

[6] UC Berkeley, "GASNet Communication System," http://gasnet.cs.berkeley.edu [Last visited: April 2009].

[7] Michigan Tech, "Collectives Reference Implementation," http://www.upc.mtu.edu/collectives/col1.html [Last visited: April 2009].

[8] Z. Zhang and S. Seidel, "Benchmark Measurements of Current UPC Platforms," in *Proc. 4th Workshop on Performance Modeling, Evaluation and Optimization of Parallel and Distributed Systems (PMEO'05)*, Denver (CO), 2005, p. 276b (8 pages).

[9] Hewlett-Packard Inc., "HP Unified Parallel C (HP UPC)," http://hp.com/go/upc/ [Last visited: April 2009].

[10] T. El-Ghazawi and F. Cantonnet, "UPC Performance and Potential: a NPB Experimental Study," in *Proc. 15th ACM/IEEE Conference on Supercomputing (SC'02)*, Baltimore (MD), 2002, p. 1–26.

[11] T. El-Ghazawi and F. Cantonnet and Y. Yao and S. Annareddy and A. S. Mohamed, "Benchmarking Parallel Compilers: A UPC Case Study," *Future Generation Computer Systems*, vol. 22, no. 7, pp. 764–775, 2006.

[12] C. Coarfa et al., "An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C," in *Proc. 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'05)*, Chicago (IL), 2005, pp. 36–47.

[13] NASA Advanced Computing Division, "NAS Parallel Benchmarks," http://www.nas.nasa.gov/Software/NPB/ [Last visited: April 2009].

[14] T. El-Ghazawi and S. Chauvin, "UPC Benchmarking Issues," in *Proc. 30th Intl. Conference on Parallel Processing (ICPP'01)*, Valencia (Spain), 2001, pp. 365–372.

[15] R. A. Salama and A. Sameh, "Potential Performance Improvement of Collective Operations in UPC," *Advances in Parallel Computing*, vol. 15, pp. 413–422, 2008.

[16] D. Bonachea, "UPC Collectives Value Interface, v1.2," http://upc.lbl.gov/docs/user/README-collectivev.txt [Last visited: April 2009].

[17] Z. Ryne and S. Seidel, "Ideas and Specifications for the new One-sided Collective Operations in UPC," http://www.upc.mtu.edu/papers/OnesidedColl.pdf [Last visited: April 2009].

[18] R. Nishtala, G. Almasi, and C. Cascaval, "Performance without Pain = Productivity: Data Layout and Collective Communication in UPC," in *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, Salt Lake City (UT), 2008, pp. 99–110.

[19] D. Bonachea, "Proposal for Extending the UPC Memory Copy Library Functions, v2.0," http://upc.lbl.gov/publications/upc_memcpy.pdf [Last visited: April 2009].

[20] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active Messages: a Mechanism for Integrating Communication and Computation," in *Proc. 25th Intl. Symposium on Computer Architecture (ISCA'98)*, Barcelona (Spain), 1998, pp. 430–440.

[21] "Finis Terrae Supercomputer at TOP500 List," http://www.top500.org/system/9156 [Last visited: April 2009].

[22] "InfiniBand Trade Association," http://www.infinibandta.org [Last visited: April 2009].

[23] Intel Corporation, "Intel MPI Benchmarks," http://www.intel.com/cd/software/products/asmo-na/eng/219848.htm [Last visited: April 2009].

[24] W. Gropp and E. Lusk, "Reproducible Measurements of MPI Performance Characteristics," in *Proc. 6th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'99). Lecture Notes in Computer Science vol. 1697*, Barcelona (Spain), 1999, pp. 11–18.