Easy Dataflow Programming in Clusters with UPC++ DepSpawn

Basilio B. Fraguela, Diego Andrade

Abstract—The Partitioned Global Address Space (PGAS) programming model is one of the most relevant proposals to improve the ability of developers to exploit distributed memory systems. However, despite its important advantages with respect to the traditional message-passing paradigm, PGAS has not been yet widely adopted. We think that PGAS libraries are more promising than languages because they avoid the requirement to (re)write the applications using them, with the implied uncertainties related to portability and interoperability with the vast amount of APIs and libraries that exist for widespread languages. Nevertheless, the need to embed these libraries within a host language can limit their expressiveness and very useful features can be missing. This paper contributes to the advance of PGAS by enabling the simple development of arbitrarily complex task-parallel codes following a dataflow approach on top of the PGAS UPC++ library, implemented in C++. In addition, our proposal, called UPC++ DepSpawn, relies on an optimized multithreaded runtime that provides very competitive performance, as our experimental evaluation shows.

Index Terms—libraries, parallel programming models, distributed memory, multithreading, programmability, dataflow

1 Introduction

While the exploitation of parallelism is never trivial, this is particularly true in the case of distributed memory systems such as clusters. The traditional approach to program these systems has relied on a message-passing paradigm provided by libraries available in the most successful programming languages for these platforms. This paradigm, based on isolated local views of separate processes that communicate by means of explicit messages, typically under an SPMD programming style, enables very good performance. Unfortunately it also leads to programs that are, in general, difficult to develop, debug and maintain. For this reason there has been extensive research on alternatives to express parallel applications on clusters.

One of the most successful abstractions proposed for the development of distributed-memory applications is that of a Partitioned Global Address Space (PGAS) [1], in which processes have both a private and a shared address space, the latter one being partitioned among them. Thanks to the distributed shared space, PGAS environments offer programming abstractions similar to shared memory, which are much more programmer-friendly than those of the traditional model based on separate local spaces for each process. For example, PGAS fosters the usage of one-sided communications, which contrary to the traditional message-passing paradigm, do not require the attention of the process that owns the accessed remote data. The shared view of the distributed data also simplifies the expression of large distributed data structures of any kind, the impact being more noticeable on irregular and pointer based data structures. Another advantage is that the distinction between private and shared space and the partitioning of the latter one allows users to reason about locality and access costs. Namely, the private space is always the one that can be more efficiently accessed and the shared local space is accessible in very similar times, while the accesses to remote shared data are the most expensive ones by far. This model has been implemented in a wide variety of languages [2], [3], [4], [5], [6] and libraries [7].

In our opinion, expressive libraries integrated in widely used programming languages simplify the adoption of new programming paradigms compared to new languages. The most important reasons for this are that they facilitate code reusability as well as integration with existing tools and frameworks, some of them so important as OpenMP or CUDA, and they usually present shorter learning curves than new languages. Given these advantages, reimplementing these languages as libraries in traditional languages that offer good expressivity looks like a promising idea. This is the case of UPC [3], which has been implemented as the UPC++ library in the widespread C++ language [8]. As an indication of its impact, UPC++ was found to be the most widely used alternative to MPI at NERSC in [9] despite its relative recency, which supports our opinion on the advantages of libraries with respect to new languages.

Despite the advantages of PGAS approaches and the large set of semantics and mechanisms they provide, they still lack important features available in other parallel frameworks. This way, a common restriction we have found is that although PGAS environments exhibit a large variety of synchronization mechanisms, from full/empty bits [5] or synchronized blocks [6] to clocks [4] and locks [3], it is difficult to efficiently develop many algorithms whose tasks present complex patterns of dependencies using these approaches. A very effective mechanism to express this kind of algorithms is to rely on the implicit dependencies between the tasks that are given by the definition of their inputs and outputs. While this strategy is supported by several tools with very different nature [10], as far as we know, none of them is integrated and relies on the semantics of a PGAS

• *Grupo de Arquitectura de Computadores, Facultade de Informática, Universidade da Coruña, Campus de Elviña, s/n. 15071 A Coruña, Spain. Phone: +34-981-167000-1219. Fax: +34-981-167160. E-mail: {basilio.fraguela, diego.andrade}@udc.es*

environment. In this paper we present UPC++ DepSpawn, which is, to our knowledge, the first tool to support this approach for the implementation of complex task-parallel applications in a PGAS environment. Given our previous discussion, which favors the use of libraries over new languages, our proposal is a library that operates on UPC++ [8] in the widely used C++ language. The pure library approach is supported by metaprogramming, extending the mechanisms and semantics of the DepSpawn library [11] in shared-memory systems to PGAS environments. As we will see, UPC++ DepSpawn not only enables a terse, simple and efficient mechanism to build codes using a dataflow model of computation in distributed memory environments, but it also facilitates the exploitation of multithreading within each process used in a PGAS application. This way, UPC++ DepSpawn is very well suited for the programming of current multi-core clusters, where it is often the case that a combination of process and thread-level parallelism is required to achieve the best performance.

The rest of this manuscript is organized as follows. First, Sections 2 and 3 briefly introduce UPC++ and Depspawn, respectively, explaining the programming style they lead to. Then, Section 4 presents our proposal to support the development of PGAS applications whose tasks are automatically synchronized and scheduled based on their implicit data dependencies. This is followed by an evaluation in Section 5 and a discussion of the related work in Section 6. The last section is devoted to our conclusions and future work.

2 PGAS PROGRAMMING WITH UPC++

Introduced in [8], UPC++ is an implementation of the UPC language [3] in C++ by means of a library. It exploits the large number of features of the C++ language, which supports several programming paradigms, such as object-oriented and generic programming, and powerful mechanisms such as operator overloading and templates. These properties not only give place to a high level of expressivity that reaches a level similar to that of new languages, but they also make C++ libraries competitive with compilers by enabling the movement of some computations from runtime to compile time, making UPC++ competitive with compiler-based approaches in [8]. In addition, the large expressivity of C++ allowed UPC++ to go beyond the properties of UPC, implementing new powerful features such as multidimensional distributed arrays or remote function invocations. Let us now briefly introduce the core features of UPC++ required to understand our contribution.

UPC++ applications consist of multiple concurrently executing control flows that in UPC are called threads. Because our proposal allows exploiting a new level of parallelism inside each one of these execution units that is akin to standard threads, in the rest of this paper we will use the term *process* to refer to the traditional UPC threads. As in any PGAS environment, each one of these processes has a private memory space, which is separate and independent for each process, and a shared global address space that all the processes can see, even if they are executing in a distributed memory environment. This shared space is partitioned among the processes, so that each process has a portion that is local to it, in the sense that it can access it faster because it is located in its physical neighborhood, while the portions associated with other processes typically incur slower access times, for example because they are located in other nodes in a cluster.

```
1  shared_array<int, 4> av(100);
2  shared_array<int> bv(100);
3
4  ...
5
6  void update(global_ref<int> a, int b) {
7    // computations that use a and b
8    // to compute tmp, e.g. int tmp = a + b;
9    a = tmp;
10 }
11
12 ...
13
14 for(int i = myrank()*4; i < 100; i += ranks()*4)
15   for(j = i; j < std::min(i+4, 100); j++)
16     update(av[j], bv[j]);
```

Listing 1: UPC++ example

While data objects built in UPC++ programs using the regular C++ notation are located in the private space of each process, the creation and manipulation of data structures in the shared memory space requires the usage of the UPC++ library. Its API provides functions to allocate and deallocate memory in this space as well as class templates that allow representing and accessing data located in the shared space. This is the case of the shared scalars `shared_var<T>` of a given type `T`, which are placed in the local memory of process 0, but can be accessed by any process. Another representative class is `shared_array<T, B>`, which provides shared arrays of elements of type `T` distributed in chunks of `B` elements across the spaces with affinity to the different UPC++ processes following a block-cyclic distribution. The argument `B` is optional, the default value being 1, which leads to a cyclic distribution of the array contents. A final example are the pointers `global_ptr<T>` and references `global_ref<T>` to elements of type `T` located in the shared space, which behave as their traditional counterparts, but just allowing to access shared data that can be either local or remote. On top of these and other related elements, UPC++ also provides higher level components such as multidimensional domains and shared arrays [8], [12].

The usage of the UPC++ data structures located in the shared space makes transparent the access to remote data because the syntax is the same no matter the accessed item is located in the portion of the shared space with affinity to the considered process or not. However for performance reasons, users should minimize the accesses to remote data as much as possible. Listing 1 exemplifies the UPC++ programming style with a parallel computation on a shared array `av` of 100 elements divided in a block cyclic fashion in blocks of four elements among the UPC++ processes. Namely, the purpose of the code is to store in each element `av[j]` the result of a computation that depends on the original value of `av[j]` and the element `bv[j]` of another array that follows a cyclic distribution. The computation is performed in function `update`, whose parameters are a global reference to an integer (`global_ref<int>`), which

just as regular references allows reading and modifying the associated integer, and a regular C++ `int`. When this function is invoked as `update(av[j], bv[j])`, the indexing of the shared arrays generates global references. The global reference obtained from `av[j]` is stored in the formal parameter `a` of the function, while the global reference from `bv[j]` is implicitly transformed into an `int`, which may imply a communication if the data is remote, and then stored in the parameter `b`. This way, through the global reference, the function can read and modify the underlying element `av[j]` and use as input the integer read from `bv[j]`. Notice that all the behaviors described are analogous to the ones we would expect when considering standard C++ references, just happening in the shared space.

A naïve approach to perform this computation would be to write a simple loop between 0 and 99 that executes `update(av[j], bv[j])` for each `j`. This would be incorrect because since the execution is SPMD, all the processes would perform the computations and assignments to all the positions of the array, resulting not only in bad performance but also in incorrect results since many processes could use as input elements of `av` already written by others. Our code follows a correct and performant approach in which the loop on `i` iterates on the beginning of each block assigned to our rank, obtained by means of the function `myrank()`, while the loop on `j` iterates on the elements of the current block. This way, not only are the results correct because each element is written by a single process, but the performance is also optimized. The reason is that each process works on the portion of the distributed array `av` located in its local memory, avoiding communications for reading and writing it. This way there will only be communications for `bv[j]` when it turns out to be remote.

Other relevant components of the API include shared space synchronous and asynchronous bulk data transfer functions, remote procedure invocation functions, and several synchronization mechanisms that will be discussed in Section 4. Interestingly, the possibility of creating both local and remote asynchronous tasks in UPC++ turns it into an Asynchronous PGAS (APGAS) environment [13], contrary to UPC.

We note for completeness that every UPC++ application must invoke function `init(&argc, &argv)` before using any UPC++ runtime routine or shared space variable. As expected, in this invocation `argc` and `argv` are the standard parameters of the `main` function. Similarly, it should shut down the UPC++ environment before exiting by invoking the provided function `finalize()`.

It deserves to be mentioned that at the point of writing this manuscript a new version of UPC++ has been proposed [14], which further proves the interest of this environment. Besides appearing after the development of our library, its specification is yet in draft mode and the implementation is still in progress. For these reasons our proposal and most discussions in this paper target [8]. While the new UPC++ implementation provides more powerful mechanisms in several areas than [8], as the other PGAS approaches we know of, it does not offer anything similar to UPC++ DepSpawn, thus it would also benefit from a version of our library. For this reason we will also briefly discuss how our work would integrate with this new proposal.

```
1  Tile A[N][N];
2
3  ...
4
5  void dgemm(Tile& dest, const Tile& a, const Tile& b) {
6    // Implements dest = dest + a x b
7  }
8
9  for(i = 0; i < dim; i++) {
10   spawn(potrf, A[i][i]);
11   for(r = i+1; r < dim; r++) {
12     spawn(trsm, A[i][i], A[r][i]);
13   }
14   for(j = i+1; j < dim; j++) {
15     spawn(dsyrk, A[j][i], A[j][j]);
16     for(r = j+1; r < dim; r++) {
17       spawn(dgemm, A[r][j], A[r][i], A[j][i]);
18     }
19   }
20 }
21
22 wait_for_all();
```

Listing 2: Cholesky DepSpawn example

3 Task parallelism with DepSpawn

DepSpawn [11], available at http://depspawn.des.udc.es, is a library that supports task-parallelism expressed by means of functions that carry their dependencies only through their arguments. A unique characteristic of DepSpawn with respect to other approaches in this field is that the labeling of the dependencies is totally implicit. Namely, rather than requiring the annotation of the functions with directives or using an explicit library API to specify the dependencies, they are directly inferred from the type of the formal parameters of the functions. In C++, which is the host language for DepSpawn, the type of the parameters of a function indicate whether the associated arguments will be passed by value or by reference, and whether they will be modifiable or not, which is indicated by the `const` type qualifier. Arguments passed by value are copied, so that their modifications inside the function cannot be observed outside. This way, DepSpawn regards all arguments passed by value as only inputs to the functions. Similarly, while the parameters that have a constant reference type do not perform such copy, being associated with the original argument, they indicate that the argument provided will not be modified. As a result, they are also only considered as task/function inputs. Finally, non-constant reference parameters allow both reading and modifying the associated argument, and constitute thus both inputs and outputs of the considered task/function.

This implicit approach for the computation of the dependencies gives place to a very terse notation for the expression of parallelism. Namely, while DepSpawn provides a richer API, two functions suffice to express complex task-parallel applications with arbitrary dependencies among their tasks. The first one is `spawn`, which requests the execution of the function `f` with the arguments `a`, `b`, `c`, ... as a parallel task that respects the dependencies on its arguments using the notation `spawn(f, a, b, c, ...)`. The second one is `wait_for_all`, which waits for all the pending tasks to finish. This is illustrated in Listing 2, which uses DepSpawn to express a task-parallel Cholesky factorization performed by tiles of a given type `Tile`. The figure includes a sketch for function `dgemm` that shows that

the task parameters rely on the standard C++ datatypes. In this case DepSpawn will infer that the first argument can be both read and written, while the remaining ones are only inputs.

While the idea of running out of order tasks whose dependencies are extracted from the parameter types could be implemented by means of a layer on top of existing task parallel frameworks with support for dependencies, DepSpawn provides its own engine, which is heavily decentralized and parallelized [11]. The current implementation of the DepSpawn runtime works on top of the Intel® Threading Building Blocks (TBB) library [15], whose low level API is used for the construction of the low level tasks. Its main responsibility is the scheduling of these tasks once DepSpawn informs TBB that they are ready for execution. Finally, it deserves to be mentioned that DepSpawn, which only supports shared-memory environments, has been compared in terms of performance and programmability to some of the most relevant alternatives in this field achieving satisfactory results [10].

4 UPC++ DepSpawn

Before delving into the details of our proposal, we will motivate its interest. UPC++ [8] provides synchronizations mechanisms based on locks, barriers, futures, finish constructs, and events. The three latter mechanisms are related to remote function invocation based on the `async` function, which allows requesting the execution of arbitrary functions in remote locations using the syntax

```
future<T> f = async(place)(function, args)
```

where `place` can be a single thread ID or a group of threads. The execution is asynchronous, so that the invoking thread only waits for the result when it tries to use the returned future. Finish constructs can be used to wait for the completion of asynchronous tasks spawned in a given syntactical scope. Finally, events are objects that specify dependencies between asynchronous tasks. Each event can be associated with one or several tasks as postcondition or precondition for its/their execution. An event is signaled when all the tasks it has been associated with as postcondition have finished. At that moment, it allows the execution of those tasks to which it has been associated as precondition. The fact that each UPC++ task can only trigger and depend on a single event severely restricts the set of dependency patterns that can be efficiently represented by this mechanism. Let us consider for example the simple task dependency graph in Fig. 1, in which task `T2` is required in two different sets of dependencies. Although multiple tasks can be registered with the same event, so that it is only complete when all of them finish, if for example we register `T1` and `T2` directly in the same event, it cannot be later used to indicate the dependencies for `Ty` without artificially including `T1` as one of its dependencies, which is not the case. The simplest solution for this problem using the syntax and semantics described in [8] is shown in Listing 3. The code triggers two separate empty tasks when `T2` finishes, which is indicated by event `e2`. This way each one of them can contribute to a different event, one of them being `ex`, which controls the execution of task `Tx` and the other one being `ey`, whose signaling allows the execution of task `Ty`.

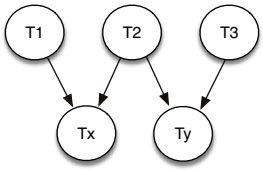

Fig. 1: A simple task dependency graph

```
1  event ex, ey, e2;
2  async(p1, &ex)(T1); // Signals ex after T1 finishes
3  async(p2, &e2)(T2);
4  async(p3, &ey)(T3);
5    // Runs empty lambda after e2 is signaled,
6    //then signals event ex
7  async_after(px, &e2, &ex)( [](){} );
8    // Runs Tx after event ex is signaled
9  async_after(px, &ex)(Tx);
10 async_after(py, &e2, &ey)( [](){} );
11 async_after(py, &ey)(Ty);
```

Listing 3: Implementation of the task dependency graph in Fig. 1 using UPC++ events

While these facilities allow the construction of complex task dependency graphs, they are clearly more limited and imply much more programming effort than a dataflow approach, where the dependencies are implicitly expressed by the data used by each task. The UPC++ futures proposed in [8] are definitely a step in the right direction, but they have many restrictions. For example, the proposed syntax can only generate one future per invocation, which forces to package together results that in an optimal implementation can have different destinations. In addition, the associated packing and unpacking contributes to the programming effort. Also, while futures are well suited to express RAW (read after write) dependencies, they do not seem a natural option to express WAW and mainly WAR dependencies. It is also noticeable that they can introduce important overheads, as the fact that they hold the result of computations implies that they cannot be directly stored in their final destination when they are not simply temporaries. This can be particularly inconvenient when it affects data in distributed structures, as it can give place to unneeded communications. Being complex objects by themselves, the correct handling of futures can require additional management in certain environments. This is the case of sharing non-thread safe futures between threads. Similarly, many future implementations are not serializable and/or cannot be shared by multiple processes in distributed memory. This makes it hard to share them as dependences of tasks to be executed in different processes, unless a master-slave model is followed in which a single process orchestrates all the tasks. Since the master needs to set up all the dependencies, receive all the notifications and submit all the tasks to the slaves upon the readiness of the involved futures, it can become a natural bottleneck. Also, because of the need to either block or poll on the futures until their results are ready, they give place to suboptimal executions of dataflow codes unless combined with constructs that allow dynamically triggering tasks only when the futures they depend on are ready. While most implementations of futures provide these mechanisms [14], [16], [17], [18], such elements are missing in [8].

Not only UPC++, but also the other PGAS implementations we know of would benefit from a totally implicit dataflow approach. For this reason, we set to provide a PGAS solution of this kind on top of UPC++ by adopting the philosophy of DepSpawn. The result, UPC++ DepSpawn, is now described first in terms of syntax and semantics, and later in terms of implementation.

4.1 Programming model

UPC++ DepSpawn seeks to enable the clean and efficient expression of dataflow computations on UPC++ distributed objects. These will be thus computations broken in tasks that only communicate through their arguments, and whose execution will respect their dependencies. The easiest way to express task-parallel computations is to write them just as in the sequential version, letting a compiler or runtime system take care of the parallelization details, which is the approach followed by DepSpawn. Our proposal inherits thus this behavior, so that the parallelized codes have the same structure as their sequential counterparts, just replacing the invocations to the functions `f(args)` that implement each task with `upcxx_depspawn(f, args)`. The library also provides the `upcxx_wait_for_all()` function in order to specify synchronization points in which all the pending tasks must have completed.

An important issue is how to express the intent of each task with respect to each parameter of the function that implements it while allowing remote accesses. For maximum flexibility, UPC++ DepSpawn supports as task arguments both regular C++ objects, which are private to the process, and UPC++ objects from the shared space. The latter are of particular interest, as they convey the dependencies with other PGAS processes that can also operate on them. The C++ pass-by-value and pass-by-reference semantics discussed in Section 3 can also be applied to UPC++ objects with some modifications. Namely, the most widely used UPC++ shared objects are by far its distributed arrays. As we saw when discussing Listing 1, when one of such arrays of elements of type `T` is indexed, we do not directly get a an item of type `T`. Rather, the indexing returns a `global_ref<T>` object that stores the process and physical address where the associated element is stored. This global reference object allows both reading from and writing to that potentially remote location. Namely, reads are supported by C++ conversion operators, which implicitly return the element of type `T` associated with the global reference when it is used as input in an expression. Writing to the remote location is achieved by means of the overload of the assignment operator of the `global_ref<T>` class, which stores in the associated location the value received as right hand side of the assignment. Altogether, this allows both the read and the write (assignment) expressions to perform as expected on `shared_array`s. Notice that global references are thus the equivalent to the `&` reference C++ operator for UPC++ shared objects, as they can be also associated with `shared_var` scalars. The equivalency also covers the `const` type qualifier, as very much like `const&` C++ standard references, `global_ref<const T>` global references only allow read accesses to the associated data. Therefore, global references will be the parameters used in functions to express reference and const reference semantics on data located in the shared space. The pass-by-value semantics are simply represented by the use of a regular C++ type, which will be locally built by the process from the potentially remote data when the dependencies on it have been fulfilled.

```
1  shared_array<Tile, 1> A(N * N);
2  #define _(i, j) ((i) * N + (j))
3  ...
4
5  void dgemm(global_ref<Tile> dest, const Tile a,
         const Tile b) {
6    // Implements dest = dest + a x b
7  }
8
9  for(i = 0; i < dim; i++) {
10   upcxx_spawn(potrf, A[_(i,i)]);
11   for(r = i+1; r < dim; r++) {
12     upcxx_spawn(trsm, A[_(i,i)], A[_(r,i)]);
13   }
14   for(j = i+1; j < dim; j++) {
15     upcxx_spawn(dsyrk, A[_(j,i)], A[_(j,j)]);
16     for(r = j+1; r < dim; r++) {
17       upcxx_spawn(dgemm, A[_(r,j)]), A[_(r,i)]),
             A[_(j,i)]));
18     }
19   }
20 }
21
22 upcxx_wait_for_all();
```

Listing 4: Cholesky UPC++ DepSpawn example

Listing 4 shows the UPC++ DepSpawn version of Listing 2, developed according to the explanations above. As we can see, the biggest complication is that `shared_array` is a unidimensional array, which forces to linearize the indexes of the tiles. It must be noted though that UPC++ developers offer more complex distributed arrays and of course users can write their own multi-dimensional array classes with the elements provided by UPC++, which is in fact what we did for our tests.

Some characteristics of the programming model are strongly related to implementation decisions. For example, since UPC++ follows a SPMD model, we could have chosen between two alternatives for the execution of the UPC++ DepSpawn applications. One, which we call centralized model, would have been to execute the dataflow code in a single process, which would act as master, and to require the other processes to call a runtime function to operate as servers during that portion of the execution. The master could either directly choose which server would execute each task or populate a task pool from which servers could take new ready tasks when idle. The second model, which we call distributed, would require all the processes to run the code in parallel, agreeing during the execution on who would execute what. The second option has been chosen, as the centralized model naturally leads to a bottleneck as the number of processes involved grows. This is reflected in Listing 4, where the code shown is executed by all the UPC++ processes so that all of them have all the information on the tasks and dependencies involved, even if each process will only execute a subset of the tasks.

Also, we felt that in current systems, where every node has one or several processors with an increasing number of cores, it made much sense to automatically support the parallel execution of multiple tasks in each process by means of multithreading. Our main aim with this was similar to

that of MPI+X approaches, namely to reduce the number of processes in order to minimize the number of interactions with the communication layer both for data exchanges and synchronizations. A second purpose more specific to our tool was to reduce the overhead associated with the dynamic generation of the task dependency graph (TDG), as in our implementation only one thread in each process needs to do it. Also, we designed this feature so that it is totally oblivious to users, except for the specification of the number of processes and threads per process to use during the execution, making it extremely convenient. This specification is performed either by invoking function `set_threads(n)`, where `n` is the number of threads requested, or by setting the environment variable `DEPSPAWN_NUM_THREADS` to this value before the application begins its execution.

While the main target of our proposal are codes with complex patterns of dependences, as they benefit the most from the dataflow behavior provided by our library, UPC++ DepSpawn can also simplify the development and maintenance of very regular codes. For example, Listing 1 implements an embarrassingly parallel loop whose only complication is the need to assign correctly the iterations to perform by each process in order to maximize locality. UPC++ DepSpawn can be used to parallelize this loop by spawning in each iteration the task to execute with the proper arguments as shown in Listing 5. Since, as we will see in Section 4.2, UPC++ DepSpawn follows a owner-computes rule in order to minimize data transfers, this code gives place to the same optimal distribution of computations per process as Listing 1 using a single loop. Notice that Listing 5 is not only arguably cleaner than the original code, but it also enjoys the advantage that while changes in the distribution of `av` for whatever reason will require manually readjusting Listing 1, or rewriting it in order to base it on the function that returns the blocking factor for array `av`, the UPC++ DepSpawn version will automatically self-adapt to any new arbitrary distribution of the arrays. But the most critical advantage is probably that while Listing 1 only provides process-level parallelism on top of UPC++, the UPC++ DepSpawn version exploits in addition multithreading within each process. This way each process can execute in parallel as many loop iterations as possible using the threads available without further programming cost. As mentioned above, replacing process-level parallelism by thread-level parallelism is a typical strategy for the optimization of applications executed in hybrid shared/distributed memory systems, since data sharing and synchronization is cheaper between threads than between processes. In the case of our example, the advantage would be that an increasing ratio of the elements of `bv` accessed would be directly read from the local memory of the process rather than obtained from other process, which is slower.

It deserves to be mentioned that while the new UPC++ version in development [14] provides much richer mechanisms and semantics than [8], it offers neither shared arrays nor global references, which play a key role in our proposal. Nevertheless, as one would expect, this UPC++ version provides a shared space with support for distributed objects that can be remotely accessed by means of global pointers. This enables to build arbitrary distributed data structures whose elements can be accessed from any process

```
1  void update(global_ref<int> a, int b) {
2    // computations that use a and b
3    // to compute tmp, e.g. int tmp = a + b;
4    a = tmp;
5  }
6
7  ...
8
9  for(int j = 0; j < 100; j++)
10   upcxx_spawn(update, av[j], bv[j]);
11 upcxx_wait_for_all();
```

Listing 5: Listing 1 rewritten with UPC++ DepSpawn

participating in the execution, including distributed arrays. This way, it is perfectly possible to replicate the same idea of globally indexable distributed arrays on [14]. As for the global references, their elimination is due to a design decision to foster the exploitation of non-blocking accesses, as the access through global references in [8] implies blocking global pointer dereferences. Still, given the existence of global pointers in the new proposal, it is easy to build based on them global reference classes analogous to those of [8]. In addition the new classes could be enhanced by exposing both blocking and non-blocking behaviors when accessing the associated element of the shared space. Regarding the functionalities of UPC++ used by our runtime, they are provided by both versions of UPC++. This way, nothing precludes the implementation of UPC++ DepSpawn on the new proposal.

4.2 Implementation

The decision to execute in parallel the dataflow code in all the processes enables all of them to have a view of the task dependency graph (TDG), each process having one copy that is shared by all its threads. Namely, the main thread of each process executes the main code and inserts the node associated with the task expressed by each invocation to `upcxx_spawn` in the TDG . During the insertion, the decision on which process will execute the task is taken and the dependencies with preceding tasks are identified and stored. Both actions are performed in a totally independent way in each process using a priority function based on the location of the arguments and their access mode and the local TDG, respectively. The selection function used in our experiments chooses to run a task in the process that owns most of its argument data, giving double weight to the data that will be written. This gives place to an owner-computes rule most of the time. Regarding the dependence discovery process, it is optimized in several ways. First, only dependencies in which at least one of the two tasks implied is local, i.e. is to be run locally, are considered, as the other ones require no action from the process. Second, tasks that are still in the TDG but are known to have finished are disregarded. Third, dependencies that are covered by already existing dependencies are ignored. Thus for example if tasks T1, T2 and T3 would have written to some common data in a sequential execution, our runtime neither explores nor records the dependency of T3 on T1 because T3 depends on T2, and T2 already depends on T1.

If no dependencies are found for a local task, it is stored in a ready pool from which any idle thread can take it for

execution, otherwise it is left in the pending pool. The remaining activities in our implementation can be performed by any thread. This way, when a worker thread finishes the execution of a task, it is also responsible for examining the local TDG in order to notify the tasks that depend on the just finished one. In the case of local tasks this implies moving them to the ready pool if this was their last pending dependency. For dependent remote tasks, the thread will have to notify their owner process on the completion of the task. After this, the thread recycles the storage associated with the task and its dependencies leaving the TDG in a consistent state. Since multiple threads can perform different kinds of updates on the TDG in parallel, our design is thread-safe. Also, for performance reasons our runtime is based on atomic operations based on the C++11 standard, our TDG being in fact almost lock-free.

It must be noted that the described design can give place to enormous pools of tasks, and thus very large TDGs to explore for each insertion despite the pruning strategy described above. This will happen for example when the average runtime of the tasks is much larger than the time that the main thread requires to perform an insertion or when the dependencies make it impossible to exploit enough task parallelism to keep these data structures within reasonable bounds. These circumstances negatively impact performance, as the time required to find the dependencies in the TDG grows, and locality is hurt because of the increase of the memory footprint. For this reason another optimization in our runtime consists in stopping the generation of new tasks when the number of live tasks in the system is very large, only resuming it when this number falls again below a given limit. During these periods, the main thread does not remain idle, devoting itself to the cooperative execution of tasks that may be available in the ready pool and the processing of remote requests.

The inter-process communications and synchronizations are naturally performed on top of UPC++, which relies on GASNet [19] for these purposes. Our library has been designed as a separate layer that operates on top of UPC++. This way, no modifications have been performed on the UPC++ runtime, which allows installing UPC++ DepSpawn on a standard UPC++ installation, only requiring that GASNet and UPC++ have been compiled in thread-safe mode. The accesses to remote data can be performed either by the runtime or by the user. The reason is that, as explained in Section 4.1, the `global_ref` objects that represent access by reference to remote data do not access the data by themselves, but only provide a pointer to it. As a result, our runtime respects the dependencies on the data associated with these objects in order to determine when it is safe to execute a task, but it does not perform the transfers associated with them. Rather, such transfers happen inside the tasks under the control of the user. Nevertheless, when the formal parameter of a task associated with a remote data is not a `global_ref` but an object of a regular C++ type that must hold the remote data, our runtime requests the data transfer in order to fill in the argument before invoking the task. This happens for example in Listing 5, where the parameter `b` of the task `update` is a regular C++ type (`int`), while the associated argument is the `global_ref<int>` returned by the `bv[j]` used in its invocation. When our runtime launches to execution the task because its dependencies have been fulfilled, the arguments provided must be copied or transformed into the parameters used by the funcion. Thus at that point the `global_ref<int>` returned by `av[j]` is directly copied in the formal parameter `a`, which has the same type, while the conversion operator of `global_ref<int>` allows transforming the `global_ref<int>` returned by `bv[j]` into the `int b`. The conversion operator takes care of bringing the data from its location and turning it into an `int`, just as it happens with the uses of the global reference `a` within the function `update`. If, on the contrary, the type of the formal parameter `b` had been `global_ref<const int>`, where the `const` modifier informs our library on the intention to only use but not modify the related argument, no transfer would have been performed at the moment of invoking the function. Rather, the `global_ref<int>` returned by `bv[j]` would have been transformed by means of a conversion operator into a `global_ref<const int>` that would be copied in the parameter `b`. Then, the actual transfer would happen inside function `update` when `b` were used with the intention of accessing the associated integer.

While the described design couples modularity with flexibility, it still lacks two critical optimizations. The first one pertains to the situation when several tasks that run in the same process access remote data items, resulting in replicated data transfers. For example, function `tsrm` in line 12 of Listing 4 requires tile (i, i) as input to update each one of the tiles (r, i), all of which can be updated in parallel within the loop in line 11. In a typical problem where there are many more tiles than processes and those tiles are cyclically or block-cyclically distributed, each process will have to update several tiles in this loop, and each one of those updates will require a copy of tile (i, i). A naïve implementation would result in a transfer of this tile from its owner process to the other ones for the execution of each `tsrm` task. Our runtime, however, includes a cache, so that there is a single transfer for each process, all the tasks except the first one reusing the cached value. Since tasks are basically assigned under an owner-computes rule, the current implementation targets data read from other processes, and it can operate either automatically or manually. Under automatic operation, the arguments that UPC++ DepSpawn identifies as read-only inputs of a task to be run locally automatically transit through the cache. The cache always returns a local address either because (a) the data is already associated with the process, (b) a local copy is found in the cache or (c) a remote request is performed to fill in a copy in the cache and then its address is returned to the runtime. Under manual operation the user can explicitly identify which accesses go to the cache by using the provided type `cached_global_ref<T>` instead of `global_ref<T>`. Regarding coherency, since UPC++ DepSpawn knows all the operations performed by every task on each piece of data, it can automatically keep the cache of each process consistent without any user intervention. The replacement policy of our cache is LRU, while its size can be modified by the user, not only before the execution of the code, but at any point in the program. Notice that this optimization is another reason why even UPC++ codes without complex dependency patterns can benefit from being rewritten using UPC++ DepSpawn. The motive is that if their remote ac-

TABLE 1: Experimental environment.

Feature	Value
#Nodes	32
CPUs/Node	2 x Intel Xeon E5-2680 v3
CPU Family	Haswell
CPU Frequency	2.5 GHz
#cores/CPU	12
Total #cores	$32 \times 2 \times 12 = 768$
Memory/node	128GB DDR4
Network	Infiniband FDR@56Gbps
Compiler	g++ 6.3
GASNet	1.28.0
Intel® TBB	2017.0

cesses exhibit some locality, caching them will be critical for achieving good performance, which is effortlessly provided by our library.

The second optimization relates to the process of discovery of dependencies in the TDG. Since UPC++ DepSpawn supports the discovery of dependencies between tasks due to the usage of arbitrary overlapping memory regions, it needs to compare every argument of each new task with those of the live tasks in the TDG. Even when this process is optimized, for example by using sorted lists of the arguments, which reduces the number of comparisons for a couple of tasks $i = 1, 2$ with T_i arguments each from $T_1 \times T_2$ to $T_1 + T_2 - 1$, this can still imply an important overhead when the number of tasks to check is large. Nevertheless, many algorithms only present dependencies that do not require checking overlaps but only exacts matches. That is, in these codes the memory regions of the task arguments either completely match or are totally disjoint. This allows a potentially faster dependency discovery algorithm based on a hash table on the addresses of the task arguments that avoids the comparison with the arguments of all the live tasks. Since the runtime cannot know in advance whether the code to run fulfills this property, this behavior, which we call the exact match mode, is only enabled under the request of the user. In fact, this and most parameters for the execution of UPC++ DepSpawn, such as the cache size mentioned before, can be provided to the runtime either by means of environment variables or configuration functions provided by our library that can be invoked at any point in the code. A minority of the configuration options are enabled by providing definitions during the compilation of the source code, typically by means of the flag `-D`. This is the case of options whose runtime management would increase unnecessarily the overheads of the library and/or whose implementation depends on data type manipulations, which are performed during compilation. The most relevant example of this second set of options is the usage of automatic caching.

5 Evaluation

The experiments were performed in the Linux cluster described in Table 1, which consists of 32 nodes with 24 cores each, totaling 768 cores. The optimization level `O3` was used in all the compilations. Also, since GASNet, the communication library on which UPC++ DepSpawn operates, provides a conduit that runs over Infiniband networks like the one in our cluster using the Open Fabrics Verbs API, our tests were performed using this conduit. The evaluation is split in two parts. First, the overheads of our proposal are evaluated in Section 5.1. Then, a comparison with other implementation strategies is performed in Section 5.2.

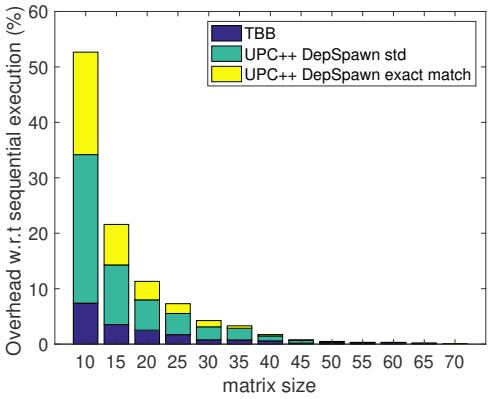

Fig. 2: Overhead with respect to a sequential execution

5.1 Runtime overheads

We have first measured the overhead of our runtime by means of an experiment consisting on executing a varying number of independent simple tasks of different granularities on a single core. The tasks were multiplications of double precision square matrices of different sizes written as three nested loops. We took as baseline a sequential implementation and measured the overhead of spawning each matrix product as a task using the TBB library, and using UPC++ DepSpawn with and without exact match mode. Figure 2 shows the overheads obtained with respect to the sequential implementation. For each matrix size we made experiments launching $2^i, 7 \leq i \leq 14$ parallel, i.e., without dependencies, tasks. The overheads measured were very stable for all the number of tasks and thus the figure shows the average value observed. We can see that the TBB runtime on which our threading relies is very lightweight, and while the overhead of UPC++ DepSpawn reaches 34% for the smallest task, which is very reasonable given its small size, it falls below 8% for the 20×20 matrix product and it is already below 0.7% for the 45×45 matrix product. It may look surprising that the exact match mode optimization was counterproductive in this test. The reason is that this optimization requires additional memory structures with respect to the standard mode, thus requiring more memory management and synchronizations. In fact the hash table used by this optimization is the only element of our runtime that uses a lock in our current implementation. The result is also a testament to the high degree of optimization of the standard mode. In fact, as one would expect, the kernel of UPC++ DepSpawn in standard mode is very much based on the runtime of DepSpawn, which showed to be on par with state of the art tools such as OpenMP in [10].

Our second experiment evaluates the worst-case impact of the runtime in a parallel execution. This overhead is associated to the execution of a thread in each node that dynamically analyzes the dependences between all the tasks spawned and schedules them accordingly. It must be noted that UPC++ DepSpawn is mostly of interest for applications

with complex patterns of dependences, where it is difficult to code by hand in regular ways the synchronizations and communications required. Therefore in these algorithms, although not only in them, one can expect that there will be very often at least one core available per node that would be otherwise idle because of communication or synchronization requirements. This core, or even just one of its hardware threads, can be thus devoted to executing our runtime without generating observable overhead. This is particularly true as the number of cores per cluster node continues to grow.

In order to measure the worst possible impact of this overhead, our experiment considers an embarrassingly parallel execution that requires all the cores available and in which no communications are needed, so that the overhead cannot be hidden. This is modeled by a doall parallel loop with as many iterations as cores involved, so that each iteration is an independent task to be run in a different core that only accesses data present in the node it is assigned to. Other measures taken to maximize the relative overhead of the runtime were to consider small uniform tasks and to implement them as repetitive operations on processor registers, so that there is almost no memory access cost involved either. Since the purpose of this test is to measure the cost due to the iteration of the runtime on the tasks and not just the standard overhead of the runtime associated to the packaging of the tasks, which is already evaluated in the previous experiment, the tasks used have at least 2^{14} flops, where we measured this overhead to be $\leq 2.5\%$ of the sequential equivalent execution time. The experiment consisted in measuring, for each number n of nodes in our cluster, which has 24 cores per node, the slowdown of spanning $24 \times n$ different tasks in a loop using UPC++ DepSpawn, so that there were 24 tasks assigned for execution to each node (i.e. one per core), with respect to performing an equivalent manual parallel execution. The manual implementation spawns at once 24 tasks of the same cost in each node using `tbb::parallel_for` from Intel TBB, and synchronizes the nodes with a barrier to ensure they all finished. Figure 3 shows the result for four task sizes using the label *naive*. The slowdown tends to grow with the number of nodes because while the baseline always performs the same amount of work per node, 24 parallel tasks, our runtime must analyze in sequence the whole task graph in each node, and in this experiment its size grows linearly with the number of nodes. This limits the scalability in this setup in which it is not possible to hide this overhead. This way, for the largest number of nodes, the slowdown for the smallest task size, which basically equates the matrix product of tiles of size 20×20, is 48.6%. This number improves as the task size increases up to 8.1% for the tasks of 2^{20} flops. Notice that this is the number of flops of a matrix product of tiles of size 80×80, which is still much smaller than the typical optimal tile sizes found in our tests in Section 5.2, as they were usually of size at least 200×200.

We must note that in the scenario described, programmers experienced in parallelism would try to optimize these codes by breaking the parallelism in two levels. This involves spawning at the top level a smaller number of tasks –in the extreme case, a single task per node– so that each one take cares of the processing of a proportional number of original tasks, and using for example `tbb::parallel_for` to perform such processing in parallel within each top level task. This strategy, which we have measured to be beneficial both for the UPC++ DepSpawn and the TBB baseline versions, considerably reduces the overhead of our runtime even when also considering the new optimized baseline. With this strategy, whose results are labeled as *opt* in Figure 3, the maximum performance lost when using all the nodes is 23.2% for the tasks of 2^{14} flops and the minimum one 5.5% for the tasks of 2^{20} flops. These results highlight that, since the analysis and scheduling overhead grows with the number of tasks to manage, two related very simple and effective approaches to reduce the runtime overhead are the increase of the task granularity and the usage of several levels of parallelism. This is generally feasible because the usage of large numbers of processing resources is usually related to the exploitation of data parallelism, in which the repetitive application of an identical operation of independent data elements makes it easy to adjust the granularity of the tasks. The exploration of further optimizations to reduce this overhead is part of our future work. A critical strategy to explore consists in pruning the task graph analyzed in each node so that only the local tasks, their dependences and their dependent tasks are considered, as this is all that is actually needed in each node.

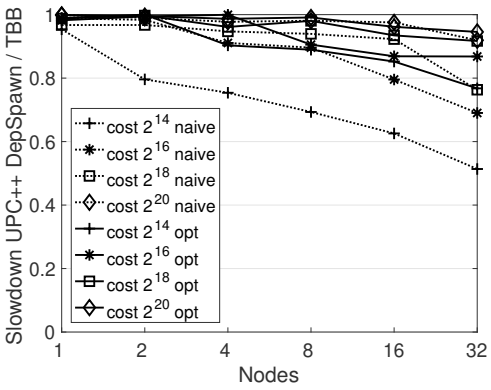

Fig. 3: Slowdown with respect to a hand written embarrassingly parallel execution of tasks of different granularities

5.2 Comparison with other approaches

The remaining experiments rely on benchmarks from [20], which elaborates on the famous Berkeley dwarfs [21] that characterize different classes of emerging applications. Namely, Gajinov et al. propose a benchmark to cover each class of dwarf and analyze its programmability under several paradigms. Their study concludes that the dwarves that are best suited for dataflow computing are dense algebra, sparse algebra, and structured grid problems. Then, there are another five dwarves in which dataflow is one of the best models for expressing it, but other model(s) can be also appropriate. In view of this, our evaluation is based on the right-looking Cholesky decomposition of a lower triangular matrix used as example in Listings 2 and 4, and a LU dense decomposition, which represent dense linear

```
for (int k = 0; k < N; k++) {
  lu0(A[k][k]);
  for (int j = k+1; j < N; j++)
    if (A[k][j] != null)
      fwd(A[k][k], A[k][j]);
  for (int i = k+1; i < N; i++) {
    if (A[i][k] != null) {
      bdiv(A[k][k], A[i][k]);
      for (int j = k+1; j < N; j++) {
        if (A[k][j] != null) {
          if (A[i][j] == null)
            A[i][j] = bmod(A[i][k], A[k][j]);
          else
            bmod(A[i][k], A[k][j], A[i][j]);
        }
      }
    }
  }
}
```

Listing 6: Sparse LU algorithm

```
while(!convergence_reached()) {
  for(int i = 1; i < N - 1; i++) {
    for(int j = 1; j < N - 1; j++) {
      m[i][j] = f(m[i][j], m[i-1][j], m[i+1][j],
                           m[i][j-1], m[i][j+1]);
    }
  }
}
```

Listing 7: Gauss-Seidel stencil

algebra problems; a sparse LU factorization, which represents the sparse algebra problems; a Gauss-Seidel stencil computation, which represents the structured grid dwarf; and the Viterbi algorithm for the Hidden Markov model, which represents the graphical models dwarf, one of the five hybrid dwarves identified in [20].

Let us notice that the sparse LU factorization, depicted in Listing 6, is both irregular and dynamic, as new tiles can be created during the execution of the program following an irregular pattern. The listing also indirectly illustrates the dense LU benchmark, since the dense counterpart is the same algorithm, just removing the conditional statements and considering only the branch where the tile is not empty.

The structured grid dwarves perform computations on regular multidimensional grids in a sequence of steps until either convergence is reached or a certain time interval is completed. This is indeed the case of the Gauss-Seidel method considered, shown in Listing 7, which is applied repetitively on a matrix until a convergence condition is met. As explained in [20], the parallelization of this method takes the form of a wavefront, which can be implemented at element or tile level, and using either barriers, with one per diagonal, or a dataflow approach. The dataflow solution, besides avoiding the barriers, usually does not synchronize at the end of each application of the method, but each n applications in order to further exploit parallelism. This is safe for small values of n ([20] suggests values around ten or twenty) because typically a few thousand applications or more are needed to converge to a satisfactory level,.

As for the Viterbi algorithm, depicted in Listing 8, it considers a number of states and a series of observations that appear in sequence. For each observation `t` it computes the most likely preceding state, which depends on the probability for every state `i` in the preceding observation,

```
for (int i = 0; i < NStates; i++) { //initialization
  initialize(delta[0][i]);
  psi[0][i] = 0;
}
for (int t = 1; t < num_observations; t++) { //main loop
  for (int j = 0; j < NStates; j++) {
    for (int i = 0; i < NStates; i++) {
      find_max_prob(delta[t-1][i], delta[t][j], psi[t][j]);
    }
    adjust(delta[t][j], psi[t][j]); //very lightweight
  }
}
```

Listing 8: Viterbi algorithm for the Hidden Markov model

stored in `delta[t-1][i]`. The probability of each state `j` for a given observation `t`, stored in `delta[t][j]`, together with the pointer to the most likely preceding state for that state, stored in `psi[t][j]`, can be computed in parallel once those of the previous observation are available. The computations are performed in function `find_max_prob`. This routine updates with a new probability and its associated pointer `delta[t][j]` and `psi[t][j]`, respectively, if the probability derived from the currently analyzed state in `delta[t-1][i]` happens to be larger than the probability previously stored in `delta[t][j]`. This way, the natural parallelization strategy is to perform in parallel loop `j` in Listing 8 and synchronize the threads/processes before the beginning of each iteration of loop `t`. In a distributed memory environment this would also be the best moment to broadcast the portions of the row `delta[t-1]` computed by each process to the other ones.

All the algorithms were written in a tiled style, and the basic blocks of the algebraic algorithms rely on BLAS operations which were provided by the highly optimized OpenBLAS library version 0.2.20 and work on double-precision matrices. Also, all the experiments use one process with 24 threads per node, one per hardware core available. As a result, since UPC++ DepSpawn was in charge of exploiting the thread parallelism, OpenBLAS was configured to run its routines in single-threaded mode.

Two critical decisions when implementing these algorithms on distributed or hybrid memory systems are the distribution of the tiles among the nodes and the selection of the tile size. The importance of the first decision is illustrated by Fig. 4, which shows the performance obtained by the dense linear algebra algorithms implemented with UPC++ DepSpawn and executed on the 32 nodes of our cluster on a 40000×40000 matrix with tiles of 250×250 elements for different tile distributions. Namely, the legend $a \times b$ corresponds to a 2D block-cyclic distribution in which the 32 nodes are organized as a $a \times b$ mesh. The figure also shows for each distribution the performance obtained when the exact match mode is used (right bar in each pair) and when it is not (left bar). The large variation of performance observed for the different tile distributions is due to a combination of factors. One of them is the more balanced work distribution that some definitions of the processor mesh can achieve with respect to others. Another important consideration is the access pattern to remote tiles that each tile distribution gives place to. Just as in hardware processor caches, each combination of data placement and access pattern can give place to widely varying cache hit rates, and given the large

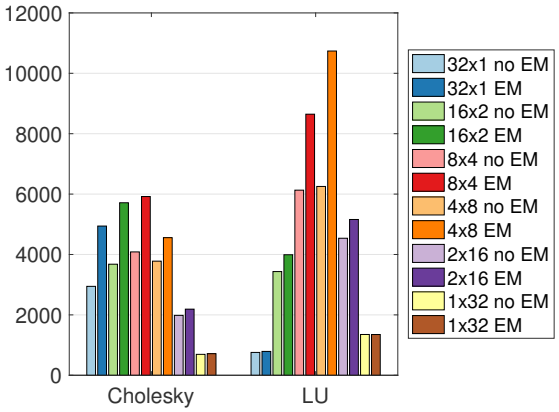

Fig. 4: Performance of Cholesky and LU as a function of the distribution of the tiles and the usage of the exact match mode working on 40000×40000 matrices with 32 nodes

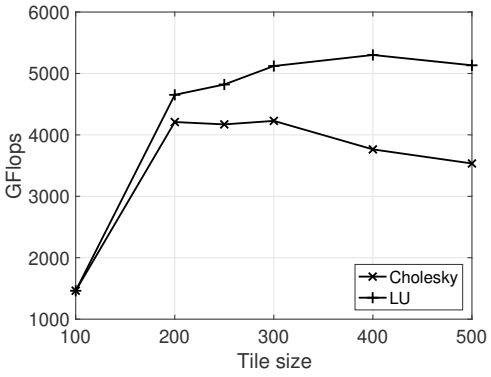

Fig. 5: Performance of Cholesky and LU as a function of the tile size

cost of the remote accesses compared to the cache hits, this results in strong performance variations. Another factor is the ratio of dependent computations that are located in the local node or in a remote node for each distribution. The reason is that the first situation only requires light thread-level synchronizations, while, even if the caches manage to avoid most data transfers, the second situation requires at least inter-node synchronization messages.

As we can see, the most squared distributions provide the best performance for these algorithms. The reason is the predominance, in the case of Cholesky, and exclusiveness, in the case of LU, of communication patterns in which each tile needs to be communicated to nodes that are in the same row or in the same column of the bidimensional mesh as the owning node. As a result, if for example we consider 32 nodes, in the 1×32 and 32×1 distributions the algorithm will often have to communicate tiles to the other 31 nodes, while in 4×8 and 8×4 distributions such communication will only be needed for the $(4-1)+(8-1) = 10$ nodes that are either in the same row or in the same column. Since LU only has patterns of this kind, the effect is stronger in it. Cholesky presents more varied patterns, the most common one being communications across rows. As a result the extreme distribution 32×1 while not optimal, provides a reasonable performance because it places all the tiles in the same row of the matrix in the same node, while conversely 1×32, which partitions block-cyclically the rows, i.e. in chunks of columns, is by far the worst one.

We have also used this figure to illustrate that, although the exact match mode resulted in some overhead in the experiment depicted in Fig. 2, it can be critical for good performance in many situations. In fact in this experiment we can see that it is responsible for 31% and 32% of the performance obtained by the best tile distribution for the Cholesky factorization and the LU decomposition, respectively. In our tests its usage had almost no negative impact on performance, meaning the runtime variations were within 1% in the executions using only one or two nodes. Nevertheless, as the amount of work available per core decreased, its role became more important.

The impact of the tile size, or equivalently the granularity of the tasks and the data transfers, is evaluated in Fig. 5, which shows the performance of these two benchmarks for different tile sizes when applied to a 60000×60000 matrix on eight nodes of the cluster. For each tile size we tried all the possible tile distributions using a constant UPC++ DepSpawn cache of 256 MBytes, and the best result has been used in the figure. We can see that the small 100×100 tile is by far the worst one for all the benchmarks. This makes sense as the smaller the tile size, the larger the number of communications, and although the message size is also smaller, there is a fixed cost per message that cannot be avoided as well as a larger number of tasks and dependencies to manage. As for the best tile size, it is at 400×400 for LU, while Cholesky begins to decline for tiles larger than 300×300. The best tile size can vary depending on the algorithm, the size of the problem to solve, the tile distribution and the number of nodes used.

Figures 6 to 10 show the performance of our implementation and related baselines for the five benchmarks as the number of nodes used grows considering two problem sizes. In most benchmarks the problem size corresponds to the size N of a $N \times N$ square matrix. In the case of Viterbi, the number corresponds to the number of states per row, as the parallelism is proportional to it, 20 observations being considered for both problem sizes. The difficulty in computing the exact flops used in the case of the sparse LU decomposition led us to label the performance based on the speedup with respect to a sequential execution rather than using GFlops. In each node we always use the 24 cores available, thus all the experiments scale from 24 to 768 cores. Also, each point corresponds to the execution with the best combination of configuration factors (mainly tile size and matrix distribution) found for the associated implementation.

In the case of Cholesky and LU, the graphs compare the performance of UPC++ DepSpawn with ScaLAPACK [22] and DPLASMA [23]. ScaLAPACK is the state-of-the-art reference implementation of these routines in distributed memory systems, while DPLASMA is the leading available implementation and it is based on the task-based dataflow-driven PaRSEC runtime [24]. As discussed in Section 6, this runtime is based on an explicit specification by the user of all the dependences found in the application, from which the framework generates a code that once merged with the computational kernels provided by the user, im-

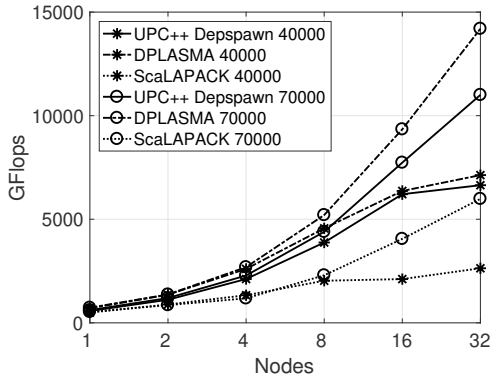

Fig. 6: Performance of the dense Cholesky descomposition benchmark

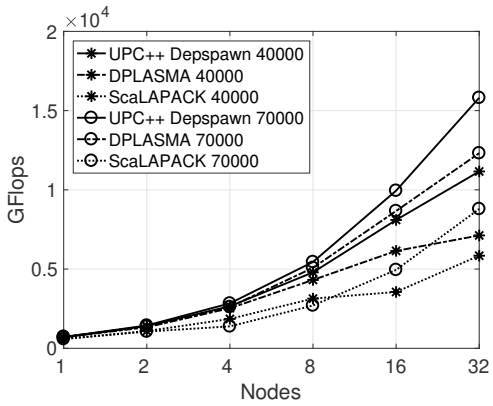

Fig. 7: Performance of the dense LU descomposition benchmark

plements the application in a dataflow fashion. ScaLAPACK and DPLASMA operate on top of MPI, the implementation used in our experiments being OpenMPI, and they use the same OpenBLAS library as UPC++ DepSpawn for the BLAS operations. In addition, DPLASMA, similar to UPC++ DepSpawn, can exploit multithreading within a node, and it relies on hwloc [25] to discover the NUMA architecture available at runtime and optimize placement and task stealing.

Figures 6 and 7 have in common the reduced performance of ScaLAPACK with respect to the other two implementations. There are two reasons for this. First, since ScaLAPACK only relies on MPI, the communications and synchronizations between the processes that run within each node involve more overhead than the multi-threaded solution within shared-memory environments supported by the other approaches. Second, ScaLAPACK is the only one that lacks the fine-grained per-task synchronization enabled by the dataflow philosophy.

Regarding the dataflow implementations, while DPLASMA always offers the best performance for Cholesky, in LU the roles are reversed and UPC++ DepSpawn consistently leads in performance. For example, while DPLASMA is 29% faster than our implementation when applying Cholesky on the largest matrix tested on 32 nodes, in the same LU test UPC++ DepSpawn is 28.4% faster than DPLASMA. Here we must take into account that the off-line static approach taken by DPLASMA greatly reduces the overhead of the runtime compared to UPC++ DepSpawn, which detects dynamically the dependences. Another factor to consider is that the availability at once of all the code dependencies allows DPLASMA to apply optimizations not available in dynamic runtimes like ours, a clear example being the use of collectives for distributing data. This, together with the tighter control of locality enabled by hwloc and the traditional better performance of MPI over PGAS [26] provide DPLASMA the largest performance potential. However, the static compiler-like approach taken by DPLASMA also means that, just as regular compilers do, depending on its internal algorithms and heuristics it may make better optimization decisions for some algorithms than for others. This way, while DPLASMA LU satisfactorily outperforms ScaLAPACK, the result of UPC++ DepSpawn shows that it can be further optimized. The other advantages of UPC++ DepSpawn with respect to static explicit approaches are on the side of usability and generality for two reasons. First, the manual specification of the dependences between tasks typically requires more effort and is more error-prone than just specifying the inputs and outputs of each task. Second, dynamic approaches like ours allow parallelizing irregular codes that cannot be considered by static approaches, this being the case of the sparse LU decomposition considered in this paper.

In the case of the three remaining algorithms the baseline used has been developed by hand combining UPC++ for cluster-level data distribution and parallelism with TBB for node-level parallelism based on threads. Our manual sparse LU implementation follows the same natural strategy as the baseline in [27], which separates with barriers three states in each iteration of the main loop (`lu0`, then `fwd` in parallel with `bdiv`, and finally `bmod`), with one added optimization. Namely, we replaced the global barrier between the `bmod` stage of each iteration and the `lu0` task of the next iteration by a thread barrier only in the node that contains the tile to be processed by `lu0`, as the task can only depend on a `bmod` operation performed in the same node. The scalability seen in Figure 8 is unsurprisingly much lower than in the dense case in Figure 7, as the amount of computation is much smaller, since for both problem sizes we used sparse matrices in which only 4% of the tiles had data. The fact that our baseline exploits thread-level parallelism within each node allows it to offer performance much more comparable to UPC++ DepSpawn than ScaLAPACK for the dense case.

The Gauss-Seidel evaluation corresponds to ten iterations of the application of the method on two matrices of the sizes indicated in Figure 9. Two manual baselines were developed for the Gauss-Seidel method. We started with one based on a barrier per diagonal as suggested in [20], but its performance was unsatisfactory. We therefore implemented an asynchronous alternative by associating a counter to each tile and sending one message to the south and another one to the east whenever a tile is computed in order to increase the counter of the neighbor tiles. When such counter reaches the value one (along the first column and the first row) or two (for all the other tiles), the associated tile can be safely processed using the data required from its west and

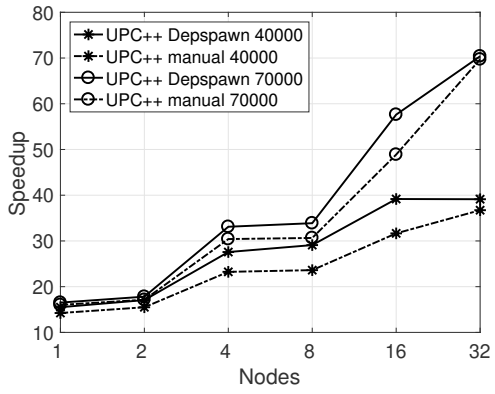

Fig. 8: Performance of the sparse LU descomposition benchmark

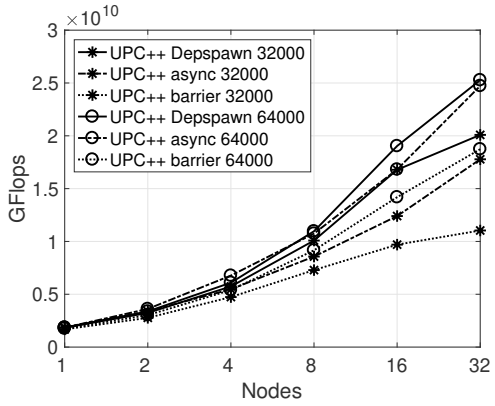

Fig. 9: Performance of the Gauss-Seidel benchmark

north neighbor tiles, when they exist. Upon the processing of the southeast-most tile of the matrix all the ranks are notified, the counters are reset, and a new iteration can begin. Given the large improvement observed in this version, this strategy was enabled for the internal processing of each tile in all the versions. This way our three versions support partitioning the matrix tiles into a second level mesh of subtiles in which several threads can participate in a wavefront from the northwest to the southeast relying on the same strategy based on counters, in this case implemented by atomic variables. It also deserves to be mentioned that since Gauss-Seidel always requires communication of the neighboring data of tiles from north to south and from west to east, using a block-cyclic distribution of the tiles results in bad performance. Thus this is the only algorithm tried in which once the matrix is partitioned into tiles, the tiles are assigned in consecutive bidimensional blocks of them to the nodes, which provides much better performance. The performance shown in Figure 9 corresponds to the best combination of tile size, subtile size and data distribution found for each version. We can see that the noticeably larger complexity of the manual asynchronous version is worth the effort with respect to the baseline based on barrier, but the much simpler UPC++ DepSpawn version outperforms it thanks to the ability to overlap the processing of different iterations of the main loop.

Finally, Viterbi is the algorithm with the most regular and simplest pattern considered. The manual baseline follows the natural strategy of broadcasting the state probabilities computed for an observation in each node to all the other nodes right before the computation of the probabilities associated to the next observation. These probabilities are then computed in a parallel loop in which each node takes care of the probabilities located in tiles in its local memory. After our experience with Gauss-Seidel we also allowed both the manual and the UPC++ DepSpawn version to divide in parallel subtasks the computation of each tile. This had a positive effect on both versions not only because of the further distribution of the computation and parallelization overhead among the cores, but also because this gave implicitly place to a tiling on the data structures used in the computation that enhanced locality. Figure 10 shows the performance for the best tile size and number of subtasks for each version. Despite the manual parallelization fitting very well the application, UPC++ DepSpawn offers on average ∼5.8% more GFlops. The main reason is that even if the size of the parallel tasks is identical, their runtime is not because of the memory accesses within the `find_max_prob` function, which is intensive in memory. Namely, the access costs vary depending on the state of the caches and the location of the executing thread with respect to the processed data in our NUMA nodes with 2 sockets. An additional related reason is that often the task subdivision that gives place to the best execution time does not generate a number of tasks that is a multiple of the 24 cores available per node, which gives place to some load unbalance. Altogether, this gives place to idle cycles for some threads in the local barrier of the manual implementation that do not exist in the UPC++ DepSpawn version thanks to its much larger scheduling flexibility.

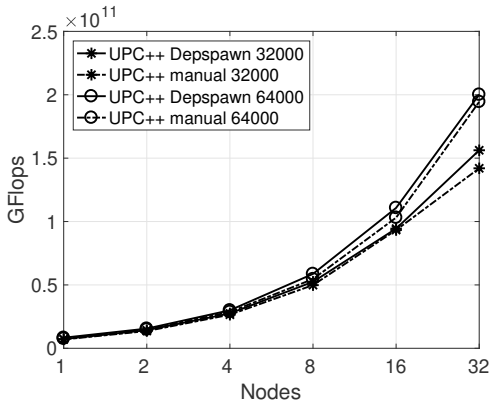

Fig. 10: Performance of the Viterbi benchmark

6 Related work

There have been several proposals to enable the effective development of task parallel algorithms with complex patterns of dependencies on distributed memory systems. Some of them have been ad-hoc implementations addressing a single problem, such as the Cholesky [28] or the LU factorizations [29], the latter one being implemented on top of the UPC language.

As for the generic approaches, they can be classified in two large groups depending on whether the dependencies

are explicitly or implicitly provided. A representative example of the first group is the Parameterized Task Graph (PTG) model [30], which is the basis for the PaRSEC framework [24]. In this model the user expresses the dependencies between tasks using a domain specific language from which a code that supports the dataflow parallel execution of the algorithm expressed is statically generated. On the one hand, the availability of the task graph in advance enables this approach to apply more ambitious optimizations than the dynamic approaches. On the other hand, the obvious shortcomings are the impossibility of expressing applications whose dependencies are irregular or can only be known at runtime, and the difficulty that the discovery and correct conveyance of the dependencies may entail for the user.

Other explicit approaches discover the dependencies during the execution with the help of program objects that the programmer uses to express such dependencies. This is the case of the UPC++ events discussed in Section 4. They are inspired by and share the same limitations as those of Phalanx [31], a programming model for distributed heterogeneous machines whose synchronization relies on them when barriers do not suffice. Single assignment variables [32], [33] and futures, these latter ones having been also discussed in Section 4, offer some degree of implicitness but they are only a natural option for read-after-write data dependencies, and even for the codes where these are the only dependencies to consider, their effective use to express optimized dataflow computations requires additional mechanisms. The reason is that in a dataflow model tasks should start their execution only when their data are ready, while these approaches require the dependent tasks to either poll or block on them until they are ready, i.e., until the dependencies they are associated with are fulfilled. The additional mechanisms needed must thus be able to associate each task with the set of dependence-carrying items it depends on, and to ensure that the task will only be launched for execution when all those futures or synchronization variables are ready. This is the case of the the dataflow objects provided by HPX [18], which supports task based programming on distributed systems on top of C++, or the combination of distributed data-driven futures and `await` clauses for asynchronous tasks in [17], which provide a similar functionality in Habanero-C MPI. Chapel's synchronization variables [32] present similar restrictions with the difference that they can be written multiple times at the cost that each write can only be read by a single synchronization access. Finally, [14] also provides mechanisms to chain the multiple futures on which a computation may depend and only launch it as a callback when they are ready, also avoiding blocking or polling.

The implicit approaches typically only require from the user the annotation of the inputs and the outputs of each task in an apparently sequential code, and sometimes also the registration of the data to use. These proposals identify the dependencies that must be fulfilled to guarantee sequential consistency by considering the data accessed by each task during the sequential order of execution of the tasks with the help of a runtime attached to the application, so that this identification happens during the execution. This implies that all the tasks in the application and their dependences must be analyzed in sequence in order to establish where and when to run each one of them, which can turn into a bottleneck that limits the scalability. In centralized approaches a single process is responsible for this analysis and the management of the TDG, while the other participants act as servers that run the tasks assigned to them. This is the case of ClusterSs [27], which annotates Java code to indicate the usage (input, output or both) of each task parameter or OmpSs [34], which relies on compiler directives similar to OpenMP to provide this and other informations on C/C++ codes. The potential scalability limitations of this approach are ameliorated in [34] by supporting the submission of tasks that can can be further decomposed and parallelized within a node. In decentralized proposals like ours every process performs independently this analysis in order to avoid the potential bottleneck associated to the master and the communications with it. Another decentralized library-based framework in this family is StarPU [35], which relies on handles to pre-registered data to express its dependencies and data distribution. StarPU uses MPI for the distributed executions, which take place on separate processes [36]. This differs from the transparent support of multithreading within each process in our approach, which reduces the communication and TDG handling overheads.

A proposal that supports task-based dataflow execution on distributed systems on top of a novel programming paradigm is Legion [37], which identifies logical regions that can be accessed by different tasks and which are decoupled from the different physical regions to which they may be mapped. The large flexibility of the model and its new concepts requires a non-trivial programming on top of a C++ library API, which led to the development of a new language and compiler [38] that substantially simplify the exploitation of Legion. Another language-based proposal that provides dataflow parallelism on distributed systems is Swift/T [39], which relies on MPI to glue user codes written in traditional languages. A distinctive property of Swift/T is its purely functional nature, which contrary to most proposals, including ours, does not allow tasks to modify their arguments.

Finally, HabaneroUPC++ [40] has in common with our proposal that it extends UPC++ in order to provide a compiler-free library that enhances the usability of this PGAS environment, in this case by supporting rich features for dynamic task parallelism within each process. Namely, it relies on UPC++ for PGAS communication and RPC, and Habanero-C++ for supporting intra-place work-stealing integrated with function shipping.

7 Conclusions

In this paper we have presented UPC++ DepSpawn, a library that enables task-based dataflow programming on distributed memory systems on top of UPC++, which provides an asynchronous PGAS environment in the widely used C++ language. A key characteristic of this proposal is the simplicity of its syntax and semantics, which gives place to programs that look very much like their sequential counterparts, and are thus easy to develop, understand and maintain. In fact the differences basically lie in the usage of the function `upcxx_spawn` to spawn the parallel

tasks and function `upcxx_wait_for_all` to wait for the completion of the dataflow algorithm, coupled with data types provided by UPC++ to represent the distributed data structures and the references to them. Given the inspiration of this library by DepSpawn, a library that offers this programming paradigm in shared memory systems, as well as the increasing number of cores per node, it was also natural that UPC++ DepSpawn offered the same transparent dataflow task-parallelism within each process.

Our implementation contains a large number of optimizations that allow it to outperform by different margins both traditional ScaLAPACK implementations for this kind of systems as well as versions manually developed on top of UPC++ and TBBs, even in algorithms that present large regularity. A comparison with the state of the art DPLASMA library, which combines MPI with threads in a dataflow execution, also shows the interest of our proposal, as each framework clearly leads in performance in different tests. In addition our proposal provides other advantages such as larger usability and support for irregular computations.

As future work we plan to implement more optimizations in UPC++ DepSpawn and study the feasibility of introducing extensions that facilitate the development of a wider range of applications on top of it. A port to a new proposed version of UPC++, once it is stable, is also a possible work. We also plan to release the code under an open-source license together with installation and configuration instructions.

ACKNOWLEDGEMENTS

This research was supported by the Ministerio de Economía, Industria y Competitividad of Spain and FEDER funds of the EU (TIN2016-75845-P), and by the Xunta de Galicia co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2017/04) as well as under the Centro Singular de Investigación de Galicia accreditation 2016-2019 (ED431G/01). We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of their computers.

REFERENCES

[1] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *Proc. 2007 Intl. Workshop on Parallel Symbolic Computation*, ser. PASCO '07, 2007, pp. 24–32.

[2] R. W. Numrich and J. Reid, "Co-array Fortran for Parallel Programming," *SIGPLAN Fortran Forum*, vol. 17, no. 2, pp. 1–31, 1998.

[3] M. G. Burke, K. Knobe, R. Newton, and V. Sarkar, "UPC language specifications, v1.2," Lawrence Berkeley National Lab, Tech. Rep. LBNL-59208, 2005.

[4] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *20th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05, 2005, pp. 519–538.

[5] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007.

[6] K. A. Yelick, S. L. Graham, P. N. Hilfinger, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen, "Titanium," in *Encyclopedia of Parallel Computing*, 2011, pp. 2049–2055.

[7] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory programming toolkit," *Intl. J. of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.

[8] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: A PGAS extension for C++," in *IEEE 28th Intl. Parallel and Distributed Processing Symp. (IPDPS 2014)*, May 2014, pp. 1105–1114.

[9] A. Koniges, B. Cook, J. Deslippe, T. Kurth, and H. Shan, "MPI usage at NERSC: Present and future," in *23rd European MPI Users' Group Meeting*, ser. EuroMPI 2016, 2016, pp. 217–217. [Online]. Available: https://www.nersc.gov/assets/Uploads/MPI-USAGE-at-NERSC-poster.pdf

[10] B. B. Fraguela, "A comparison of task parallel frameworks based on implicit dependencies in multi-core environments," in *50th Hawaii Intl. Conf. on System Sciences*, ser. HICSS'50, 2017, pp. 6202–6211.

[11] C. H. González and B. B. Fraguela, "A framework for argument-based task synchronization with automatic detection of dependencies," *Parallel Computing*, vol. 39, no. 9, pp. 475–489, 2013.

[12] A. Kamil, Y. Zheng, and K. Yelick, "A local-view array library for partitioned global address space C++ programs," in *ACM SIGPLAN Intl. Workshop on Libraries, Languages, and Compilers for Array Programming*, ser. ARRAY'14, 2014, pp. 26:26–26:31.

[13] V. Saraswat, G. Almasi, G. Bikshandi, C. Cascaval, D. Cunningham, D. Grove, S. Kodali, I. Peshansky, and O. Tardieu, "The asynchronous partitioned global address space model," in *First Workshop on Advances in Message Passing*, ser. AMP'10, June 2010.

[14] J. Bachan, D. Bonachea, P. H. Hargrove, S. Hofmeyr, M. Jacquelin, A. Kamil, B. van Straalen, and S. B. Baden, "The UPC++ PGAS library for exascale computing," in *2nd Annual PGAS Applications Workshop*, ser. PAW17, 2017, pp. 7:1–7:4.

[15] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*, 1st ed. O'Reilly, July 2007.

[16] S. Tasirlar and V. Sarkar, "Data-driven tasks and their implementation," in *2011 Intl. Conf. on Parallel Processing*, ser. ICPP'11, Sept 2011, pp. 652–661.

[17] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *2013 IEEE 27th Intl. Symp. on Parallel and Distributed Processing*, ser. IPDPS 2013, May 2013, pp. 712–725.

[18] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *8th Intl. Conf. on Partitioned Global Address Space Programming Models*, ser. PGAS '14, 2014, pp. 6:1–6:11.

[19] D. Bonachea, "Gasnet specification," University of California at Berkeley, Berkeley, CA, USA, Tech. Rep. CSD-02-1207, oct 2002.

[20] V. Gajinov, S. Stipić, I. Erić, O. S. Unsal, E. Ayguadé, and A. Cristal, "Dash: A benchmark suite for hybrid dataflow and shared memory programming models," *Parallel Computing*, vol. 45, pp. 18 – 48, 2015, computing Frontiers 2014: Best Papers.

[21] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, D. Wawrzynek, J.and Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56–67, Oct. 2009.

[22] L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley, "ScaLAPACK: A linear algebra library for message-passing computers," in *8th SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[23] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra, "Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA," in *2011 IEEE Intl. Symp. on Parallel and Distributed Processing Workshops and Phd Forum*, May 2011, pp. 1432–1441.

[24] A. Danalis, H. Jagode, G. Bosilca, and J. Dongarra, "PaRSEC in practice: Optimizing a legacy chemistry application through distributed task-based execution," in *2015 IEEE Intl. Conf. on Cluster Computing*, Sept 2015, pp. 304–313.

[25] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *2010 18th Euromicro Conf. on Parallel, Distributed and Network-based Processing*, Feb 2010, pp. 180–186.

[26] D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez-Tato, R. Doallo, and J. C. Mouriño, "Performance evaluation of MPI, UPC and OpenMP on multicore architectures," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting*. Berlin, Germany: Springer-Verlag, 2009, pp. 174–184.
[27] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, "A high-productivity task-based programming model for clusters," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 18, pp. 2421–2448, 2012.
[28] F. G. Gustavson, L. Karlsson, and B. Kågström, "Distributed sbp cholesky factorization algorithms with near-optimal scheduling," *ACM Trans. Math. Softw.*, vol. 36, no. 2, pp. 11:1–11:25, Apr. 2009.
[29] P. Husbands and K. Yelick, "Multi-threading and one-sided communication in parallel LU factorization," in *2007 ACM/IEEE Conf. on Supercomputing*, ser. SC'07, 2007, pp. 31:1–31:10.
[30] M. Cosnard and M. Loi, "Automatic task graph generation techniques," in *28th Annual Hawaii International Conference on System Sciences*, ser. HICSS'28, vol. 2, Jan 1995, pp. 113–122 vol.2.
[31] M. Garland, M. Kudlur, and Y. Zheng, "Designing a unified programming model for heterogeneous machines," in *2012 Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, ser. SC'12, Nov 2012, pp. 67:1–67:11.
[32] Cray Inc, "Chapel language specification version 0.984," Oct 2017.
[33] J. Breitbart, "A dataflow-like programming model for future hybrid clusters," *Intl. J. of Networking and Computing*, vol. 3, no. 1, pp. 15–36, 2013.
[34] J. Bueno, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta, "Implementing OmpSs support for regions of data in architectures with multiple address spaces," in *27th Intl. Conf. on Supercomputing*, ser. ICS '13, 2013, pp. 359–368.
[35] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.
[36] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Harnessing clusters of hybrid nodes with a sequential task-based programming model," in *Intl. Workshop on Parallel Matrix Algorithms and Applications (PMAA 2014)*, Jul 2014.
[37] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 66:1–66:11.
[38] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, "Regent: A high-productivity programming language for HPC with logical regions," in *Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15, 2015, pp. 81:1–81:12.
[39] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/T: Large-scale application composition via distributed-memory dataflow processing," in *13th IEEE/ACM Intl. Symp. on Cluster, Cloud, and Grid Computing*, May 2013, pp. 95–102.
[40] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, "HabaneroUPC++: a compiler-free PGAS library," in *8th Intl. Conf. on Partitioned Global Address Space Programming Models*, ser. PGAS '14, 2014, pp. 5:1–5:10.

Basilio B. Fraguela received the M.S. and the Ph.D. degrees in computer science from the Universidade da Coruña, Spain, in 1994 and 1999, respectively. He is an associate professor in the Departamento de Enxeñaría de Computadores of the Universidade da Coruña since 2001. His primary research interests are in the fields of programmability, high performance computing, heterogeneous systems and code optimization. His homepage is http://gac.udc.es/~basilio.

Diego Andrade received the M.S. and Ph.D. degrees in computer science from the Universidade da Coruña, A Coruña, Spain, in 2002 and 2007, respectively. He is a Lecturer at the Departamento de Enxeñaría de Computadores, Universidade da Coruña, since 2006. His research interests focuses in the fields of performance evaluation and prediction, analytical modeling, and compiler transformations.