# Design and Implementation of an Extended Collectives Library for Unified Parallel C

Carlos Teijeiro[1], *Student Member, IEEE*, Guillermo L. Taboada[1], Juan Touriño[1], *Senior Member, IEEE, Member, ACM*, Ramón Doallo[1], *Member, IEEE*, José C. Mouriño[2], Damián A. Mallón[3], and Brian Wibecan[4]

[1]*Computer Architecture Group, University of A Coruña, A Coruña 15071, Spain*

[2]*Galicia Supercomputing Center (CESGA), Santiago de Compostela 15705, Spain*

[3]*Jülich Supercomputing Centre, Institute for Advanced Simulation, Forschungszentrum Jülich, Jülich D-52425, Germany*

[4]*Industry Standard Servers Group, Hewlett-Packard Company (HP), Montgomery, Alabama 36117, USA*

E-mail: cteijeiro,taboada,juan,doallo}@udc.es, jmourino@cesga.es, d.alvarez.mallon@fz-juelich.de, brian.wibecan@hp.com

Received month day, year

**Abstract** Unified Parallel C (UPC) is a parallel extension of ANSI C based on the Partitioned Global Address Space (PGAS) programming model, which provides a shared memory view that simplifies code development while it can take advantage of the scalability of distributed memory architectures. Therefore, UPC allows programmers to write parallel applications on hybrid shared/distributed memory architectures, such as multi-core clusters, in a more productive way, accessing remote memory by means of different high-level language constructs, such as assignments to shared variables or collective primitives. However, the standard UPC collectives library includes a reduced set of eight basic primitives with quite

This work has been funded by Hewlett-Packard (Project "Improving UPC Usability and Performance in Constellation Systems: Implementation/Extensions of UPC Libraries") and partially supported by the Ministry of Science and Innovation of Spain under Project TIN2010-16735.

limited functionality. This work presents the design and implementation of extended UPC collective functions that overcome the limitations of the standard collectives library, allowing, for example, the use of a specific source and destination thread or defining the amount of data transferred by each particular thread. This library fulfills the demands made by the UPC developers community and implements portable algorithms, independent of the specific UPC compiler/runtime being used. The use of a representative set of these extended collectives has been evaluated using two applications and four kernels as case studies. The results obtained confirm the suitability of the new library to provide easier programming without trading off performance, thus achieving high productivity in parallel programming to harness the performance of hybrid shared/distributed memory architectures in High Performance Computing (HPC).

**Keywords** Unified Parallel C (UPC), collective operations, programmability, Partitioned Global Address Space (PGAS), High Performance Computing (HPC)

## 1 Introduction

Although multi-core processors mitigate single-core processor problems, such as the power wall, the memory wall and the instruction-level parallelism wall, they have raised the programmability wall. Thus, current developers, generally trained only for the development of sequential programs, have to confront the growing complexity of programming clusters of multi-core processors. In this scenario the use of a suitable parallel programming model is highly recommended in order to facilitate the development for multi-core platforms, compared to the cumbersome process of using a sequential programming model together with a parallel library. This paradigm shift from sequential to parallel programming models demands an extensive number of available tools and libraries in order to support the productive development of parallel software.

The PGAS parallel programming model is grabbing the attention of developers of parallel applications looking for programmability. This model provides a shared memory view that simplifies code development while it can take advantage of the scalability of distributed memory architectures. In PGAS languages each program is executed by N threads that have two different memory spaces: (1) a private space that is only accessible by each thread, and (2) a shared space that allows communication among threads. Collective primitives, which involve data movements and computational operations among several threads, contribute significantly to the programmability of PGAS as they implement common operations such as broadcast, scatter and gather of data, thus allowing a more rapid development.

One of the most extended PGAS languages is Unified Parallel C (UPC) [1], a parallel extension of ANSI C. Every C code can be run in parallel with UPC, thus favoring its adoption. Moreover, UPC programmability is supported both by: (1) standard language constructs, such as implicit data transfers in assignments of shared variables and the predefined constants *THREADS* (total number of threads in a program) and *MYTHREAD* (identifier of each thread), and (2) libraries that provide high level constructs, such as collective functions, which is the focus of this paper. The UPC standard collectives library [2], which is part of the UPC standard specification [3], includes eight data-movement (e.g., broadcast, scatter and gather) and computational (reduce) functions, most of them already used in traditional parallel programming approaches, such as message passing libraries like MPI. However, UPC collectives have not become very popular because of two main reasons: (1) the generally low performance of many of these functions, which has led programmers to replace them by bulk data copy functions for efficiency purposes [4] although at the cost of increasing programming complexity, and (2) some limitations for their use that make them unsuitable for different situations: for instance, source and destination arrays must be different and stored on shared memory, and the amount of data per thread involved in the operation should be the same. In the last years, the UPC community [5] has made different proposals in order to provide extended functionality to the standard collectives library, but their implementation is still pending.

This paper presents a new library of extended UPC collective functions that aims to improve programmability in UPC by addressing the current limitations of the standard collectives library. The organization of this work is as follows. First, Section 2 comments the most relevant related work on UPC collective functions and the motivation of this work. After that, the design and implementation of the collectives is explained in Section 3, discussing the decisions and strategies that have driven the development of this library. Section 4 presents six examples of application of these collectives, in order to illustrate the suitability and potential benefits of their use. The evaluated codes are four computational kernels (Matrix Multiplication, both for dense and sparse computation, Integer Sort and 3D Fast Fourier Transform), and two MapReduce applications. Section 5 shows the performance results of the previous codes, and finally, the conclusions derived from this work are discussed in Section 6.

## 2 Related Work and Motivation

The main goal of this work is to provide new collective functions that overcome the limitations of the standard UPC collectives library, and also addressing any performance issues on them in a portable way. Up to now, the most relevant proposals on extensions for UPC collectives were described in a technical report [6] and in a draft specification [7] elaborated at Michigan Technological University (MTU), where a Reference Implementation of UPC standard collectives was also developed [8]. These documents propose the implementation of several extensions to them using concepts already present in other parallel programming languages and libraries (e.g. MPI [9]), such as the definition of variable-sized data blocks for communications (vector-variant collectives), a simplified interface for communications in the same shared array (in-place collectives), the use of teams (subsets of the threads that execute a UPC program), and also asynchronous data transfers. The use of one-sided communications, that is, communications in a single direction with an active and a passive peer, is proposed as the main basis to implement collectives [10]. Other related proposals are Value-Based Collectives [11], which use single-valued variables, either shared or private, as source and destination of communications.

However, the vast majority of these works on extended UPC collectives represents just a sketch on how these collectives could be implemented. In fact, only the MTU report [6] presents some implementation details and a preliminary benchmarking for a small subset of these functions, whereas many other issues, such as the implementation of teams or in-place operations, are simply mentioned, without further discussion. One main reason for this is the fact that the UPC community has scarcely adopted the use of collectives, because of their limitations and generally poor efficiency. Therefore, the main research efforts on UPC collectives have traditionally focused on performance analysis [12] and the proposal of potential performance optimizations to the standard collectives library [13], whereas the improvement of programmability for collectives has been considered as a secondary issue. Even though UPC is regarded as a good language for programmability, there are still very few works that focus on evaluating or improving this feature: the most relevant ones are an early study of UPC programmability in terms of Source Lines of Code (SLOCs) [14], and two recent works on implementing strided collectives for their use on different computational kernels [15] and tuning collective functions at low level in order to build a more efficient library [16]. This latter work also comments some hints about the use of routines to put a subset of threads in a team according to the affinity of the data involved in the collective call, similarly to MPI communicators. Nevertheless, few implementation details are given on this issue, because it focuses mainly on performance testing with different systems.

Thus, in order to improve the work on programmability for UPC, the current paper presents an extensive set of the collectives discussed in previous proposals (in-place, vector-variant, team-based), and also new functions (get-put-priv) whose features are also combined with the previous ones (e.g., get-put-priv vector-variant collectives). As a result, this library covers practically all the demands of the collective communications required by UPC programmers on their applications. Moreover, an implementation of UPC teams at library level has been developed to support team-based collectives, thus alleviating its lack in the standard UPC language. The main contributions of this work are not only the definition of the interfaces and the operation of each function, but also (1) the implementation of the one-sided primitives using standard UPC constructs, thus making the library completely portable to any compliant UPC compiler and runtime, and (2) the design decisions taken to implement some operations, which are evaluated in terms of performance and programmability through its application in the development of several UPC codes.

## 3 Design and Implementation of Extended Collective Primitives

The functions included in the new extended UPC collectives library are distributed in four different groups, each of them focusing on overcoming a specific limitation of the standard UPC collectives. Figure 1 presents four significant limitations in the standard collective framework (left-hand side), and the groups of implemented collective functions (right-hand side) that address the corresponding issue. Thus, these groups of collectives provide different programmability improvements:

- *In-place collectives*: overcome the need of using different buffers for source and destination data.
- *Vector-variant collectives*: allow the communication of a varying data size per thread.
- *Team-based collectives*: execute collective operations within a particular team of

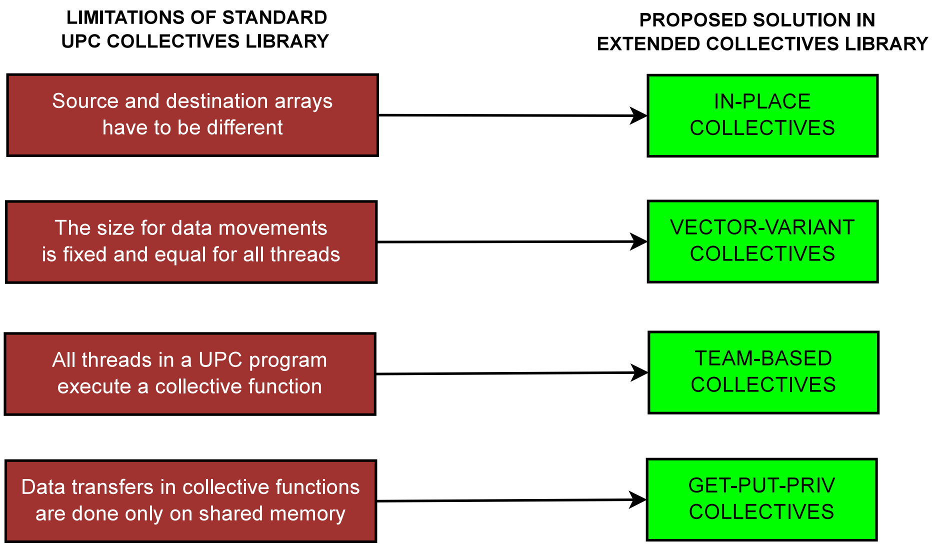

LIMITATIONS OF STANDARD UPC COLLECTIVES LIBRARY
PROPOSED SOLUTION IN EXTENDED COLLECTIVES LIBRARY
Source and destination arrays have to be different
IN-PLACE COLLECTIVES
The size for data movements is fixed and equal for all threads
VECTOR-VARIANT COLLECTIVES
All threads in a UPC program execute a collective function
TEAM-BASED COLLECTIVES
Data transfers in collective functions are done only on shared memory
GET-PUT-PRIV COLLECTIVES

Fig. 1. Scheme of the Extended Collectives library

threads.

- *Get-Put-Priv collectives*: skip the limitation of using shared memory addresses as function parameters.

The arguments of all the extended collectives are always derived from their standard UPC collective counterparts, and new parameters are added in order to implement the required extended functionality. Additionally, several versions of many extended collectives are also included in this library. Every version adds a specific feature to the associated extended collective, in order to allow more flexibility. One common version for the whole set of implemented collectives consists in the use of an additional parameter to specify the thread that will act as root for the collective, which is referred as *rooted version* in this work.

The operations implemented are those present in the standard UPC collective library, namely broadcast, scatter, gather, allgather, exchange, permute, reduce and prefix-reduce, alongside one more function: allreduce. The next subsections detail each group of extended collectives with its associated additional versions.

## 3.1 In-place Collectives

These collectives use only one argument to specify both the source and destination of the data involved in the operation, in order to facilitate communications within the same array. The rest of parameters to these functions are the same as in the standard counterparts. Listing 1 presents the signatures of two representative in-place collectives (broadcast and reduce).

The codes for these collectives are highly dependent on the operation performed. For instance, an in-place broadcast presents a straightforward implementation, because in this case the source data is moved directly to different locations in remote thread's memory. This situation is analogous for scatter and gather. However, other implemented collectives (e.g., permute, exchange, allgather) have to operate on the source data locations, therefore it is necessary to implement different levels of synchronization to execute the collective correctly.

```
void upc_all_broadcast_in_place (
        shared void *srcdst , size_t nbytes ,
        upc_flag_t sync_mode
);
void upc_all_reduceD_in_place (
        shared void *srcdst , upc_op_t op ,
        size_t nelems , size_t blk_size ,
        double (*func)(double,double) ,
        upc_flag_t sync_mode
);
```

List. 1. Signature of representative in-place collectives

Regarding the permute collective, its algorithm has been implemented using an auxiliary private array in each thread to perform the data exchanges between them. The reduce, prefix reduce and allreduce collectives also present a behavior similar to their standard counterparts: first, in parallel, each thread performs the reduction operation on its data; after that, all threads are synchronized, and finally the partial results are sent among threads to produce the final result. Regarding the allgather and exchange in-place algorithms, some optimizations have been introduced to minimize the number of communications, thus favoring a more efficient processing.

Figure 2 presents the data movements implemented for the in-place allgather collective using 3 threads. Here the numbering at each arrow indicates the order in which each communication is performed, thus the arrows with equal numbering represent parallel data movements. First, each thread moves its source data chunk to its corresponding final location within the shared memory space. Then, a synchronization is needed to make sure that all threads have performed this first copy, otherwise source data could be overwritten. Finally, each thread sends its corresponding chunk to the rest of the threads without further synchronizations.

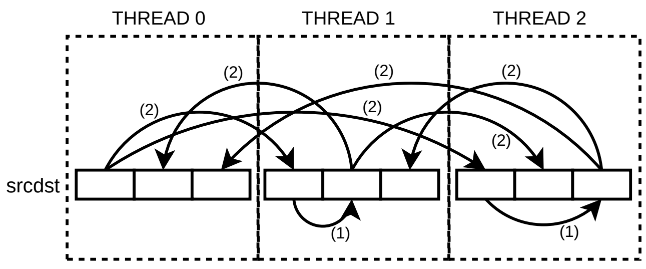


Fig. 2. Communications for `upc_all_gather_all_in_place` (3 threads)

The in-place exchange algorithm is pre-

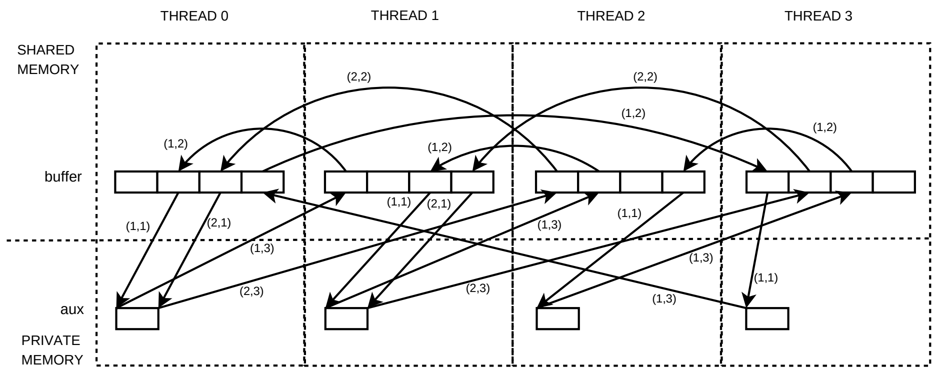


Fig. 3. Communications for `upc_all_exchange_in_place` (4 threads)

sented in Figure 3 using four threads. It uses a concatenation-like procedure [17], including additional logic that avoids the overwriting of source data and also balances the workload among threads. Moreover, it only needs a single private array of *nbytes* (being this the value passed as parameter to the collective) as extra memory space. This algorithm is performed, at most, in (*THREADS/2*) stages. Each stage always consists of three steps: (1) a piece of local source data is moved from shared memory to an auxiliary private array, (2) the corresponding remote data is copied to that source location, and (3) the private memory copy of the source data is moved to the remote location used in the previous step. In the first stage, each thread copies data from/to its right neighbor (that is, thread *i* and thread *(i+1)%THREADS* are associated), and in the next stages data exchanges continue with the following neighbors (for thread *i*, it would be thread *(i+s)%THREADS*, where *s* is the number of stage). In order to avoid data dependencies, all threads are synchronized after the execution of each stage. When the number of threads is even, the last stage only needs to be performed by a half of the threads (in this implementation, the threads with an identifier less than *THREADS/2*). The arrow numbering for Figure 3 consists of two values, that indicate the number of stage (left) and step (right) in which the data movement is performed. No synchronizations are required between steps or stages, and in the ideal scenario all communications with the same numbering would be executed in parallel.

Additionally, a rooted version has been implemented for the four in-place collectives where it is possible to define a root thread (broadcast, scatter, gather and reduce). Listing 2 presents the signature of the rooted in-place broadcast collective as an example of them. As commented before, these functions take the same arguments as the corresponding in-place collectives, but including the identifier of the root thread (parameter *rth*). Their internal implementation is also very similar to their associated extended function, but changing the source (broadcast, scatter) or the destination (gather, reduce) address according to the given root thread.

```
void upc_all_broadcast_rooted_in_place (
        shared void *srcdst , size_t nbytes ,
        int root , upc_flag_t sync_mode
);
```

List. 2. Signature of a representative rooted in-place collective

The main advantage of in-place collectives is that they operate on a single array, and thus the memory allocation for the destination array is not required. To illustrate their use, a common routine for time measuring is presented in Listing 3. The use of the selected extended collective (`upc_all_reduceD_all_in_place`) returns the final result in the shared memory associated to all threads using only a shared array of *THREADS* elements.

```
shared double times[THREADS];
...
times[MYTHREAD] -= getCurrentTime();
...
times[MYTHREAD] += getCurrentTime();
upc_all_reduceD_all_in_place(times, UPC_MAX,
        THREADS, 1, NULL, sync_mode);
```

List. 3. Time measuring routine using in-place collectives

## 3.2 Vector-variant Collectives

This set of collectives allows the definition of a variant number of elements for communications in each thread. The library includes vector-variant implementations for the eight standard UPC collectives plus allreduce. Additionally, a generalized memory copy function named `upc_all_vector_copy` is also provided. This function performs a custom number of data movements between any pair of threads, allowing the user to specify the displacement and number of elements for each communication, thus supporting a high level of flexibility in the library. The signatures of four representative vector-variant collectives are included in Listing 4.

The size of the data type of the elements in the arrays (*typesize*) is used as an argument to these collectives, as it allows the definition of the number of elements in each communication instead of the data size, thus making the code more intuitive. However, the number of arguments used to implement these collectives may vary for each function, because of their different data requirements to compute the position of every memory chunk involved in communications. For example, the broadcast collective requires the definition of the block size of the destination array, whereas scatter and gather also need the value of the block size of

```
void upc_all_broadcast_v (
  shared void *dst, shared const void *src, shared int *ddisp, int nelems,
  size_t dst_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_scatter_v (
  shared void *dst, shared const void *src, shared int *ddisp, shared int *nelems,
  size_t dst_blk, size_t src_blk, size_t typesize, upc_flag_t sync_mode
);
void upc_all_reduceI_v (
  shared void *dst, shared const void *src, upc_op_t op, shared int *sdisp,
  shared int *ndisp, int nchunks, size_t blk_size, int (*func)(int,int),
  upc_flag_t sync_mode
);
void upc_all_vector_copy (
  shared void *dst, shared const void *src, shared int *ddisp, shared int *sdisp,
  shared int *nelems, int nchunks, size_t dst_blk, size_t src_blk,
  size_t typesize, upc_flag_t sync_mode
);
```

List. 4. Signatures of representative vector-variant collectives

the source array (*src_blk* in Listing 4). The additional parameters related to displacements on source/destination arrays (*sdisp* and *ddisp*, respectively) and number of elements per thread (*nelems*) are defined in the interface as shared variables in order to favor a global view of the parameters, but the access to these variables is internally privatized for each thread to avoid performance issues.

The vector-variant implementations of the standard data-movement collectives (broadcast, scatter, gather, allgather, exchange and permute) follow a similar structure: all the arrays that define the displacements and the number of elements in each communication have *THREADS* elements (except for broadcast, that only needs a scalar value). Nevertheless, the implementation of reduce, prefix-reduce, allreduce and vector-copy is substantially different, because these operations can involve the definition of more than one data movement per thread. In fact, the best option in terms of programmability is to let the user define a custom number of chunks in the source array to execute the collective regardless of their thread affinity, thus this approach has been used for these four collectives. Figure 4 shows the operation of a call to `upc_all_reduceI_v` using three chunks, whose communications associated to the reduced data are labeled with the same number as in the arrays containing displacements (*sdisp*) and elements per chunk (*ndisp*).

Four additional versions of these functions have been included in the library:

- *rooted*: these versions (only available for broadcast, scatter and gather) include the label `_rooted` at the end of the name and an additional integer argument that

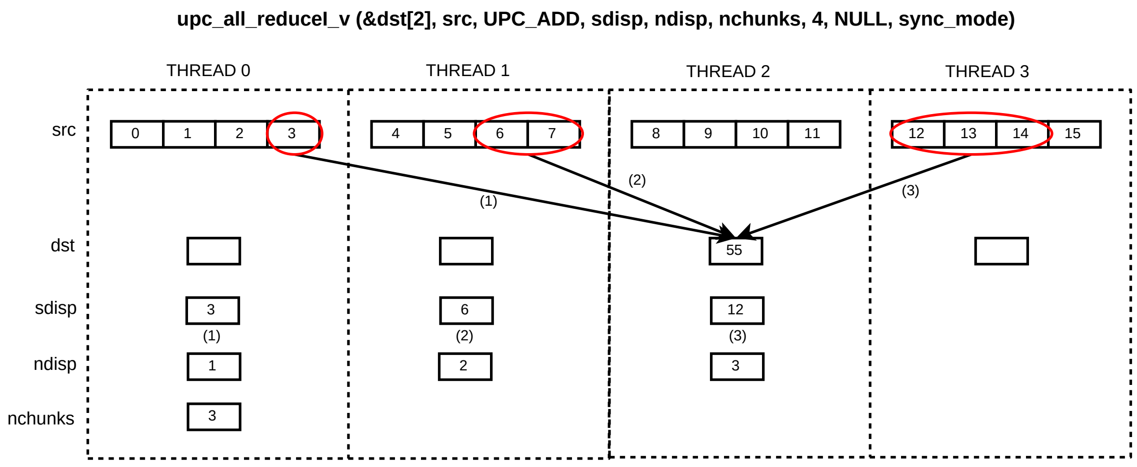


Fig. 4. Communications for `upc_all_reduceI_v` (4 threads)

represents the root thread.

- *local*: they provide the possibility of defining array displacements as relative positions inside a thread, instead of using the default absolute array values. These versions add the label `_local` at the end of their name, and they are available for broadcast, scatter, gather, allgather, permute and exchange.

- *raw*: these functions (labeled with `_raw`) allow the user to define the number of bytes transferred by each thread analogously to the standard collectives, instead of using the number of elements and the element size. Therefore, they use a parameter *shared size_t *nbytes* instead of parameters *nelems* and *typesize*.

- *privparam*: these versions (only available for allgather and exchange) take the parameters of source/destination displacements and number of elements as private variables. These collectives (allgather and exchange) perform multiple accesses to these arrays in order to obtain the source and destination locations for each data chunk, so keeping this data private improves performance and scalability.

Additionally, a *merge* version for the vector-variant exchange has been implemented. The difference lies in the way the elements are gathered by each thread: the `upc_all_exchange_v` collective copies each chunk to the same relative position in the destination array as in the source thread, whereas the `_merge` version puts all chunks in consecutive memory locations. It uses an additional array argument with *THREADS* positions that indicates the location of the first element copied from thread 0 to each thread, and the rest of the elements are copied consecutively using that value as reference.

An example of use of the vector-variant collectives is the copy of an upper triangular matrix from vector *A* to *B*, which is implemented in Listing 5.

```
shared [N*N/THREADS] int A[N*N], B[N*N];
shared int sdisp[N], ddisp[N], nelems[N];
...
// Initialization of shared argument arrays
upc_forall (i=0; i<N; i++; &sdisp[i]) {
    sdisp[i]=i*N+i;  ddisp[i]=i*N+i;
    nelems[i]=N-i;
}
upc_barrier;
upc_all_vector_copy(B, A, ddisp, sdisp,
    nelems, N, N*N/THREADS, N*N/THREADS,
    sizeof(int), sync_mode);
```

List. 5. Copy of a triangular matrix using vector-variant collectives

Here the initialization consists in setting the displacement arrays for the source and destination addresses, as well as the number of elements for each of the N rows (chunks) of the matrices. After that, a call to `upc_all_vector_copy` is enough to perform all necessary data movements.

### 3.3 Team-based Collectives

These collectives are based on teams, which are subsets of the UPC threads running an application. The use of teams has been addressed by the UPC community, mainly focusing on an implementation at language level [15][7], although the use of MPI has also been suggested [18]. However, up to now no standard UPC team implementation has been defined. In order to overcome this limitation and support the use of teams in collectives, this section presents a library-based support for UPC teams, which uses a C structure to define the necessary variables to implement them.

```
struct teamContent {
  shared t_boolean
    *isThreadInTeam;     // THREADS elements
  shared int *numthreads;
  shared int *counterBarrier; // 2 elements
  shared int *flagBarrier;    // 2 elements
  upc_lock_t *lockTeam;
  shared void *shared
    *pointerArg;         // THREADS elements
};
typedef struct teamContent team;
```

List. 6. Structure for UPC teams support

Listing 6 presents the struct data type that defines a team. It uses an array of *THREADS* boolean elements (*isThreadInTeam*) that indicate whether a thread is included in the team or not. A team identifier (tid) for each thread is assigned in increasing order of the UPC thread identifier. The variable *numthreads* indicates the number of threads in the team. The *counterBarrier* and *flagBarrier* arrays include two thread counters and two flags, respectively, that are used as auxiliary variables for the implementation of synchronization barriers through an active-wait algorithm. The lock variable *lockTeam* is used to implement atomic operations in the team (e.g., in the execution of team barriers or team management operations, such as thread insertions). Finally, the *pointerArg* variable is an auxiliary array used for memory allocation within the team. Using these variables, our implementation is able to provide full support for team-based collectives.

The team-based collectives have been implemented independently from the underlying team library, by dealing with teams through different management functions for basic team operations, such as barriers or memory allocation routines. Using these functions as interface, the separation between collectives and team libraries is achieved. Thus, in order to implement each team-based collective, only an additional team variable is added to the arguments of the standard collective counterpart.

```
void upc_all_gather_team (
 shared void *dst , shared const void *src ,
 size_t nbytes , team t , upc_flag_t sync_mode
);
```

List. 7. Signature of a team-based collective

Listing 7 shows the signature of the team-based gather collective as an example. All team-based collectives (labeled with `_team`) present the same arguments as their standard counterparts, plus the private variable that represents the team description. This team-based implementation interprets the argument that represents the size of communications (*nbytes*) as the total amount of data that is transferred by all threads in the team. Thus, *nbytes/numthreads* bytes are transferred by each thread, and the first chunk goes to the thread with tid 0. Only the members of the team can invoke these functions.

Additionally, the scatter, gather, allgather and exchange collectives admit the implementation of a filter version (labeled with `_team_allthr`). It has the same behavior as the standard counterpart, but here the team prevents the threads that are not included in it from executing the operation. Therefore, this version does not consider team identifiers for communication, and the argument *nbytes* is considered as the amount of data transferred by each thread. Nevertheless, all these filter operations maintain the same type of arguments as the corresponding `_team` collective.

Figure 5 illustrates the behavior of both types of team-based collectives with a scatter collective executed in a program with 4 threads. Both functions have a source array of 12 KB, which is distributed according to the definition of each collective among the three threads (0, 1 and 3, that have tids 0, 1 and 2, respectively) included in team *t*. The piece of code in Listing 8 presents an example of the use of these collectives: two teams execute the same function (computation of Pi using the Monte Carlo method) in parallel in the same UPC program.

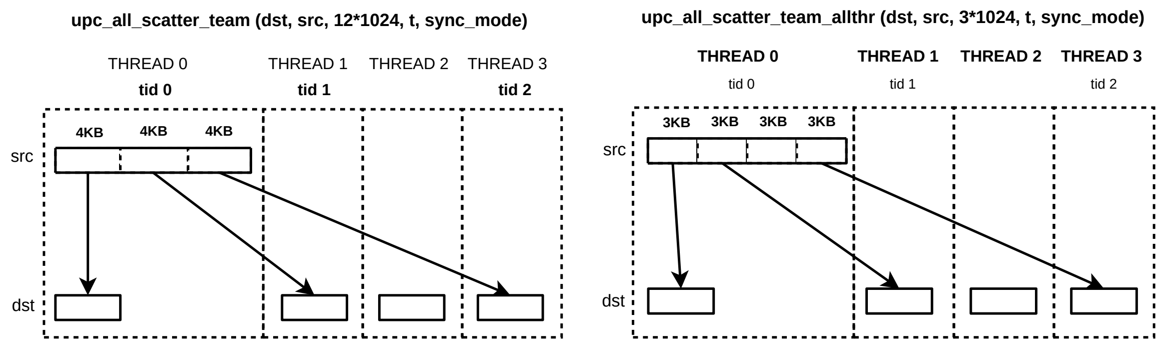


Fig. 5. Communications for team-based scatter operations (4 threads)

```
void computePiMontecarlo(int trials, team t,
  shared double *estimation) {
    shared int global_hits[THREADS];
    shared int local_hits[THREADS];
    double piEstimation;
    // Filter all threads that are not
    // included in the team
    if (!isThreadInTeam(MYTHREAD, t)) return;
    // Auxiliary function
    int nth = getNumThreads(t);
    // Compute local hits, put result
    // in local_hits[MYTHREAD]
    ...
    upc_all_reduceI_all_team(global_hits,
      local_hits, UPC_ADD, nth, 1, NULL, t,
      sync_mode);
    // Compute pi estimation
    ...
    *estimation = piEstimation;
    return;
}

int main() {
    team t1, t2;
    shared double est1, est2;
    // Initialize variables and create
    // teams (disjoint sets)
    ...
    // Execute tasks
    computePiMontecarlo(trials1, t1, &est1);
    computePiMontecarlo(trials2, t2, &est2);
    if (MYTHREAD == 0) {
      printf("Estimation: %lf\n",
        (est1*trials1+est2*trials2)/
         (trials1+trials2));
    }
}
```

List. 8. Computation of Pi using team-based collectives

This code has been implemented using team-based collectives, in which both teams execute their tasks independently. This type of execution is highly interesting for heterogeneous architectures, which require custom support for every kind of resource: a team could group different processors (even hardware accelerators such as GPUs) according to their features, thus helping handle workload imbalance.

## 3.4 Get-Put-Priv Collectives

These collectives allow to use a private array as source and/or destination parameter in all the previous extended collectives (in-place, vector-variant and team-based, alongside their own additional versions), and also in the standard collectives plus allreduce. Thus, they represent the largest subset of collectives included in the extended library. These collectives can be classified in three subgroups:

- *Get collectives*: shared source and private destination

- *Put collectives*: private source and shared destination
- *Priv collectives*: the source and destination are both private

In-place collectives only have the *priv* version as they have the same array as source and destination. For illustrative purposes, Listing 9 shows the signatures of the allgather get-put-priv versions.

```
void upc_all_gather_all_get (
            void *dst, shared const void *src,
            size_t nbytes, upc_flag_t sync_mode
);
void upc_all_gather_all_put (
            shared void *dst, const void *src,
            size_t nbytes, upc_flag_t sync_mode
);
void upc_all_gather_all_priv (
            void *dst, const void *src,
            size_t nbytes, upc_flag_t sync_mode
);
```

List. 9. Signatures of representative get-put-priv collectives

The algorithms implemented in these collectives minimize the number of communications, avoiding unnecessary remote data transfers and maximizing parallel processing among threads using as few synchronization points as possible. In general, only the *priv* collectives require the allocation of additional shared memory to allow the data transfers, at most the same size as the communications performed. As an example, Figure 6 shows the *priv* version of a broadcast that uses 4 threads. First, thread 0 stores its data on an auxiliary buffer in shared memory, then a synchronization is necessary to make sure that the data has been made available, and finally all threads copy the data to the final location. Thread 0 only needs to move data inside its private memory space, implementing such data copy with the `memcpy` system library routine.

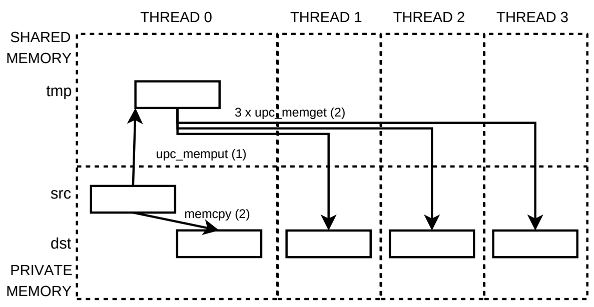


Fig. 6. Data movements for `upc_all_broadcast_priv` (4 threads)

The usefulness of these functions can be assessed in a parallel image filtering algorithm, presented in Listing 10. The input data, a matrix of $N$x$N$ elements, is stored in the private memory of thread 0 (in matrix *img*), thus the private versions of the standard scatter and in-place broadcast perform the necessary data movements to distribute the workload to the private memory spaces of all threads (in matrix *aux*). The filtering algorithm is executed on private memory in order to efficiently exploit data locality, because the operation with private memory is more efficient than dealing with shared memory [4]. Finally, the private version of the standard gather collective returns the final result in the private memory of thread 0 (in matrix *filteredImg*).

```
// Initialize 'img' as the source N*N image,
//   'aux' as an auxiliary array of
//   (N*N/THREADS) elements on each thread
//    and 'filter' as a 3x3  matrix
// All private variables, init. on thread 0

upc_all_scatter_priv(aux, img,
  (N*N/THREADS)*sizeof(double), sync_mode);
upc_all_broadcast_in_place_priv(filter,
  3*3*sizeof(double), sync_mode);

filterMatrix(aux, filter, N, N, 3, 3);

upc_all_gather_priv(filteredImg, aux,
  (N*N/THREADS)*sizeof(double), sync_mode);
```

List. 10. Image filtering using get-put-priv collectives

## 4 Use of Extended Collectives: Case Studies

The extended collectives improve programmability for a wide variety of problems by reducing the number of SLOCs and favoring code expressiveness. Nevertheless, their adoption requires that their performance should also be competitive when compared to their equivalent implementation in standard UPC. Thus, this section analyzes the impact of introducing extended collectives on different codes, justifying the benefits in terms of programmability obtained from their use. Next section presents the performance of the implemented kernels.

The selected codes are four kernels (dense and sparse matrix multiplication, Integer Sort and 3D Fast Fourier Transform) and two UPC MapReduce applications.

### 4.1 Dense Matrix Multiplication Kernel

Listing 11 presents an optimized standard UPC code that multiplies two dense $N$x$N$ matrices ($\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$). The source matrices $A$ and $B$ are stored in the private memory of thread 0. All matrices are stored in a linearized form (1D) according to the UPC standard. To parallelize the operation, matrix $A$ is split in chunks, distributed evenly among all threads and stored in their private memory spaces together with a copy of matrix $B$, which is broadcast from thread 0. After the local multiplication, the result matrix is finally gathered in thread 0.

All the data movements between threads are performed using a significant number of one-sided communications with memory copy functions, because of the lack of collectives support for operating with different private memory spaces as source and destination addresses, and thus auxiliary shared arrays (*temp_A*, *temp_B* and *temp_C*) and synchronizations (three calls to `upc_barrier`) are necessary to perform the data transfers between threads.

The use of extended collective functions can reduce significantly the complexity of the UPC implementation of this kernel, as presented in Listing 12. Here all the data movements associated to the source and destination arrays are implemented within the extended collective function with private arguments.

```
#define chunk_size N*N/THREADS;
double *A, B[N*N], *C;
double local_A[chunk_size],
   local_C[chunk_size];
shared [chunk_size] double temp_A[N*N],
   temp_C[N*N];
shared [] double temp_B[N*N];
if (MYTHREAD == 0) {
  // Allocate and initialize arrays A, B, C
  ...
  memcpy(local_A, A,
     chunk_size*sizeof(double));
  for (i=1; i<THREADS; i++) {
     upc_memput(&temp_A[i*chunk_size],
        &A[i*chunk_size],
        chunk_size*sizeof(double));
  }
}
upc_barrier;
if (MYTHREAD != 0) {
   upc_memget(local_A,
      &temp_A[MYTHREAD*chunk_size],
      chunk_size*sizeof(double));
}
if (MYTHREAD == 0) {
   upc_memput(temp_B, B,
      N*N*sizeof(double));
}
upc_barrier;
if (MYTHREAD != 0) {
   upc_memget(B, temp_B,
      N*N*sizeof(double));
}

computeSubmatrix(local_A, B, local_C,
   N/THREADS, N, N);

if (MYTHREAD != 0) {
   upc_memput(&temp_C[MYTHREAD*chunk_size],
      local_C, chunk_size*sizeof(double));
}
upc_barrier;
if (MYTHREAD == 0) {
   memcpy(C, local_C,
      chunk_size*sizeof(double));
   for (i=1; i<THREADS; i++) {
      upc_memget(&C[i*chunk_size],
         &temp_C[i*chunk_size],
         chunk_size*sizeof(double));
    }
}
```

List. 11. Original UPC dense matrix multiplication code

Matrix $A$ is evenly distributed to all threads using a scatter *priv* collective, whereas an in-place broadcast transfers the whole matrix $B$ to all threads. Finally, the gathering of the result matrix $C$ is performed by a gather *priv* collective. It is important to note that the programmer does not need to deal with any temporary buffer to perform the communications, as the extended collectives handle the auxiliary memory space transparently to the user. Therefore, extended collectives allow the parallelization of this code without requiring the user to deal with shared memory addresses or temporary buffers, and they even take advantage transparently of efficient communication algorithms for data transfers.

```
double local_A[chunk_size],
    local_C[chunk_size];
upc_all_scatter_priv(local_A, A,
  (N*N/THREADS)*sizeof(double), sync_mode);
upc_all_broadcast_in_place_priv(B,
   N*N*sizeof(double), sync_mode);

computeSubmatrix(local_A, B, local_C,
   N*N/THREADS, N, N);

upc_all_gather_priv(C, local_C,
  (N*N/THREADS)*sizeof(double), sync_mode);
```

List. 12. UPC dense matrix multiplication code with extended collectives

### 4.2 Sparse Matrix Multiplication Kernel

This kernel performs the multiplication of a sparse matrix (stored in Compressed Sparse Row -CSR- format) by a dense matrix. Here

```
// Initialize variables: 'val', 'col_ind', 'row_ptr'(values, column index and row pointer
//  arrays in CSR format), 'B' ('k'*'n' dense source matrix), 'C', 'C_dist' (final 'm'*'n'
//  and partial result matrices), 'nz' (number of non-zero values), 'disp...' (some
//  displacement vectors), 'nelems', 'nrows' (element size parameters)
...
upc_all_scatter_v_priv(val_dist, val, disp, nelems, nz, sizeof(double), sync_mode);

upc_all_scatter_v_priv(col_ind_dist, col_ind, disp, nelems, nz, sizeof(int), sync_mode);

upc_all_vector_copy_priv(row_ptr_dist, row_ptr, disp_rows_dst, disp_rows_src,
 nrows, THREADS, m+1, m+1, sizeof(int), sync_mode);

upc_all_broadcast_in_place_priv(B,  k*n*sizeof(double), sync_mode);

// Modify 'nrows' to allow separate calls to the multiplication algorithm on each thread
...
computeMMSparse(val_dist, col_ind_dist, row_ptr_dist, B, C_dist, nrows[MYTHREAD], k, n);

// Modify variables to gather the computed submatrices
...
upc_all_gather_v_priv(C, C_dist, disp, numvalues, m*n, sizeof(double), sync_mode);
```

List. 13. UPC sparse matrix multiplication code with extended collectives

the work distribution is done by subdividing the different arrays that define the compressed sparse matrix (values, column index and row pointer), selecting the number of rows that each thread should process to obtain balanced workloads. As the number of elements in each array can be different for each thread, the scatter and gather operations are performed using vector-variant collectives. Listing 13 shows the most relevant parts of the sparse matrix multiplication code that use extended collectives. The multiplication routine is performed by calling the sequential multiplication routine separately on each thread, which involves some small modifications to the CSR arrays (both for standard UPC and using the extended library) in order to use this function, considering each sparse matrix chunk as an independent matrix. The equivalent standard code requires many more SLOCs in order to support the vector-variant data transfers using loops and array subscriptions, which are here avoided by using collectives.

## 4.3 Integer Sort Kernel

The Integer Sort (IS) kernel from the NAS Parallel Benchmark (NPB) suite for UPC [19] has been traditionally used in UPC benchmarking [20][21][22]. The core of the kernel is the `rank` function, which performs the bucket sort of a set of integer keys, and a piece of its code consists in redistributing the keys by means of an all-to-all operation with data chunks of different sizes.

Listing 14 presents the original implementation of the data exchange performed in the `rank` function of IS. The keys are stored in a shared array (*key_buff1_shd*), and the information about the data chunks that correspond to each thread is stored in a private array of THREADS structures (*infos*). Each structure in this auxiliary array contains the number of elements and the offset of the first element for each data chunk. The chunks received by a thread after the all-to-all communication are stored consecutively in its private memory (array *key_buff2*).

```
// (Init. of variables would be here)
upc_barrier;
for (i=0; i<THREADS; i++) {
  upc_memget(&infos[i],
   &send_infos_shd[MYTHREAD][i],
   sizeof(send_info));
}
for (i=0; i<THREADS; i++) {
 if (i == MYTHREAD)
  memcpy(key_buff2 + total_displ,
   key_buff1 + infos[i].displ,
   infos[i].count * sizeof(INT_TYPE));
 else
  upc_memget(key_buff2 + total_displ,
   key_buff1_shd+i+infos[i].displ*THREADS,
   infos[i].count * sizeof(INT_TYPE));
 total_displ += infos[i].count;
}
upc_barrier;
```

List. 14. Original UPC code in Integer Sort

Listing 15 shows the implementation of the all-to-all communications of `rank` in IS using the *get* version of the `upc_all_exchange_v_merge_local` extended collective. As the displacements used are relative array positions, a *local* version is required, and the *get* variant is necessary to use the same source and destination arrays as in the original code. However, the extended collective handles the displacements (*send_displ_shd*) and element counts (*send_count_shd*) separately, thus the array of structs is split in two separate shared arrays. Additionally, a displacement vector (*disp*) is required by the collective call to indicate the offset for the first element received at the destination (set to *0* for all threads in this code) as well as the block size of the source array (*SIZE_OF_BUFFERS*), which is a predefined constant in the code. For a better understanding of this code, the use of `upc_all_exchange_v_merge_local` is illustrated in Figure 7 using a simple scenario with two threads.

```
// Initialization of variables
...
upc_all_exchange_v_merge_local_get(
  key_buff2, key_buff1_shd, send_displ_shd,
  send_count_shd, disp, SIZE_OF_BUFFERS,
  SIZE_OF_BUFFERS, sizeof(int), sync_mode);
```

List. 15. UPC code in Integer Sort using extended collectives

## 4.4 3D Fast Fourier Transform Kernel

The 3D Fast Fourier Transform (FFT) is another kernel from the UPC NPB suite. It computes the Fourier transform algorithm on a three-dimensional matrix using different domain decompositions, representing a widely extended code in scientific and engineering computing. This UPC kernel has been derived from the OpenMP FFT NPB implementation, and

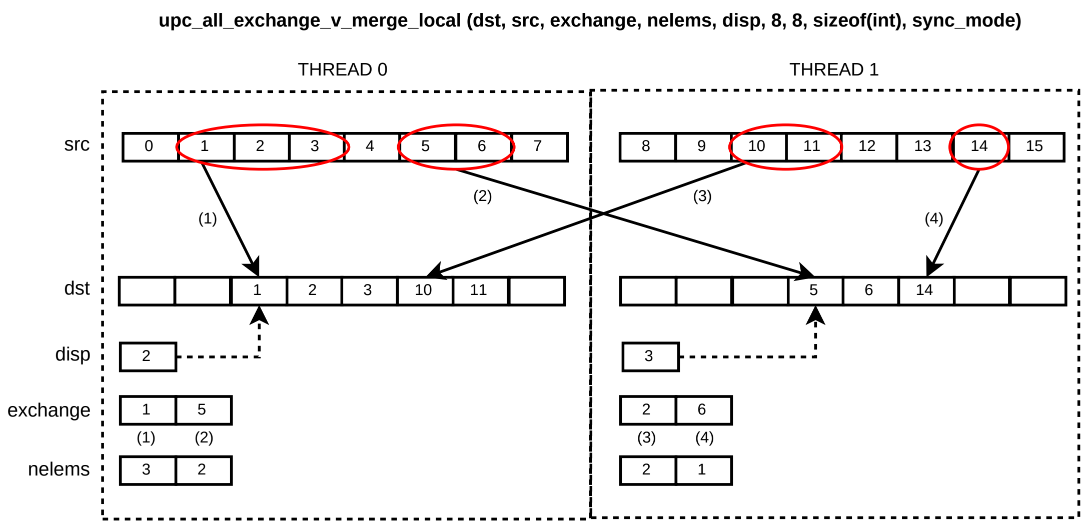


Fig. 7. Communications for `upc_all_exchange_v_merge_local` (2 threads)

thus includes some significant changes on the variables with respect to the original Fortran code, in order to allow a better adaptation to its syntax (e.g. user-defined data types are used to facilitate the storage of complex values on each thread). The main computations of this kernel are performed in private memory using an array of structs (*u0*) that stores the initial conditions of the system in a linearized way analogously to the matrices of Section 4.1, and two working arrays (*u1* and *u2*) that assist the computation of the Fourier transform by storing intermediate calculations. The key part of this code is the computation of the transpose of the linearized matrix (stored in array *u1*) in *u2*, which is performed using a heavy all-to-all communication. Listing 16 shows the implementation of the remote communications used in the matrix transposition in the UPC FFT code, using `upc_memget` to obtain the corresponding array chunk from each thread: the data associated to a thread is stored in an array of complex values defined as the member of a struct, which is defined for each thread in an array of shared structs with *THREADS* elements. This technique is used to avoid the definition of an array with a very large block size, which would affect performance.

```
for (i = 0; i < THREADS; i++) {
  upc_memget(
    (dcomplex *)&u2[MYTHREAD].cell[chunk*i],
    &u1[i].cell[chunk*MYTHREAD],
    sizeof(dcomplex) * chunk );
}
```

List. 16. Original UPC code in 3D FFT

Here the introduction of an extended exchange collective represents a better solution to implement all-to-all communications. However, the definition of the array of shared structs to store the data does not allow a direct application of this collective, as the source data is split. Therefore, some small changes are performed: the shared source array is referenced by a private pointer for each thread, and the *priv* variant of the exchange collective is applied here to obtain higher performance, as stated in Listing 17. Considering that the copy from *u1* to *u2* in this transposition is performed just to simplify the code, a second in-place solution is proposed, in which the all-to-all communication is performed on the same array *u1*. This approach only implies that the results of the communication are stored in *u1* instead of *u2*, which does not affect the final results of the FFT because the source data in *u1* is not reused after this communication. Both implementations with extended collectives, alongside with the standard approach, will be evaluated and tested in Section 5.1.

```
// First approach: u1 as src and u2 as dst
upc_all_exchange_priv(
    (dcomplex *)my_u2->cell,
    (dcomplex *)my_u1->cell,
    sizeof(dcomplex) * chunk, sync_mode);
// Second approach: in-place comms in u1
upc_all_exchange_in_place_priv(
    (dcomplex *)my_u1->cell,
    sizeof(dcomplex) * chunk, sync_mode);
```

List. 17. UPC code in 3D Fast Fourier Transform using extended collectives

## 4.5 UPC MapReduce Framework

MapReduce [23] is an emerging programming model for coarse-grain parallelism. It is based on the application of a *map* function to each element of an input data set, which generates another set of intermediate elements that are combined using a *reduce* operation to generate a final output. MapReduce implementations are typically written using object-oriented languages, such as Java and C++, although this programming model can also take advantage of the PGAS features of UPC on multicore clusters. Thus, a UPC MapReduce framework has been implemented [24].

The proposed framework consists of two template functions: (1) `ApplyMap`, which generates a list of intermediate values according to a list of input elements and a sequential *map* function, and (2) `ApplyReduce`, which merges the intermediate values from all threads according to a sequential *reduce* operation. Listing 18 presents the signatures of `ApplyMap`, `ApplyReduce` and the two user-defined *map* and *reduce* functions that should be passed as argument to them.

The communications in the UPC MapReduce framework involve all threads, although two facts prevent the use of the standard UPC collectives: (1) UPC MapReduce operates in the private memory space, which cannot be used as parameter to the standard collectives, and (2) the processing of different workloads per thread and a varying number of elements in the reduction phase cannot be handled by standard collectives.

```
void *mapfunc(void *input, void *key,
 void *value);
void *reducefunc(void *key, void *value,
 int nelems, void *result);
int ApplyMap(
 int (*mapfunc)(void *,void *,void *),
 void *inputElems, int nelems,
 int userDefDistrFlag, int algorithm,
 int *weights);
void *ApplyReduce(
 int (*reducefunc)(void*,void*,int,void*),
 int nelems, int gathFlag, int collFlag,
 int sizeKey, int sizeValue);
```

List. 18. Signatures of the basic functions in UPC MapReduce

In order to solve these issues, the UPC MapReduce framework uses extended collectives. The most relevant one is the gathering of intermediate data among all threads in `ApplyReduce`: this is done through a call to a *priv* version of the vector-variant allgather collective (`upc_all_gather_all_v_priv`), in which the source arrays of each thread are the variable-size lists that result from the call to `ApplyMap`. However, if all intermediate elements are not equal in size, the *raw* version of the previous collective has to be used in order to indicate the amount of raw data that is transferred to each thread, whereas the MapReduce framework keeps track of the start position of every intermediate element to reconstruct them and perform the reduction.

The two applications used to evaluate this UPC MapReduce framework are a malware detection code and the computation of a linear regression. The details of both of them will be presented in the following section.

## 5 Performance Evaluation

This section presents a performance analysis of the codes discussed in the previous section on two testbed systems. The first one is the JuRoPa supercomputer (from now on, JRP) at Forschungszentrum Jülich (ranked 63rd in the TOP500 List of June 2012), which consists of 2208 compute nodes, each of them with 2 Intel Xeon X5570 (Nehalem-EP) quad-core processors at 2.93 GHz and 24 GB of DDR3 memory at 1066 MHz, and also InfiniBand QDR HCA with non-blocking Fat Tree topology. The second system is the Finis Terrae supercomputer (from now on, FT) at Galicia Supercomputing Center (CESGA), which consists of 142 HP Integrity RX 7640 nodes, each of them with 8 Montvale Itanium 2 (IA64) dual-core processors at 1.6 GHz, 128 GB of memory and InfiniBand as interconnection network (4X DDR, 16 Gbps of theoretical effective bandwidth). On both systems, the UPC compiler is Berkeley UPC [25] v2.14.2, using Intel icc v11.1 as backend C compiler, and its InfiniBand Verbs conduit for distributed memory communications on InfiniBand. The Intel Math Kernel Library (MKL) v10.2 has also been used in the matrix multiplication kernels.

The experimental results of the evaluated codes have been obtained using two different configurations of number of threads per node: (1) one thread per node, and (2) the maximum number of cores per node in each testbed system, that is, 16 threads for FT and 8 for JRP. Each of the analyzed codes has one implementation using standard UPC operations and an alternative version using the extended collectives library (see Section 4 for further details about its application to these codes). The standard UPC versions of NPB IS and FFT are available at [19], whereas the other codes that use standard UPC operations have been implemented following general guidelines for UPC hand-optimized codes [4].

## 5.1 Evaluation of Numerical Kernels

Figure 8 shows the performance in terms of execution times and GFLOPS for the dense matrix multiplication of two 4480×4480 matrices of double precision floating-point elements, using a standard UPC code (labeled as "Standard UPC") and the extended collectives ("Extended Colls"). Here each UPC thread calls a sequential MKL matrix multiplication function, and all data movements associated to the workload distribution are performed by the collective functions. The results indicate that the use of extended collectives represents the best option for implementing this code in nearly all test cases, and especially when using one thread per node, where the differences are larger.

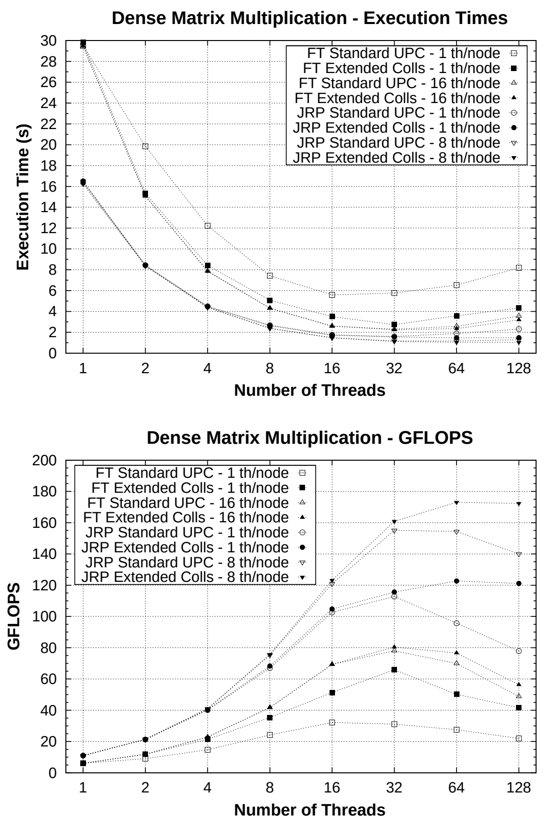
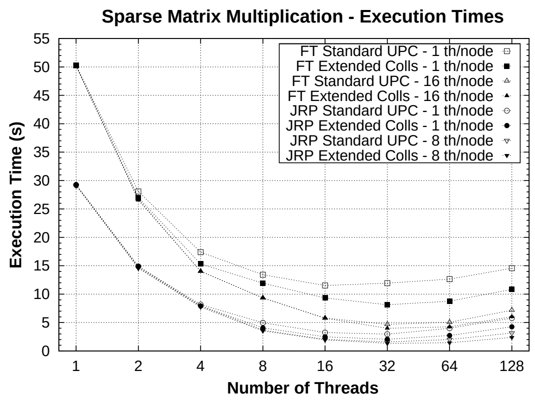


Fig. 8. Performance of dense matrix multiplication (4480×4480)

The reason for this is the internal implementation of the extended collectives, which obtains good performance on hybrid shared/distributed memory systems transparently to the user. These collectives use a flat-tree algorithm for intranode communications and a binomial-tree approach for internode communications, thus taking advantage of both shared memory and high-speed interconnection networks. For the executions using the maximum number of threads per node in each system, the differences between the standard code and the extended collectives are smaller because of the maximization of flat-tree intranode communication. The benefits of their use are noticeable specially when using 32 threads or more in both testbeds. In these cases, some performance benefits are also obtained, mainly in FT, by using message partitioning in chunks in order to optimize the use of caches and the memory consumed (i.e., here the memory buffers are limited to the chunk size).

Figure 9 displays the performance of the sparse matrix multiplication code. The sparse matrix is a symmetric 16614×16614 matrix with 1,091,362 non-zero entries (0.4% of non-zero elements), taken from the set of matrices generated by the FIDAP package in the SPARSKIT Collection [26], and the dense matrix has 16614×4480 double precision elements. Once again, almost all the tests performed show that the extended collectives provide better performance than the standard UPC code, especially as the number of threads increases. This improvement is a bit larger than in the dense case because of the larger amount of communications involved. As a result, the use of more efficient communication algorithms, which take advantage of hybrid shared/distributed memory architectures especially when maximizing intranode communications, is also a key factor in this scenario.



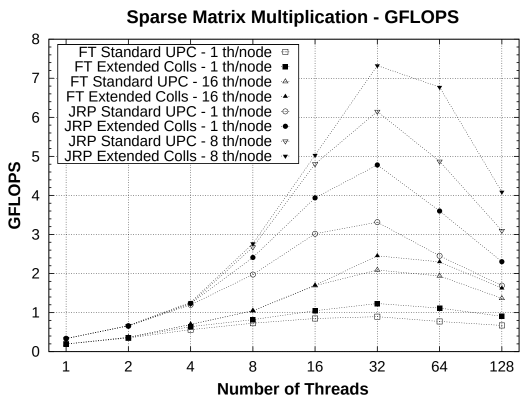


Fig. 9. Performance of sparse matrix multiplication (16614×16614 sparse matrix and 16614×4480 dense matrix)

Here the execution times are higher (the GFLOPS smaller) than for the dense case. This is due to the communications overhead introduced by the data transfers required for this kernel (implemented with the vector-variant collectives in Listing 13), which makes the optimizations depend on a well balanced work distribution in terms of the number and size of the memory chunks processed by each thread.

Figure 10 presents the performance of NPB UPC IS in terms of execution times and millions of operations per second (Mop/s), compared to a version using the extended exchange collective (merge-local-get vector-variant, see Figure 7 for a similar example). For a small number of threads, the standard NPB code obtains similar results to the version using the extended collectives for all configurations, although the difference increases for a larger number of threads, because the binomial-tree algorithms for the extended collective functions help improving the performance. It is also important to note that this kernel takes advantage of intranode communications when using a single node on FT, whereas when using 32 or more threads the best results are always obtained using one thread per node, which is also the most efficient configuration for all tests in JRP. The IS kernel performs several all-to-all operations that exchange a significant number of messages between threads, and here the large memory size of the FT nodes allows a more efficient intra-node data exchange, but the contention of the InfiniBand adapter on inter-node communications with 16 threads per node (in executions with multiple nodes) causes a significant performance drop. Regarding JRP, the use of a maximum of 8 threads per node limits the impact of the network communication bottleneck, allowing better communication scalability.

Regarding the different testbed systems, the results in JRP clearly outperform the results of FT for all the previous kernels, because of the higher processor power and the higher efficiency of the InfiniBand network. As a result of this, the next evaluations (3D FFT and MapReduce applications) will only include for clarity purposes the performance results of the JRP system.

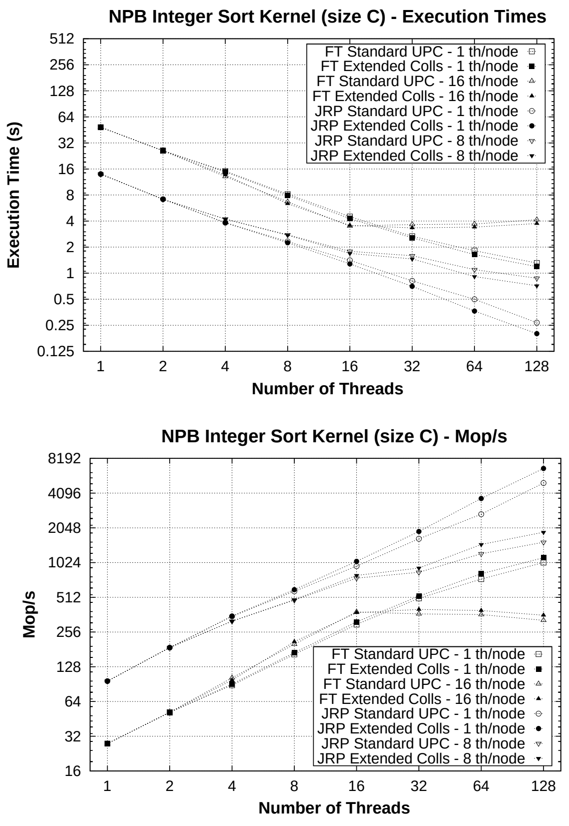


Fig. 10. Performance of NPB Integer Sort

Figures 11 and 12 show the performance results in terms of execution times and billions ($10^9$) of operations per second (Gop/s) of different implementations of the 3D FFT kernel, for 1 and 8 threads per node respectively.

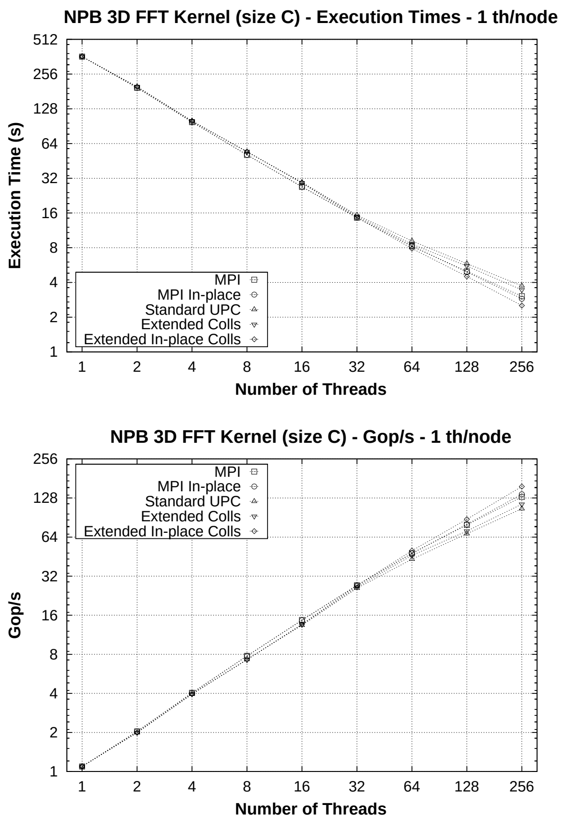


Fig. 11. Performance of NPB 3D FFT in JRP with 1 thread/node

In addition to the UPC codes described in Section 4.4 (the original standard UPC code, the one using a *priv* extended collective and the one using the *priv* in-place function), two more codes that use the C MPI library have been implemented in order to have a traditional parallel programming approach as a reference implementation for the UPC results: one code uses the standard `MPI_Alltoall` collective and the other uses the same collective with the *MPI_IN_PLACE* option, thus the array *u1* is used both as source and destination. These MPI codes follow an analogous approach to the original UPC kernel to show a fair comparison with the UPC codes. The MPI compiler used in JuRoPa is ParaStation MPI 5.0.26-1 [27].

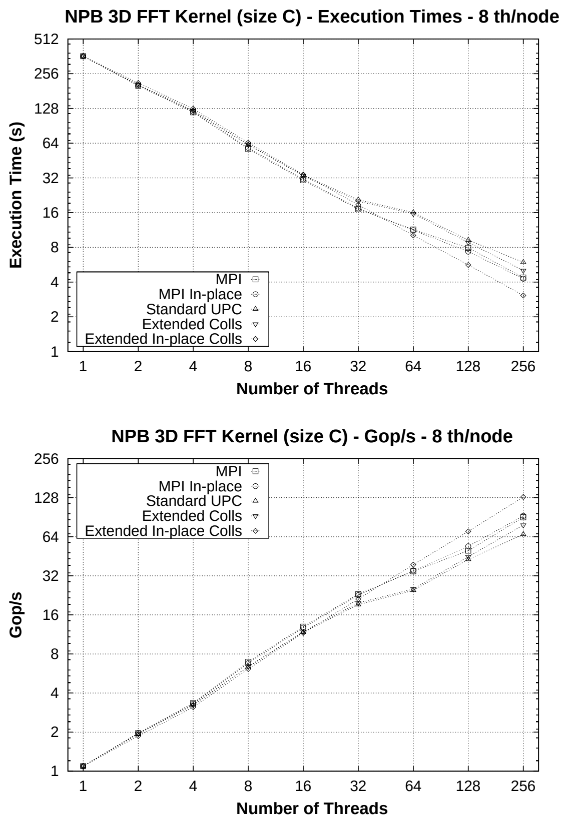


Fig. 12. Performance of NPB 3D FFT in JRP with 8 threads/node

The comparison of these five codes gives out that MPI obtains slightly better performance for up to 32 threads, but from then on the UPC in-place collective clearly presents the most efficient results. The main reason is the use of the algorithm described in Figure 3 of Section 3.1, which is able to maximize the parallelism at a lower computational cost. Moreover, the higher the communication cost of the all-to-all, the better the *priv* in-place collective is able to perform, thus an execution with a high number of threads highlights the benefits of this implementation. Regarding the number of threads per node, the use of all cores in a node is worse than using only one for all UPC and MPI codes, which is due to the contention of InfiniBand communications similarly to the IS case.

### 5.2 Evaluation of the UPC MapReduce Framework

The performance of UPC MapReduce has been assessed using two applications. The first one processes a large corpus of HTML files in order to count the number of occurrences of a set of 256 words associated to malware ("Spam Count" code from now on). The Spam Count code consists of two sequential functions, *map* and *reduce*, that are passed as arguments to `ApplyMap` and `ApplyReduce`, respectively, according to the interfaces shown in Listing 18. The *map* function takes a file name as its first argument and returns the number of words in the malware set that have been read from the files. This number of occurrences are collected by all threads using a *priv* allgather collective in `ApplyReduce` (because all threads present the same amount of counters, one for each detected word). Next, the user-defined *reduce* function is applied to the whole set of target malware words to obtain the total number of occurrences for each one of them.

The second application performs a linear regression, which consists in taking two lists of double-precision paired values of two random variables X and Y, and computing the line that fits best for them ("LinReg" code from now on). The *map* function processes a determined set of elements on each thread, and then the definition of the adjusted line is obtained at the reduce stage. As in the previous case, the number of elements transferred at the reduce stage is the same in all threads: a single correlation value obtained from the results processed by each thread.

Figure 13 presents the performance results (execution times and speedup) for the Spam Count application using 100,000 input files from the Webb Spam Corpus [28] (which has been widely used for similar tests in the area of Information Retrieval), and for LinReg using 1 billion ($10^9$) of paired values for variables X and Y. In order to obtain the best performance for a large number of threads, the reduce stage is here implemented with the *priv* version of the allgather collective (see Section 3.4).

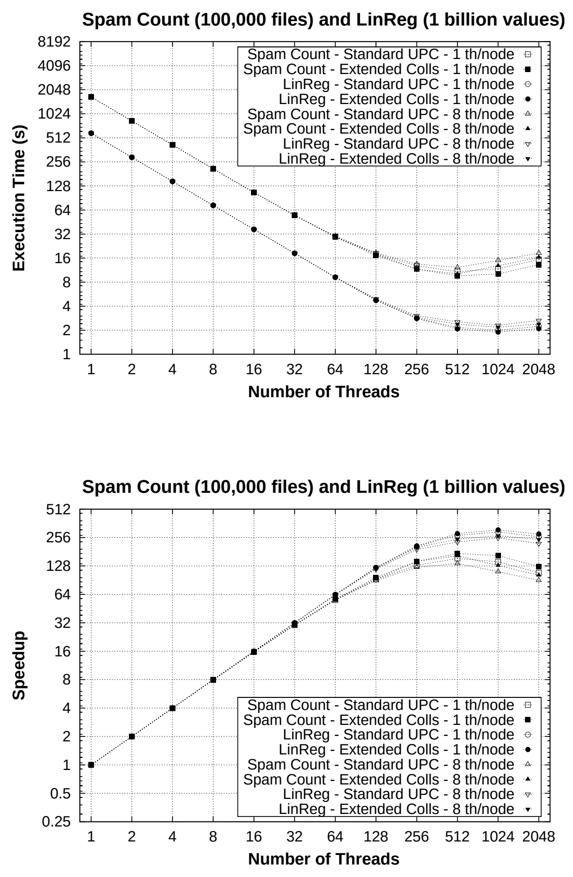


Fig. 13. Performance of UPC MapReduce in JRP for Spam Count and LinReg

The results show that both implementations are able to scale up to a large number of threads, mainly because of the large computational power and high scalability provided by the JRP system, as well as the small weight of the reduce stage in the total execution time of both applications. As a result of this, the differences between the standard communications and the extended collectives are only noticeable for more than 64 threads in Spam Count, and even more threads in LinReg.

In general, the size of communications at the reduce stage for both applications is incremented proportionally with the number of threads, and therefore the speedup tends to decrease for a large number of threads. This happens when the execution time of the reduction stage becomes relevant when compared to the map stage, especially for reduced workloads. Despite these facts, the benefits obtained by the extended allgather collective for a large number of threads are more significant for the Spam Count code than for LinReg, that is, when the amount of data transferred per thread at the reduce stage is larger.

## 6 Conclusions

This paper has presented the design, implementation and evaluation of a library of extended UPC collectives focused on providing higher programmability and overcoming the limitations of the standard UPC collectives library by enabling: (1) communications from/to private memory, (2) customized message sizes on each thread, and (3) the use of teams, among other features. The library consists of about 50 in-place, vector-variant and team-based collective functions (not including the type variations of reduce, prefix-reduce and allreduce), alongside versions of many of them (e.g. rooted) and *get-put-priv* functions, which in total sum up more than 15000 SLOCs. The algorithms implemented for these collectives are intended to maximize parallelism and efficiency by exploiting one-sided communications with standard memory copy functions. Moreover, an implementation at library level of teams has been developed to support team-based collectives, providing a general functionality that can be applied to any UPC code.

Six representative codes have been used for a comparative evaluation of the implemented library, and the results have shown that the use of the extended collectives has provided good solutions for all tested cases in terms of both performance and programmability. The extended collectives have been able to provide a more compact implementation of different communication patterns for the selected applications. Moreover, the use of efficient collective algorithms enhanced performance for all the tests, especially for the 3D FFT code, in which the results have outperformed even the MPI counterpart (the UPC in-place collective obtained up to 28% of performance improvement for 256 threads). As a general outcome of the evaluation, these functions are able to obtain a better exploitation of computational resources as the number of threads and the amount of data to be transferred increases. In summary, these collectives provide a powerful and productive way for inexperienced parallel programmers to implement custom data transfers and parallelize sequential codes, as well as a wide variety of resources for expert UPC programmers, that can transparently take advantage of the optimizations implemented in this library.

## References

[1] Unified Parallel C at George Washington University. http://upc.gwu.edu. Accessed 4 July 2012.

[2] UPC Consortium. UPC Collective Operations Specifications v1.0. http://upc.gwu.edu/docs/UPC_Coll_Spec_V1.0.pdf. Accessed 4 July 2012.

[3] UPC Consortium. UPC Language Specifications v1.2. http://upc.gwu.edu/docs/upc_specs_1.2.pdf. Accessed 4 July 2012.

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

[5] UPC Wiki - Main Page. http://upc.lbl.gov/wiki. Accessed 4 July 2012.

[6] Z. Ryne and S. Seidel. A Specification of the Extensions to the Collective Operations of Unified Parallel C. http://www.upc.mtu.edu/papers/cstr05-08.pdf. Accessed 4 July 2012.

[7] Z. Ryne and S. Seidel. UPC Extended Collective Operations Specification. http://www.upc.mtu.edu/papers/UPC_CollExt.pdf. Accessed 4 July 2012.

[8] Reference Implementation of UPC Collective Functions. http://www.upc.mtu.edu/soft ware/-CollExt.tar.gz. Accessed 4 July 2012.

[9] MPI: A Message-Passing Interface Standard. Version 2.2. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf. Accessed 4 July 2012.

[10] Z. Ryne and S. Seidel. Ideas and Specifications for the new One-sided Collective Operations in UPC. http://www.upc.mtu.edu/papers/Onesided Coll.pdf. Accessed 4 July 2012.

[11] D. Bonachea. UPC Collectives Value Interface, v1.2. http://upc.lbl.gov/docs/user/README-collectivev.txt. Accessed 4 July 2012.

[12] 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 *11th IEEE Intl. Conference on High Performance Computing and Communications (HPCC'09), Seoul (Korea)*, pages 69–78, 2009.

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

[14] F. Cantonnet, Y. Yao, M. M. Zahran, and T. El-Ghazawi. Productivity Analysis of the UPC Language. In *18th Intl. Parallel and Distributed Processing Symposium (IPDPS'04), Santa Fe (NM, USA)*, page 254a, 2004.

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

[16] Rajesh Nishtala, Yili Zheng, Paul Hargrove, and Katherine Yelick. Tuning collective communication for Partitioned Global Address Space programming models. *Parallel Computing*, 37(9):576–591, 2011.

[17] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, 1997.

[18] J. Dinan, P. Balaji, E. L. Lusk, P. Sadayappan, and R. Thakur. Hybrid Parallel Programming with MPI and Unified Parallel C. In *7th ACM Intl. Conference on Computing Frontiers (CF'10), Bertinoro (Italy)*, pages 177–186, 2010.

[19] GWU Unified Testing Suite (GUTS). http://threads.hpcl .gwu.edu/sites/guts. Accessed 4 July 2012.

[20] T. El-Ghazawi, F. Cantonnet, Y. Yao, S. Annareddy, and A. S. Mohamed. Benchmarking Parallel Compilers: a UPC Case Study. *Future Generation Computer Systems*, 22(7):764–775, 2006.

[21] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez, R. Doallo, and J. C. Mouriño. Performance Evaluation of MPI, UPC and OpenMP on Multicore Architectures. In *16th European PVM/MPI Users' Group Meeting (EuroPVM/MPI'09), Espoo (Finland)*, pages 174–184, 2009.

[22] Z. Zhang and S. Seidel. Benchmark Measurements of Current UPC Platforms. In *19th Intl. Parallel and Distributed Processing Symposium (IPDPS'05), Denver (CO, USA)*, 2005.

[23] J. Dean and S. Ghemawat. MapReduce: A Flexible Data Processing Tool. *Commun. ACM*, 53(1):72–77, 2010.

[24] Carlos Teijeiro, Guillermo L. Taboada, Juan Touriño, and Ramón Doallo. Design and Implementation of MapReduce using the PGAS Programming Model with UPC. In *17th International Conference on Parallel and Distributed Systems (ICPADS'11), Tainan (Taiwan)*, pages 177–186, 2011.

[25] Berkeley Unified Parallel C (UPC) Project. http://upc.lbl.gov. Accessed 4 July 2012.

[26] NIST Matrix Market. FIDAP011: Matrices generated by the FIDAP Package. http://math.nist.gov/MatrixMarket/data/SPARSKIT/fidap/fidap011.html. Accessed 4 July 2012.

[27] ParaStation MPI - ParTec. http://www.par-tec.com/products/parastation-mpi.html. Accessed 4 July 2012.

[28] Webb Spam Corpus. http://www.cc.gatech.edu/projects/doi/WebbSpamCorpus.html. Accessed 4 July 2012.

**Carlos Teijeiro** received his B.S. (2006) and M.S (2008) degrees in Computer Science from the University of A Coruña, Spain. He is currently a research fellow and a PhD candidate in the Department of Electronics and Systems at the University of A Coruña. His research interests focus on High Performance Computing (HPC), especially parallel programming languages and distributed computing in the cloud.

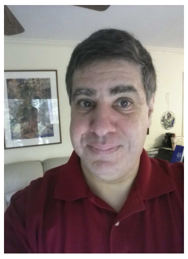
**Guillermo L. Taboada** received his B.S. (2002), M.S. (2004) and PhD (2009) degrees in Computer Science from the University of A Coruña, Spain. Currently he is an Associate Professor in the Department of Electronics and Systems at the University of A Coruña. His main research interest is in the area of High Performance Computing (HPC), focused on high-speed networks, programming languages for HPC, cluster/grid/cloud computing and, in general, middleware for HPC.

**Juan Touriño** received his B.S. (1993), M.S. (1993) and PhD (1998) degrees in Computer Science from the University of A Coruña, Spain. In 1993 he joined the Department of Electronics and Systems at the University of A Coruña, Spain, where he is currently a Full Professor of Computer Engineering and Director of the department. He has extensively published in the areas of compilers and programming languages for HPC, and parallel and distributed computing. He is coauthor of more than 120 technical papers on these topics.

**Ramón Doallo** received his B.S. (1987), M.S. (1987) and PhD (1992) degrees in Physics from the University of Santiago de Compostela, Spain. In 1990 he joined the Department of Electronics and Systems at the University of A Coruña, Spain, where he became a Full Professor in 1999. He has extensively published in the areas of computer architecture, and parallel and distributed computing. He is coauthor of more than 140 technical papers on these topics.

**José C. Mouriño** is graduated in Computer Science in 2000 from the University of A Coruña. He holds a PhD in Computer Science from the same University (2006). Between 1999 and 2005 he worked for the University of A Coruña and University of Santiago de Compostela as researcher in several I+D projects. He works at CESGA as Applications Senior Technician. His research interest includes parallel and distributed computing, Cloud computing and HPC in the Cloud.

**Damián A. Mallón** finishes his M.S. in Computer Science in 2008 in the University of A Coruña, being awarded with the academic excellence award due to his academic performance. The same year he started his professional career working in the Galicia Supercomputing Center, in a joint project between that center, Hewlett Packard, the University of A Coruña and the University of Santiago de Compostela. At the same time he started pursuing his PhD in Computer Science. Currently he develops his career at Forschungszentrum Jülich, and is a member of the ExaCluster Laboratory and the DEEP consortium.



**Brian Wibecan** received his M.S. in Computer Science from Boston University, Massachusetts. He did much of the compiler work for the first complete implementation of the UPC language, which became the first commercial version of UPC. He is one of the initial members of the UPC Consortium, in which he is still involved. Currently he works actively inside the UPC community with the development of the HP UPC compiler at Hewlett-Packard, and holds two US patents.