The New UPC++ DepSpawn High Performance Library for Data-Flow Computing with Hybrid Parallelism

Basilio B. Fraguela( ) [0000−0002−3438−5960] and
Diego Andrade[0000−0001−5670−7425]

Universidade da Coruña, CITIC, Grupo de Arquitectura de Computadores.
Facultade de Informática, Campus de Elviña, S/N. 15071. A Coruña, Spain.
{basilio.fraguela, diego.andrade}@udc.es

Abstract. Data-flow computing is a natural and convenient paradigm for expressing parallelism. This is particularly true for tools that automatically extract the data dependencies among the tasks while allowing to exploit both distributed and shared memory parallelism. This is the case of UPC++ DepSpawn, a new task-based library developed on UPC++ (Unified Parallel C++), a library for parallel computing on a Partitioned Global Address Space (PGAS) environment, and the well-known Intel TBB (Threading Building Blocks) library for multithreading. In this paper we present and evaluate the evolution of this library after changing its engine for shared memory parallelism and adapting it to the newest version of UPC++, which differs very strongly from the original version on which UPC++ DepSpawn was developed. As we will see, while keeping the same high level of programmability, the new version is on average 9.3% faster than the old one, the maximum speedup being 66.3%.

Keywords: Data-flow computing · hybrid parallelism · PGAS · runtimes · high-performance computing · task-based parallelism

1 Introduction

The data-flow computing paradigm is very attractive for parallel computing because it does not impose any particular order of execution among tasks, only relying on the necessary satisfaction of the dependencies inherent to them. This approach is particularly interesting in the presence of complex and irregular patterns of dependencies, which require the fine grained synchronization among tasks that it provides in order to achieve good performance. Furthermore, its application can also simplify the development and maintenance of applications if the paradigm is applied through tools that can automatically extract the tasks dependencies from simple information such as the inputs and the outputs of each task. This is the case of the `depend` clause introduced in OpenMP 4.0, which is the most popular approach in this field. In fact, due to the higher complexity of distributed memory programming, with its need to distribute and

This version of the contribution has been accepted for publication, after peer review (when applicable) but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/10.1007/978-3-031-08751-6_55. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms

communicate data among processes, most proposals in this field are restricted to shared memory systems.

In recent years there has been increasing interest in bringing the advantages of data-flow runtimes to distributed memory environments [2,10,11,14,19,21], not only because they allow working on larger problems, but also because their benefits can be even larger in these systems. Indeed, since a data-flow runtime has all the information on the placement of the data and their relation with the tasks and the dependencies, it can also automate the transmission of data among the distributed memories on which the different processors work, a task particularly cumbersome for programmers in the presence of the complex irregular patterns of dependences for which these tools excel.

This paper focuses on one of the most recent proposals in this area, the UPC++ DepSpawn library [14], which allows exploiting this strategy in clusters of multi-core processors. One of the most original features of this library is that it operates on a Partitioned Global Address Space (PGAS) environment. In this model each process has a private space that is exclusive to it as well as a portion of a common shared space that all the processes can access. As expected, this shared space, which is manipulated by means of one-sided accesses, offers much lower latency and larger bandwidth in the portion that has affinity to the accessing process or to other processes that are located in the same node in the cluster compared to the portions located in other nodes. The PGAS model arguably simplifies the development of distributed memory applications compared to the traditional MPI-based paradigm based on totally separate memory spaces.

The name UPC++ DepSpawn stems from the fact that its PGAS environment is provided by UPC++ [23], a library that supports in C++ the abstractions of the UPC (Unified Parallel C) language [12], while most of the runtime for data-flow computing is derived from a shared-memory library called DepSpawn [17]. Both DepSpawn libraries provide native shared memory parallelism on top of the well-known Intel TBB library.

In the past few years the two external components on which UPC++ DepSpawn relies experienced very strong changes, which called for its redesign. First, while UPC++ had not received meaningful updates since its inception in 2014, a radically new version called UPC++ 1.0 [4] was released in 2019. This version both provided new functionalities and removed features provided by the original library, now called v0.1. Although some of the features lost were intensively used by UPC++ DepSpawn, UPC++ v0.1 had lacked support for a long time, while UPC++ 1.0 has a very active community and is strongly supported. This, together with the advantages of the new version validated in [4] called for changing this component of UPC++ DepSpawn. The second very relevant change was the decision by Intel of replacing Intel TBB by a new library called oneTBB [1]. Similar to the case of UPC++, the new version lacks components intensively used by UPC++ DepSpawn, and since Intel TBB is no longer distributed, an update in this area was also required.

This paper describes the modifications performed in the library due to the changes in UPC++ and TBB. This includes a brief description of our experience with the new limitations in UPC++ 1.0, and more particularly in the aspects related to its interaction with multithreading, which have not been discussed in the literature as far as we know. The changes performed are evaluated showing a positive impact on performance in both cases. The final contribution of the manuscript is the public release of UPC++ DepSpawn under a permissive open source license at https://github.com/UDC-GAC/upcxx_depspawn.

The rest of the paper is organized as follows. The related bibliography is discussed in Section 2, which is followed by a description of UPC++ DepSpawn in Section 3. The changes performed in the runtime are described in Section 4, while Section 5 is devoted to the evaluation. Finally, Section 6 is devoted to our conclusions and future work.

2 Related work

There are two main approaches for providing data-flow computing in distributed memory applications, a static one and a dynamic one. The first one requires statically providing the set of tasks and dependencies among them in some domain-specific language (DSL). This is fed to a tool that generates a code that supports the parallel execution of the tasks described respecting the dependencies specified and performing the data communications required. Since the discovery of the dependencies is performed off-line and the code generator has all the information on the whole program, this approach should lead to minimal overheads during the execution and the best possible planning and discovery of potential optimizations. On the other hand, there are two shortcomings to this strategy. One is the impossibility of dealing with dependencies that can be only known at runtime, and sometimes, depending on the tool, with irregular dependencies, even if they can be known statically. The second problem is the need to identify and correctly state all the dependences in the code, which can be cumbersome. The static strategy is followed by the Parameterized Task Graph (PTG) model [13], on which the DAGuE/PaRSEC framework [10] is based. The most notable software developed on this framework is DPLASMA [9], the leading implementation for dense linear algebra algorithms in distributed memory systems.

As for the dynamic approaches, which can deal with dynamic and irregular dependencies, we can break them in two families. The first one relies on a wide array of mechanisms to define and enforce dependencies among tasks thanks to their manipulation during the execution of the program. This is the case of locks, full/empty bits, synchronized blocks, and futures, among others.

The last alternative consists of writing an apparently sequential version of the algorithm in which the parallel tasks are identified, and their inputs and outputs are labelled. Then, during the execution of this instrumented code in a parallel environment, a runtime finds the tasks to run and their dependencies, and it schedules the executions and performs the necessary data transfers in such a way that all the dependencies are met. In our opinion this is the most attractive strategy from the point of view of simplicity and elegance, but it is also the one with the higher expected overheads, as it is the most demanding on the runtime. We identify two main alternatives for implementing this approach. The first one, tested in [11,20], consists of running this code that provides the tasks and dependencies in a single process, which acts as master or client, while all the other processes involved act as slaves or servers, executing the tasks under its control. SYCL [18], which can manage multiple distinct memories and devices under this model from a single host application can also be classified in this group. The second strategy, followed by UPC++ DepSpawn and the MPI-based StarPU [2,3], executes the code in parallel in all the processes involved, so that they all have a view of the task graph and they agree on which process runs what and when. Another key difference of UPC++ DepSpawn with respect to StarPU in addition to the PGAS approach is the transparent exploitation of thread-level parallelism within each process.

It deserves to be mentioned that there are projects that involve both complex runtimes and advanced compilers. For example, a project that started following a library-based dynamic approach but later developed a new language and compiler in order to be able to apply more optimizations and remove runtime overhead was [5], the new alternative being presented in [19]. Another alternative based on a new language that achieves parallelism through data-flow is [21], which is characterized by focusing on workflows and being purely functional.

Finally, although [22] deals with a superscalar scheduler of PLASMA restricted to shared memory, it is strongly related to our work, since it describes the change of this scheduler from a proprietary solution to the OpenMP standard. Our work, however involves changes both in the shared and the distributed memory aspects of our library.

3 Data-flow computing with UPC++ DepSpawn

Since UPC++ DepSpawn integrates in applications written using UPC++ [23], this section first presents the basics of the latter before introducing UPC++ DepSpawn, so that its syntax and semantics can be understood.

3.1 UPC++

UPC++ [23] supports the development in C++ of parallel programs following the PGAS paradigm and more concretely the concepts of the UPC language [12] but without the need for a new language and related compiler. In UPC++ programs the standard datatypes define data that is located in the local private memory of the process, while the UPC++ types `shared_array<T,B>` and `shared_var<T>` define, respectively, unidimensional arrays of elements and scalars of type `T` located in the shared memory that all the processors can access. By default, when the optional parameter `B` is not provided, the distribution of the elements of the arrays is purely cyclic, so that if there are P processes, element i has affinity to process $i \bmod P$, which means that it is placed in the portion of the shared space associated to that process. The template argument `B` allows changing the distribution to block cyclic, the size of the block being `B`. Relatedly, just as C++ has pointer and reference types, UPC++ provides `global_ptr<T>` and `global_ref<T>` that play the same role, respectively, for the data item placed in the global shared space.

Just as UPC, UPC++ follows a SPMD style, which is enabled by functions that provide a unique identifier to each process as well as the number of processes involved. The library also provides other interesting features, but they are not required to write UPC++ DepSpawn programs.

3.2 UPC++ DepSpawn

UPC++ DepSpawn provides data-flow computing on top of UPC++ in an elegant way that requires very little effort. This procedure can be summarized in three steps. The first one involves writing a sequential version of the code to parallelize where each task is encapsulated as a function that expresses all its dependencies with other tasks through its parameters. This library does not require to label each function parameter in order to inform the runtime on its usage by the function. Rather, the metaprogramming capabilities of C++ are exploited in order to analyze the data type of each formal parameter and infer from it its nature. This way, data passed by value or constant reference are assumed to be exclusively inputs, as the function can read but not modify them, while data passed by non-constant reference is assumed to be both an input and an output. In the case of UPC++ DepSpawn, the references will be object of the template class `global_ref<T>`, where `T` will be a `const` type in the case of constant references. Then, the second step involves rewriting the function invocations from the usual `f(a, b, ...)` notation to `upcxx_spawn(f, a, b, ...)`. Finally, an invocation to `upcxx_wait_for_all()` must be inserted at the point where the algorithm finishes and we want to wait for the results.

Listing 1.1 exemplifies the usage of UPC++ DepSpawn for the implementation of a Cholesky decomposition on a shared array `A` of `N`×`N` tiles. The code uses the macro `_` to map from a natural bidimensional indexing to the linear indexing required by the unidimensional `shared_array` class provided by UPC++. In addition to the main algorithm, the code includes the definition of the function `dgemm`, responsible for making a matrix product between tiles. From the data types of its parameters UPC++ DepSpawn can infer that `dest` is both read and written by the function, while `a` and `b` are only inputs.

As we can see, an algorithm written with UPC++ DepSpawn looks like a sequential implementation, making the development and the maintenance very easy. The code, however, must be run by all the UPC++ processes. This allows each process to learn the set of tasks and dependencies in the code and to schedule the executions making sure that all the dependencies are fulfilled. Each process also independently infers who will be responsible for running each task based on the location of its arguments following a strategy explained in [15]. The main topic introduced in this latter publication is however a variation of `upcxx_spawn` called `upcxx_cond_spawn`. This function takes as first argument a boolean that informs the runtime on whether the task is involved in the part of the global task dependency graph (TDG) that pertains to the tasks executed

```
shared_array<Tile> A(N * N);
#define _(i, j) ((i) * N + (j))
...

void dgemm(global_ref<Tile> dest, Tile a, Tile b) {
  // dest = dest + a x b
}

for(i = 0; i < N; i++) {
  upcxx_spawn(potrf, A[_(i,i)]);
  for(r = i+1; r < N; r++) {
    upcxx_spawn(trsm, A[_(i,i)], A[_(r,i)]);
  }
  for(j = i+1; j < N; j++) {
    upcxx_spawn(dsyrk, A[_(j,i)], A[_(j,j)]);
    for(r = j+1; r < N; r++) {
      upcxx_spawn(dgemm, A[_(r,j)]), A[_(r,i)]), A[_(j,i)]));
    }
  }
}

upcxx_wait_for_all();
```

Listing 1.1. Cholesky factorization in UPC++ DepSpawn

in this process or not. This can reduce the overhead of the runtime by quickly dismissing tasks that the process can safely ignore.

4 An improved runtime

As explained in Section 1, UPC++ DepSpawn was deeply changed due to the important recent novelties in the Intel TBB and UPC++, affecting its mechanisms to exploit both shared and distributed memory parallelism. This section discusses the decisions taken and the modifications performed in both fields.

4.1 Shared memory parallelism migration

DepSpawn [17] and UPC++ DepSpawn [14] relied on the Intel TBB not only for the creation of threads and parallel tasks, but also for the scheduling of the ready tasks and the definition of many synchronization objects. At this point we must bear in mind that when DepSpawn [17] was written the C++11 standard, with its support for threading and synchronization, was not fully available everywhere, and relying on the Intel TBB solved this. Nowadays it no longer makes sense to not rely on the C++11 facilities whenever possible.

In 2020 Intel dropped the Intel TBB library as it had been known, and it integrated it into its oneAPI initiative, which goes well beyond the capabilities we need. The new version is called oneTBB [1], and while the high level functionalities are mostly the same, the low level tasking API was removed. Since the DepSpawn libraries relied on this API, a change was needed. One possibility was to adapt our libraries to the higher level API of oneTBB. Another option was to move the threading and tasking on top of C++11, designing ourselves the thread and task pools, task packaging and scheduling. This latter alternative was attractive because (a) removing the oneTBB dependency simplifies the deployment of the library, (b) being provided by a company, the portability of oneTBB is much more compromised than that of the C++11 standard, and (c) we suspected that while TBB/oneTBB provide very good performance, we could implement a runtime suited to our needs with even better performance for two reasons. The first one is that the generic nature of the TBB leads to generic APIs, with inherent encapsulation costs, and to a runtime that may be more complex than what we actually need. As a result, by writing our own implementation we could avoid TBB-related overheads. The second and most important reason is the lack of control on the order of execution of the ready tasks in TBB. This is in general an advantage of TBB that allows it to apply complex work-stealing strategies in order to balance the workload among its threads. However, a data-flow computing engine is a specific context in which we have a subset of tasks that are part of a general graph that are ready to be executed. Our intuition is that in this situation, if we have different ready tasks that can be run at a given point, it is better to run the oldest tasks before the newest ones. One reason is that this tends to fulfill older rather than newer dependencies in the TDG, resulting in a more balanced progress through the TDG. This results in more locality in the triggering of dependencies, so that older tasks on which newer tasks depend tend to complete their execution before.

As a result of this analysis, the threading, tasking and scheduling in DepSpawn and UPC++ DepSpawn were rewritten to be based on C++11 instead of the TBB. Our task objects are designed for maximum simplicity and minimum overhead, and just as other objects frequently built and destroyed, they have their own memory pools. Contention was minimized by basing almost every synchronization on atomic operations, resulting in a nearly lockless design. As for our new threading runtime, it is based on a pool of C++11 threads that extracts ready tasks from a thread-safe FIFO queue. The queue has a fixed maximum size in order to reduce memory management overheads and minimize insertion time. When this maximum is reached, the pushing thread runs tasks from the head until there is space for the new task. This policy also avoids storing too many ready tasks without actually participating in their execution.

The queue ensures that tasks are executed in the order in which they are ready, which is something we could not achieve directly with TBB. This helps follow the policy of executing older tasks before newer ones because, as a general trend, the older a task the earlier it tends to be ready for execution. However, our queue does not further enforce the execution of older tasks before newer tasks by acting as a priority queue. That is, it does not reorder its contents by sorting the ready tasks it holds according to their id, which is an integer that is smaller the older the task is. A critical reason for this is that DepSpawn, which provides the new threading system to UPC++ DepSpawn, does not support this id in its tasks, as its original purpose in UPC++ DepSpawn is to uniquely identify a task when its completion is notified to another process. A second reason is that with the current design, push and pop operations are extremely fast thanks to being based on very cheap non-blocking atomic operations and our decision to use a fixed size queue. Enforcing an order among the tasks stored would have implied locks, explorations of the structure upon every push and/or pop, and in general a more complex data structure with a much more expensive management and potential to become a bottleneck for the participating threads.

Two extra steps with low cost were taken to promote the ordered execution of the tasks in each process. First, whenever a task finishes, it releases its dependencies in order, i.e., from the oldest to the newest dependent task, so that tasks tend to enter in order the queue of ready tasks. Second, whenever our runtime receives notifications of termination of remote tasks, they are sorted according to their id so that older tasks release their dependencies before newer tasks.

4.2 Distributed memory parallelism migration

The new UPC++ 1.0 [4] largely differs from the version 0.1 of 2014 on which UPC++ DepSpawn was originally written. In fact the new UPC++ is not an update to the old one, but a complete rewrite that is totally different both at the API and implementation levels. They do not even rely on the same low level libraries, as [23] used GASNet [6] for communications, while UPC++ 1.0 uses the new GASNet-EX [7] communication library for exascale.

Unfortunately, UPC++ 1.0 dropped elements that were used in UPC++ DepSpawn programs such as the distributed arrays or the global references. While UPC++ DepSpawn does not use itself global arrays, as seen in Listing 1.1 this abstraction is convenient for writing global data-flow programs. Global references were more related to UPC++ DepSpawn because they mimicked exactly the same semantics as regular references provide to DepSpawn in shared memory.

The good news is that UPC++ 1.0 offers all the elements needed to build and manipulate globally shared data, and thus any desired components can be built on top of it. We felt that global arrays are a nice abstraction that has been very successful, being widely implemented both by languages [12] and libraries [16], and thus we wrote our own global array class. Our class improves upon the one provided by [23] in several ways, mostly by being bidimensional, supporting not only full but also upper and lower triangular matrices to save space, and providing generic 2D block cyclic distributions and well as simple row and column cyclic distributions or even placement in a single process. Let us notice however that this type is provided for convenience and users are free to obtain the pointers to the global data they want to manipulate from any data structure they wish.

Contrary to global arrays, global references played a direct role in UPC++ DepSpawn. As explained in Section 3.2, the library analyzes the type of the formal parameters of the task functions and it infers from it the kind of usage that the function can make of the associated argument. Namely, arguments passed by value or constant reference i.e., parameters of type `T` or `global_ref<const T>`, are read-only inputs, while values passed by non-constant reference, that is, with parameters of type `global_ref<T>`, can be both inputs and outputs.

In this case we felt that forcing programmers to use a class written by us that replaced the now unsupported `global_ref` class was not the best option, particularly when we could just assign its meaning to global pointers, which do exist in UPC++ 1.0. This way we decided that when a formal parameter to a spawned function is a global pointer to a constant type, i.e. of the form `global_ptr<const T>`, the implication is that the function will read the associated data through the pointer, but it will not change it, and thus it will be a read-only input. Similarly, global pointer parameters that point to a non-constant type indicate that the function can both read and write the data they point to.

Besides these external changes, the migration to UPC++ 1.0 also implied internal changes in the runtime. Replacing the communication and synchronization mechanisms of [23] by those of [4] was relatively straightforward except for three details. The first one is related to thread safety, which is critical to our runtime given not only its multi-threaded nature, but also the aggressive level of asynchrony and optimization applied. Both versions of UPC++ can be compiled in a thread-safe mode, but while this suffices to ensure thread safety in UPC++ v0.1 this is not the case with the new UPC++. Indeed, as indicated in the UPC++ 1.0 specification [8], important elements such as futures and promises are not thread-safe. As a result, proper measures must be taken to safely manage these objects in a runtime like ours. Our strategy relied on minimizing as much as possible the use of these elements and trying to restrict the use of the ones used to a single thread. Finally, the few futures for which it is beneficial to allow several threads to operate on them are protected by mutexes.

The second issue is related to a new limitation on the activities that user code is allowed to perform when it is executed during calls to UPC++. This user code are the RPCs and the callbacks, as they are only executed when the UPC++ runtime makes what is termed as user-level progress, as opposed to the internal level progress, which cannot be observed by the application. The main way in which user-level progress takes place in UPC++ 1.0 is by invoking the new function `progress`. However, there are other very relevant and typical ways of entering this state such as waiting for a future to be ready. In fact this concept is so important that the UPC++ 1.0 specification [8] informs on the kind of progress that every single function can perform.

The new limitation that constitutes the second problem is that attempts to enter user-level progress in user code that is already run within a user-level progress context may result in a no-op every time. This way, for example, if during a callback or RPC our code tries to wait for a future to be ready, since this implies trying to make user-level progress in code already executed in that mode, the application can, and in fact will, very probably, hang. This situation happened in the UPC++ DepSpawn runtime because this limitation did not exist in [23]. As a result, the runtime was rewritten to move every attempt to enter UPC++ user-level progress out of RPCs and callbacks. This implied the creation of new task queues where tasks that required this kind of progress found during user-level progress were stored, so that they can be safely executed after leaving this state.

A third particularity of the new UPC++ that required special consideration were the new *persona* objects and the notification affinity for futures [8]. This latter concept is derived from the fact that in UPC++ 1.0 each future is associated to a single persona, each persona can only be associated to a single thread at a time, and only that thread is able to signal the completion of the futures associated to that persona. Thus additional care must be taken in the management of futures shared by several threads. In addition, only the thread that owns a special persona called *master persona* can perform important operations, such as executing RPCs. In our runtime any thread must be able to run these tasks if needed. Thus, the new UPC++ DepSpawn runtime also ensures the properly synchronized management of the master persona among all its threads.

5 Evaluation

The experiments have been run in a cluster with 32 nodes, each one consisting of 2 Intel Xeon E5-2680 v3 at 2.5 GHz and 128GB of memory. Since each processor has 12 cores, the experiments use up to 768 cores, which are configured with hyperthreading disabled, so that at most one thread can be run per core. Codes were compiled with g++ 6.4 and optimization level O3, release 2021.3.0 of UPC++ 1.0 being the one used.

The evaluation relies on the right-looking Cholesky factorization in Listing 1.1, the LU decomposition, and the Gauss-Seidel stencil. The BLAS computational kernels benchmarks rely on the OpenBLAS library version 0.3.1 using a single thread. As a result, all the shared memory parallelism comes from the exploitation of our runtime, which is configured to use a single process per node with 24 threads, one per core. Also, all the UPC++ DepSpawn codes apply the optimization presented in [15], which is thus the baseline performance.

Figures 1 to 3 present the performance achieved by the old and the new version of UPC++ DepSpawn for each one of the three benchmarks. Each graph is a strong scaling study that considers two problem sizes and represents the performance as a function of the number of nodes used. In addition, the Cholesky and LU graphs include the performance obtained by the current state-of-the-art implementation, provided by DPLASMA [9] and relying on the same OpenBLAS library for the computational kernels. As explained in Section 2, DPLASMA follows a data-flow approach that relies on the Parameterized Task Graph (PTG) model [13] based on a static description of the algorithm. Another difference is that DPLASMA uses MPI for the communications, OpenMPI 2.1.1 being used in the experiments. Every performance point, both for the UPC++ DepSpawn and the DPLASMA measurements, corresponds to the best combination of tile size and matrix mapping on the number of nodes considered following a block cyclic distribution.

As we can see, the improvement is very noticeable in Cholesky. The new version always outperforms the old one, and it allows scaling to 32 nodes the small

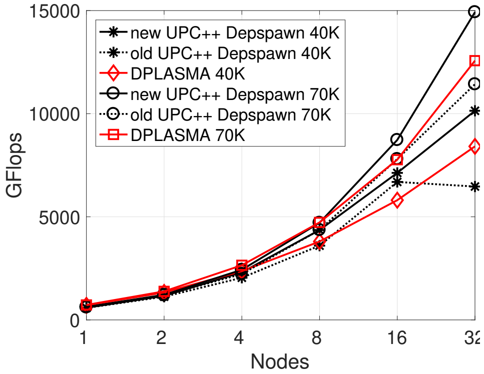

Fig. 1. Performance of the Cholesky descomposition benchmark

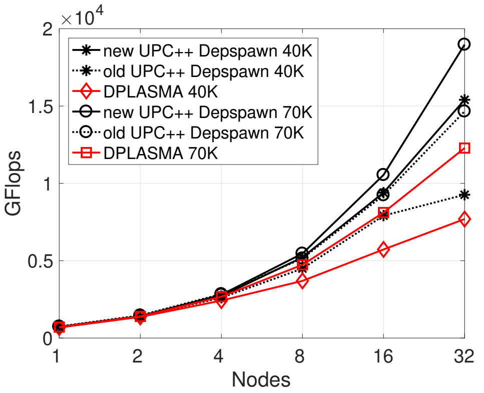

Fig. 2. Performance of the LU descomposition benchmark

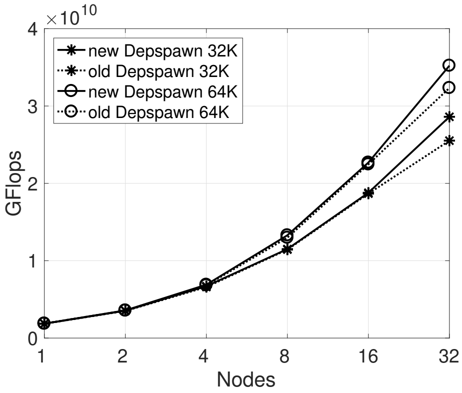

Fig. 3. Performance of the Gauss-Seidel benchmark

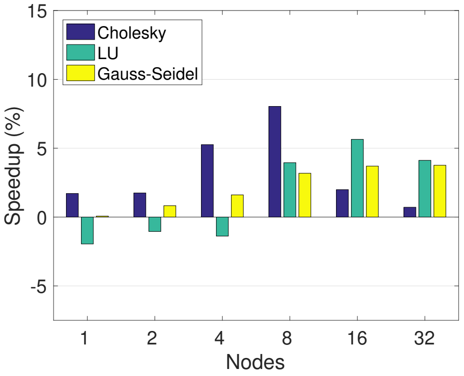

Fig. 4. Speedup of the new multithreading runtime with respect to the old one

problem size, which did not scale above 16 nodes with the old version. The improvement also allows UPC++ DepSpawn to clearly outperform DPLASMA for both problem sizes, the difference growing as the number of nodes used increases. In LU, the old UPC++ DepSpawn always scaled reasonably and outperformed DPLASMA, although we can see that the performance growth was slowed down for the small problem size when 32 nodes were used in the evaluation. The new version also systematically increases the performance, particularly as the number of nodes grows, and solves this scalability issue. Finally, Gauss-Seidel is a very lightweight computation with a pattern much simpler than Cholesky and LU. The performance of both versions is basically the same up to 16 nodes, but the tendency of the new version to provide better performance as the number of nodes grows is finally seen when 32 nodes are used.

Altogether, the maximum speedup for the new version with respect to the old one is 66.3%, which happens for the small problem size of LU at 32 nodes thanks to the increase in performance from 9.27 TFlops to 15.41 TFlops. The average speedup across the Cholesky, LU and Gauss-Seidel tests is 13.9%, 11.5% and 2.4%, respectively, the global average being thus a noticeable 9.3%.

In order to measure the relative influence on the new performance obtained of the changes performed in the multithreading engine and the adaptation to the new UPC++, a version of the old UPC++ DepSpawn based on UPC++ v.01 that relied on the new C++11 runtime for multithreading was developed. Figure 4 shows the speedup that it obtains compared to the original one based on Intel TBB. While the new runtime can lose up to 2% speedup, it can achieve improvements of up to 8%, the average across all the experiments being 2.3%. As a result, roughly one fourth of the improvement of the new runtime comes from the changes in the multithreading environment.

LU is the most computationally demanding benchmark, and the number of tasks per process is very large when few nodes are used, making less important the control of the order of execution of tasks in that situation compared to the smart work-stealing facilities of the Intel TBB. In the other two codes this control is more clearly beneficial. This is particularly true in Gauss-Seidel, which is the benchmark with the fewest computational needs and whose few tasks are more sensitive to the order of execution. In Cholesky, which sits in between LU and Gauss-Seidel in terms of computational needs, the benefit of the new runtime is still positive but drops when more than 8 nodes are used. This probably happens because after this point the smaller number of tasks per node makes it more unlikely that there are delays in the execution of tasks that are critical to trigger subsequent dependent tasks. The difference with respect to Gauss-Seidel resides in the fact that the latter has so few live tasks in each moment that any delay in the execution of tasks in the critical path will surely lead to idle cores, while Cholesky has a reasonable number of tasks to keep busy the cores used.

6 Conclusions

UPC++ DepSpawn is a recently introduced library for exploiting task parallelism in distributed memory systems. The tasks are executed under an an efficient scheduler that respects the implicit dependencies among them extracted by the library from their arguments and formal parameter types. In this paper the library largely evolved, changing both its shared and distributed memory components. This way, multithreading was moved from Intel TBB to a manually managed thread pool and related task queue, while in the PGAS component the obsolete and no longer supported UPC++ v0.1 library that relied on GASNet was replaced by the new UPC++ 1.0 based on GASNet-EX. This critical redesign of the implementation only involved minor changes in the API, not impacting the ease of use. Performance, however, noticeably improved, reaching maximum speedups of 66.3% when using 768 cores, the average speedup across varying number of cores, benchmarks and problem sizes tested being 9.3%.

The manuscript also describes the challenges that the new limitations imposed by UPC++ 1.0 on multithreading pose for our library, which did not exist in the older version of UPC++, and how we addressed them.

Furthermore, the library has been now made publicly available under an open source license at https://github.com/UDC-GAC/upcxx_depspawn.

As future work we want to evaluate UPC++ Depspawn in conjunction with heterogeneous devices typically found in clusters such as GPUs. Developing distributed BLAS and/or LAPACK libraries on top of UPC++ Depspawn also seems a useful contribution given the results observed.

Acknowledgements This research was supported by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00 / AEI/ 10.13039/501100011033), and by the Xunta de Galicia co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2021/30). We acknowledge also the support from the Centro Singular de Investigación de Galicia "CITIC", funded by Xunta de Galicia and the European Union (European Regional Development Fund- Galicia 2014-2020 Program), by grant ED431G 2019/01. Finally, we acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of their computers.

References

1. oneAPI Threading Building Blocks (oneTBB). https://github.com/oneapi-src/oneTBB, accessed: 2022-03-26
2. Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., Thibault, S.: 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)
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011)
4. Bachan, J., Baden, S.B., Hofmeyr, S., Jacquelin, M., Kamil, A., Bonachea, D., Hargrove, P.H., Ahmed, H.: UPC++: A high-performance communication framework for asynchronous computation. In: 2019 IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS). pp. 963–973 (May 2019)
5. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: Intl. Conf. on High Performance Computing, Networking, Storage and Analysis. pp. 66:1–66:11. SC'12 (2012)
6. Bonachea, D.: Gasnet specification. Tech. Rep. CSD-02-1207, University of California at Berkeley, Berkeley, CA, USA (oct 2002)
7. Bonachea, D., Hargrove, P.H.: GASNet-EX: A high-performance, portable communication library for exascale. In: Languages and Compilers for Parallel Computing. pp. 138–158. LCPC'19 (2019)
8. Bonachea, D., Kamil, A.: UPC++ v1.0 Specification, Revision 2021.3.0. Tech. Rep. LBNL-2001388, Lawrence Berkeley National Laboratory (March 2021)
9. Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A.,

Dongarra, J.: 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. pp. 1432–1441 (May 2011)

10. Bosilca, G., Bouteiller, A., Danalis, A., Hérault, T., Lemarinier, P., Dongarra, J.: DAGuE: A generic distributed DAG engine for high performance computing. Parallel Computing **38**(1-2), 37–51 (2012)
11. Bueno, J., Martorell, X., Badia, R.M., Ayguadé, E., Labarta, J.: Implementing OmpSs support for regions of data in architectures with multiple address spaces. In: 27th Intl. Conf. on Supercomputing. pp. 359–368. ICS'13 (2013)
12. Burke, M.G., Knobe, K., Newton, R., Sarkar, V.: UPC language specifications, v1.2. Tech. Rep. LBNL-59208, Lawrence Berkeley National Lab (2005)
13. Cosnard, M., Loi, M.: Automatic task graph generation techniques. In: 28th Annual Hawaii International Conference on System Sciences. HICSS'28, vol. 2, pp. 113–122 vol.2 (Jan 1995)
14. Fraguela, B.B., Andrade, D.: Easy dataflow programming in clusters with UPC++ DepSpawn. IEEE Transactions on Parallel and Distributed Systems **30**(6), 1267–1282 (June 2019)
15. Fraguela, B.B., Andrade, D.: High-performance dataflow computing in hybrid memory systems with UPC++ DepSpawn. The Journal of Supercomputing **77**(7), 7676–7689 (July 2021)
16. Fraguela, B.B., Bikshandi, G., Guo, J., Garzarán, M.J., Padua, D., von Praun, C.: Optimization techniques for efficient HTA programs. Parallel Computing **38**(9), 465–484 (Sep 2012)
17. González, C.H., Fraguela, B.B.: A framework for argument-based task synchronization with automatic detection of dependencies. Parallel Computing **39**(9), 475–489 (2013)
18. Reyes, R., Brown, G., Burns, R., Wong, M.: Sycl 2020: More than meets the eye. In: Intl. Workshop on OpenCL. IWOCL'20 (2020)
19. Slaughter, E., Lee, W., Treichler, S., Bauer, M., Aiken, A.: Regent: A high-productivity programming language for HPC with logical regions. In: Intl. Conf. for High Performance Computing, Networking, Storage and Analysis. pp. 81:1–81:12. SC'15 (2015)
20. Tejedor, E., Farreras, M., Grove, D., Badia, R.M., Almasi, G., Labarta, J.: A high-productivity task-based programming model for clusters. Concurrency and Computation: Practice and Experience **24**(18), 2421–2448 (2012)
21. Wozniak, J.M., Armstrong, T.G., Wilde, M., Katz, D.S., Lusk, E., Foster, I.T.: Swift/T: Large-scale application composition via distributed-memory dataflow processing. In: 13th IEEE/ACM Intl. Symp. on Cluster, Cloud, and Grid Computing. pp. 95–102 (May 2013)
22. YarKhan, A., Kurzak, J., Luszczek, P., Dongarra, J.: Porting the PLASMA numerical library to the OpenMP standard. Int. J. Parallel Program. **45**(3), 612–633 (2017)
23. Zheng, Y., Kamil, A., Driscoll, M.B., Shan, H., Yelick, K.: UPC++: A PGAS extension for C++. In: IEEE 28th Intl. Parallel and Distributed Processing Symp. (IPDPS 2014). pp. 1105–1114 (May 2014)