This is a post-peer-review, pre-copyedit version of an article published in The Journal of Supercomputing. The final authenticated version is available online at: http://dx.doi.org/10.1007/s11227-020-03607-1

High performance dataflow computing in hybrid memory systems with UPC++ DepSpawn

Basilio B. Fraguela · **Diego Andrade**

Pubished online: 8 January 2021

Abstract Dataflow computing is a very attractive paradigm for high performance computing, given its ability to trigger computations as soon as their inputs are available. UPC++ DepSpawn is a novel task-based library that supports this model in hybrid shared/distributed memory systems on top of a Partitioned Global Address Space (PGAS) environment. While the initial version of the library provided good results, it suffered from a key restriction that heavily limited its performance and scalability. Namely, each process had to consider all the tasks in the application rather than only those of interest to it, an overhead that naturally grows both with the number of processes and tasks in the system. In this paper this restriction is lifted, enabling our library to provide higher levels of performance. This way, in experiments using 768 cores the performance improved up to 40.1%, the average improvement being 16.1%.

1 Introduction

Developing parallel applications is a complex task, particularly in distributed memory systems. This has led to a wide range of proposals to enhance the programmability of these systems. A notable example is the abstraction of Partitioned Global Address Space (PGAS)[24], which provides both private separate memory spaces to each process as well as a common space accessible by all the processes in an application. This latter space allows manipulating what is actually distributed memory as if it were a shared space, thus simplifying the development. However, given the large cost of remote accesses in distributed memory, this space is partitioned so that each portion of the shared space is local to a different process. This characteristic allows programmers to

Basilio B. Fraguela · Diego Andrade
Grupo de Arquitectura de Computadores, Universidade da Coruña, A Coruña, Spain
E-mail: {basilio.fraguela,diego.andrade}@udc.es

reason on locality in order to minimize remote accesses. This paradigm has been implemented both on languages [19,8,10,9,25] and libraries [18,26,3].

Also, while many applications present relatively simple patterns of parallelism, in others the patterns of dependences between tasks may be highly irregular and even impossible to predict in advance. In addition, even in situations in which the patterns are regular, the ability to trigger a computation as soon as its specific dependences have been satisfied, rather than waiting for a global synchronization, can enhance the performance of an application. Therefore there is also much research on enhancing the programmability of parallel applications with complex patterns of dependencies, which can also benefit applications with simpler patterns. Although this problem can be tackled following a large number of approaches, a particularly promising one is dataflow computing, in which a runtime manages the tasks that constitute the parallel application ensuring that they are run only when their dependencies have been satisfied. One of the most convenient ways to provide those dependencies consists in labeling the inputs and outputs of each task and specifying the order in which such tasks would be run in a sequential program. With this information the runtime can infer the task dependency graph (TDG) that enables the dataflow parallel execution. While most efforts in this area have targeted shared memory systems, the more challenging distributed memory environments have also been considered [22,1].

A recent library that conjugates PGAS and dataflow computing is UPC++ DepSpawn [15], which builds upon UPC++ [26], a library-based implementation of the Unified Parallel C (UPC) [8] language. UPC++ DepSpawn requires that the parallel tasks are expressed as functions, so that each task consists in the execution of a user-provided function. Then, the library examines the types of the formal parameters of each task function in order to infer which task arguments are only inputs to the function and which ones can be modified by it. This, coupled with the invocation of all the tasks in the application in sequential order allows the library to build the TDG and schedule the task according to their dependencies. Another very interesting feature of the library is that it can also schedule parallel tasks among multiple threads within each process. This makes it naturally fit for the hybrid shared/distributed memory systems that are so common in current clusters and supercomputers since the advent of multi-core processors.

In this paper we enhance UPC++ DepSpawn by raising a very critical restriction. Namely, in [15] each process had to build the whole TDG in order to cooperate successfully with the other processes, even when each process actually only needs to know about the subset of the TDG related to the tasks it is in charge of. Removing this restriction is particularly important for scalability, as the larger the number of processes considered, the smaller the fraction of a TDG that is of interest to each process, and thus the bigger is the waste of resources associated to the construction and analysis of the whole TDG in each process.

The rest of this manuscript is organized as follows. First, Section 2 introduces UPC++ and the library. Then, Section 3 discusses the limitation of the

library considered in this paper and how it was tackled. This is followed by an evaluation in Section 4 and a discussion of the related work in Section 5. The last section of the paper is devoted to our conclusions and future work.

2 UPC++ DepSpawn

Since UPC++ DepSpawn works within an UPC++ application, UPC++ will be briefly introduced before delving into the details of our library.

As indicated in Section 1, UPC++ [26] is a C++ library that provides the facilities and semantics of the UPC [8] language and a number of additional ones. UPC++ applications consist of multiple processes that can run in a distributed memory environment following a SPMD style enabled by a unique rank for each process. While regular variables belong to the private space of each process, library data types such as `shared_var<T>` or `shared_array<T, B>` express scalar and array variables of elements of type `T` located in the shared space accessible by all the processes, respectively. While the scalar variables are physically located in the memory of process 0, shared arrays are distributed among the processes in a block cyclic fashion, the block size being `B`, which if not specified defaults to 1, thus giving place to a pure cyclic distribution. UPC++ also generalizes for the shared space the C++ pointers and references with the data types `global_ptr<T>` and `global_ref<T>`, respectively. The library takes advantage from the C++ operator overloading facilities, so that these objects can be built from its shared variables using the expected C++ operators such as `&` or `[]`, in the case of arrays. This way UPC++ provides a very natural way of manipulating the global space variables, as it is analogous to that of the private variables. The library also provides other facilities that are of little relevance to our explanation.

The interest of UPC++ is supported both by its degree of success among users compared to other alternatives to MPI [17], as well as by the development of a new version of the library [3] that emphasizes asynchronous computation. This newer version does not natively provide key elements for our proposal such as global references or shared distributed arrays. As a result, although possible, the integration of UPC++ DepSpawn with the new proposal seems less natural and straightforward. However we plan to tackle it future work.

UPC++ DepSpawn is an evolution of DepSpawn [16], a library for dataflow computing in shared memory, that targets the UPC++ environment. This way the new library supports dataflow execution in a distributed memory system managed by UPC++. The library integrates also the shared memory runtime of the original DepSpawn, which was compared to state of the art approaches in [14] providing good results, thus enabling in addition task parallelism within each UPC++ process.

The requirements to parallelize an application on top of UPC++ DepSpawn are simple. First, each task must be packaged in a C++ function that only communicates with other tasks by means of its arguments. The library will infer the nature of each argument from the type of its associated formal pa-

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

void dgemm(global_ref<Tile> dest, const Tile a, const Tile b) {
  // Implements 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 Cholesky factorization implemented with UPC++ DepSpawn

rameter. Namely, parameter data types that preclude the modification of the argument will indicate that the associated argument is only an input to the function. This is the case of constant references or non-reference types, which imply that the argument is passed by value. Non constant references, on the other hand, will hint the ability of the task to modify the associated argument, thus constituting a potential input and output of the task. As a result, contrary to other approaches, UPC++ DepSpawn requires no programming overhead for labeling the usage that each task will make of each argument. Once the tasks have been expressed in this fashion, the library is invoked by calling each task/function `f` on its arguments `args` using the syntax `upcxx_depspawn(f, args)` and invoking function `upcxx_wait_for_all()` at the point where the specification of the tasks has finished and we wish to wait for their completion.

Listing 1 illustrates the usage of the library for the parallelization of a complex algorithm, namely the Cholesky decomposition. The distributed array `A`, composed by `N`×`N` tiles, is provided by means of a UPC++ `shared_array` in which each element is a tile. Since this data type only provides unidimensional arrays, it is built as a vector of `N*N` tiles. The code then indexes it using macro `_` in order to map from the bidimensional indexing of the 2D matrix it represents to the linear indexing required by `shared_array` objects. The algorithm has exactly the same look as a sequential implementation, with the difference that each function is invoked by means of the `upcxx_depspawn` routine described above and that a `upcxx_wait_for_all()` finishes the algorithm. Notice that since UPC++ programs run in SPMD mode, this strategy implies that all the

processes go through the whole algorithm, thus having a complete picture of all the tasks to execute and their dependencies.

The processes independently decide which one runs each task by computing for each process the priority it has for being assigned the task. This priority is based on the location of the task arguments. Let us consider a task T with n arguments so that $l_j, 1 \leq j \leq n$, is the process in whose memory argument j resides and $w_j, 1 \leq j \leq n$, is an integer with the value 1 if the argument j is written by T and 0 otherwise. Then, the priority p_i associated to process i for the task T is given by $p_i = \sum_{j=1}^{n}(l_j == i) * (1 + w_j * W)$, where $a == b$ is an operation that returns 1 if $a = b$ and 0 otherwise, and W is an extra weight associated to written arguments. This way, the process with the highest priority p_i is assigned the task. Since the number of arguments of a task is limited and typically (quite) smaller or similar to the number of processes used in an execution, our algorithm does not compute p_i for every process. Rather, it iterates on the task arguments, accumulating the value $(1 + w_j * W)$ associated to argument j to p_{l_j}, the priority of the process whose memory has that argument. The p_i values are stored in a C++ `std::unordered_map` container that maps from process id i to priority p_i. The container only has entries for the processes that are related to at least one argument in the current task, and the maximum p_i is found iterating on the existing entries. If several processes happen to be associated to the maximum priority computed, the first one found is assigned the task. There are two important things to notice on this. First, since this container is unordered, the process chosen is not necessarily the one with the lower rank. Second, the process is chosen in a deterministic fashion based on the internal implementation of the container, which enables all the processes to obtain the same value independently, without any communication.

Listing 1 also includes the definition of the function `dgemm(a, b, c)`, which computes `a = a + b * c`. Since the function modifies its first argument but not the other two ones, the first parameter must be a non-constant reference, while other two ones can be just tiles, implying pass by value, or constant references to tiles. Since the arguments are tiles located in the shared space, the references should be provided by the UPC++ `global_ref<T>` data type so that they can be associated. As for arguments passed by value, there is no need to use parameters based on UPC++ data types, as UPC++ will implicitly obtain copies from the original arguments and place them in the associated parameters that have the corresponding regular C++ data type.

3 Partial TDG construction

Expressing the whole algorithm in each process that participates in the dataflow execution has the advantage that, as seen in Listing 1, the code of the parallel algorithm is totally analogous to that of the sequential version. This minimizes the development and maintenance cost. However, this also implies that every process has to store the information and analyze the potential dependencies of

every task, even when it may be possible to know in advance that the process will be in charge of portions of the TDG totally unrelated to that task. In practice, a process only needs to store and keep track of (a) the tasks it has to run, (b) those that provide the input dependences of those tasks, and, finally, (c) those that depend on the tasks assigned to the process. Let us notice that the larger the number of processes that participate in a UPC++ DepSpawn parallel computation, the smaller the portion of the TDG that is actually of interest to each process. As a result, the relative overhead associated to the processing of the whole TDG in each process grows with the number of processes used, thus limiting the scalability.

The library has been extended with a new function in order to deal with this problem. Namely, function `upcxx_cond_spawn(condition, f, args)` enables the user to specify the condition that a task has to fulfill in order to be of interest to the executing process. If the condition holds, the function acts as a regular `upcxx_spawn(f, args)` invocation since the process must know about the task. Otherwise the library neither builds nor processes the task, although it still performs two actions, both of them with negligible cost. We now explain them in turn.

The first action required is to record the existence of the task in the UPC++ DepSpawn task counter. This counter is local to each process and it is increased each time a `upcxx_spawn` or `upcxx_cond_spawn` function is invoked. The task associated to each one of these invocations receives the value of the counter at that moment as unique identifier. This identifier is internally used mainly for notifying to other processes when a given task has finished. Notice that this implies that each task must receive the same identifier in all the processes that have it in their TDG. Our implementation achieves this by ensuring that every task has the same identifier in all the processes. This is accomplished by making each process iterate over the whole algorithm to implement, making the `upcxx_spawn` or `upcxx_cond_spawn` invocation associated to every task in the global TDG. The standard `upcxx_spawn` invocations and the `upcxx_cond_spawn` ones in which the condition is true will in addition actually create the task. On the contrary, the `upcxx_cond_spawn` invocations whose condition does not hold will not create any task. However, they will increase the local task counter to make sure that future tasks that are of interest to the calling process receive the same identifier as in the other processes that need to know about such tasks.

The second action taken when the condition of a `upcxx_cond_spawn` does not hold is to record the task just skipped as completed, as otherwise it would be considered as live. The reason for this behavior is that, as explained in [15], the runtime periodically stops the generation of new tasks when it detects that the number of live tasks goes above a reasonable threshold. UPC++ DepSpawn follows this strategy in order to avoid generating a TDG too large as well as to speed up the processing of the pending live tasks assigned to the process. When this happens, the main thread tries to execute pending tasks that are are ready for execution and to answer remote requests from other processes. Our runtime computes the number of live tasks that can trigger this behavior

as the difference between the number of tasks created, which is given by the current value of the task counter discussed in the previous paragraph, and the number of tasks completed. This latter value is also another local counter in each process. Marking the tasks ignored by `upcxx_cond_spawn` with a false condition as completed helps the runtime to make its periodic stops only when it is actually worthwhile. As we can see, just as in the case of the first action, this recording does not need any synchronization with other processes and it is very cheap, just requiring the increment of a counter.

The new `upcxx_cond_spawn` function can be exploited in many algorithms. For example, in stencil computations we know that only the tasks that operate on the data assigned to our process or the immediately neighboring ones can be of interest. Similarly, in tile-based computations such as the Cholesky decomposition in Listing 1 it is possible to infer which are exactly the tasks that are related due to dependencies based on the location in the matrix of the tiles used by each task. As a practical example, one can infer from the Cholesky code that the `potrf` computation on tile `(i,i)` is only of interest to the `trsm` computations that operate on all the tiles below it in column `i`. Therefore, the `potrf` task only needs to be known to the processes that own tiles in this column. If we assume a 2D block-cyclic distribution of the tiles on a 2D mesh of processors, which is the usual one for this algorithm, this means that the processes in the same column of the 2D mesh as the one who owns tile `(i,i)` are the interested ones. Thus, if `proc_col` stands for the number of the column of the current process in the 2D mesh, and `col_cyc` is the period for the distribution of the columns of the matrix on this mesh, the computation only belongs to the TDG of the processes in which `(i % col_cyc)` yields `proc_col`. As a result, this task would be spawn as

```
upcxx_cond_spawn((i % col_cyc) == proc_col, potrf, A[_(i,i)]);
```

An interesting question is whether compiler technology could be applied to generate these expressions, saving users from having to identify them. While we do not know of any implementation that achieves this from actual source code, DAGuE/PaRSEC [6], discussed in Section 5, can accept as input a sequential pseudo-code, consisting of simple loop nests, from which it can infer the exact dependencies among tasks in such pseudo-code relying on the integer programming framework Omega-Test [20]. We must note however that this strategy can only be applied in codes in which all the dependencies can be statically stablished at compile time, which is the scope to which PaRSEC is restricted. On the contrary, as seen in [15], UPC++ DepSpawn supports applications whose dependencies can only be known at runtime, and even applications in which the dependencies can dynamically change during the execution. Given the dynamic nature of these problems, the automated identification of the conditions under which each task is of interest to a given process does not seem feasible for a compiler.

4 Evaluation

Our evaluation has been performed in a cluster with 32 nodes interconnected by Infiniband FDR@56Gbps. Each node consisted of 2 Intel Xeon E5-2680 v3 processors at 2.5GHz with a 128GB DDR4 memory. Since each processor has 12 cores this results in a total of 768 cores. Compilations were performed with g++ 6.4 and optimization level O3.

Our evaluation relies on four benchmarks: the right-looking Cholesky factorization in Listing 1, the LU decomposition, the sparse LU decomposition, and the Gauss-Seidel stencil computation. The four algorithms fulfill the necessary condition for this evaluation that it is possible to determine the subset of tasks that are of interest to each process. The sparse LU decomposition is particularly interesting and challenging, given the irregular nature of its computations, which not only depends on the sparsity pattern of the matrix, but even evolves during the execution of the algorithm, as the decomposition generates fill in.

Figures 1 to 4 show the performance of the Cholesky factorization, LU decomposition , sparse LU decomposition and Gauss-Seidel stencil for different numbers of nodes, respectively. The experiments were performed for all the benchmarks using two problem sizes and one process per node with one thread per core. In the case of the first wo codes we also plot the performance achieved by DPLASMA [5], which is the leading implementation for these algorithms in this environment. This library also follows a task-based dataflow-driven approach provided by the PaRSEC runtime [6], discussed in Section 5. Notice that DPLASMA cannot provide a baseline for our sparse LU decomposition because PaRSEC cannot support irregular codes. Both DPLASMA and our implementations rely on the highly optimized OpenBLAS library version 0.3.1 for the BLAS operations, but while DPLASMA relies on OpenMPI 2.1.1 for the computations, UPC++ DepSpawn runs on top of GASNet 1.28.0. The performance of these benchmarks depends both on the tile size as well as on the mapping of the tiles to the processes used. Thus, for all our experiments we always plot the performance achieved for the best tile size and mapping.

Since the impact of the optimization is difficult to see for small numbers of nodes, Figure 5 shows the speedup of UPC++ DepSpawn with partial TDG generation with respect to the original UPC++ DepSpawn implementation for each benchmark, problem size and number of nodes. When a single node is used, the process must build the whole TDG necessarily, thus this figure starts at two nodes. Gauss-Seidel is the benchmark that benefits the most from this improvement, while the impact on the other codes is around the same order of magnitude. The reason is that Gauss-Seidel has a complexity $O(6N^2)$, which is much smaller than the $O(N^3/3)$ and $O(2N^3/3)$ of the Cholesky and LU decompositions, respectively. As a result, the relative cost of the construction of the whole TDG is noticeable bigger for this algorithm than for the other ones.

The figures also show that while the impact of this optimization is discrete for small number of nodes, it is more noticeable from 8 nodes onwards. This

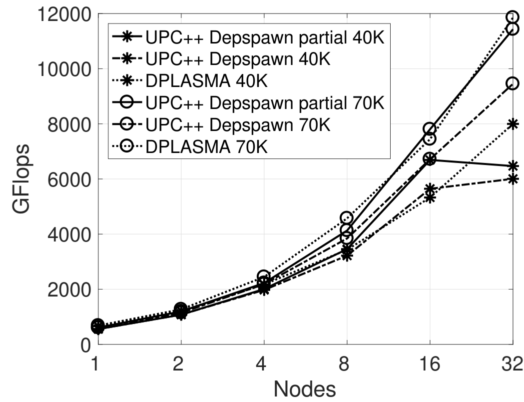

Fig. 1 Performance of the Cholesky descomposition benchmark

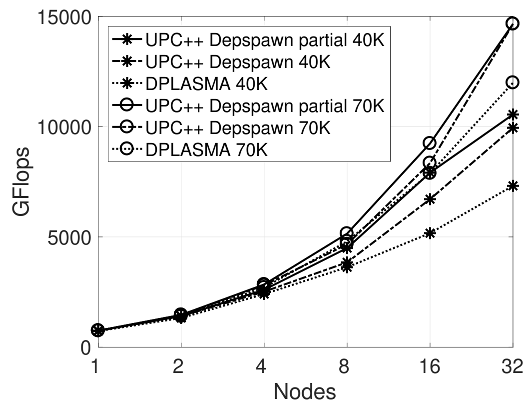

Fig. 2 Performance of the LU descomposition benchmark

makes sense, since the more processes are used in the parallel execution, the smaller the portion on the TDG actually needed in each node, and thus more time can be saved by this optimization. We can see however that in the linear algebra benchmarks the relative impact stalled and even diminished in most tests at 32 nodes. This is related to the fact that it is increasingly likely there are idle threads during the execution as the number of processes used to solve a given problem grows for two reasons. First, the smaller amount of work per node reduces the load of each process. Second, the more distributed memories we use in the execution, the more unlikely it is that the data involved in a dependency is found in the local memory of the executing process, thus resulting in more communications and related stalls. Since the UPC++ DepSpawn runtime can take advantage of these idle cycles to overlap the construction and analysis of the TDG with the execution of the algorithm, the availability of idle threads reduces the impact of the optimization. This factor is less important in Gauss-Seidel than in the other benchmarks because its optimal mapping of tiles to processes assigns a consecutive block of tiles to each pro-

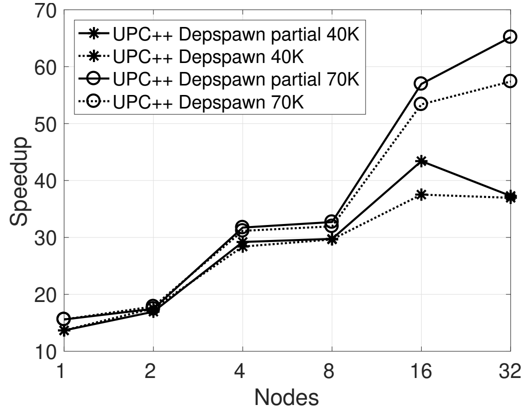

Fig. 3 Performance of the sparse LU descomposition benchmark

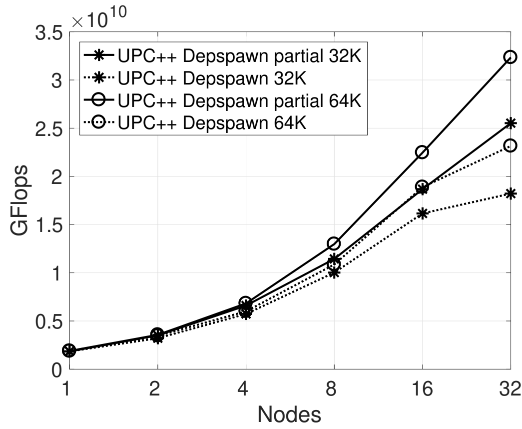

Fig. 4 Performance of the Gauss-Seidel benchmark

cess, thus reducing to the minimum the communications, which only happen for the rows and columns along the borders of the block of tiles assigned to each process. The other algorithms in contrast require communications for the vast majority of the tasks as the number of processes grows, and in addition those communications do not consist in a single row or column of a tile, but in a whole tile each.

We can see that the partial generation of the TDG plays a key role in improving the qualitative comparison of UPC++ DepSpawn with a state-of-the-art approach such as DPLASMA for Cholesky. This way, once this optimization is applied the performance of the two tools is quite similar for the large matrix test. As for the smaller matrix, there are both situations in which the two proposals offer very similar performance as well as others where one of them clearly outperforms the other one. Regarding LU, where the UPC++ DepSpawn baseline already offered the best results, the application of the op-

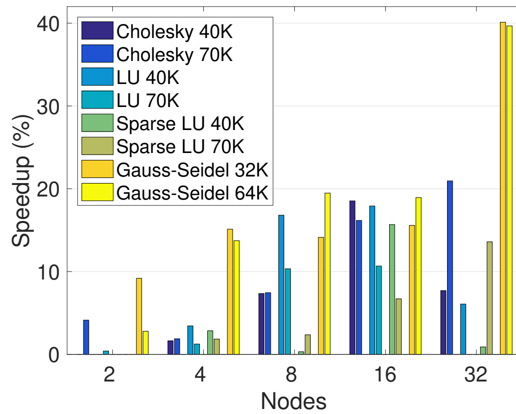

Fig. 5 Speedup achieved by the TDG partial generation

Table 1 Source lines of code

Implementation	Cholesky	LU	sparse LU	Gauss-Seidel
UPC++ DepSpawn	13	13	31	16
UPC++ DepSpawn optimized	20	22	37	18
PTG DPLASMA	44	39	–	–

timization has smaller effects that slightly further improve the performance of the runtime or at worst have a negligible effect.

It is also interesting to evaluate the impact on this optimization on programmability, as the specification of the conditions that a task must fulfill to be part of the minimum TDG required in each process involves additional coding. Table 1 compares the number of lines, excluding comments and empty lines, of the baseline and the optimized UPC++ DepSpawn implementations of the algorithms used. The table also includes for comparison purposes the minimum number of lines required for the Parameterized Task Graph (PTG) specification of the Cholesky and LU benchmarks provided by DPLASMA (see an example in [13]). This means that for the measurement we have cleaned everything that is not strictly required in the PTG.

Most of the growth of the number of lines of the UPC++ DepSpawn implementation is not associated to the conditions themselves, as they can be written in a single line per task type, but to the computation of parts of the conditions that can be reused in several conditions in separate variables in order to improve the readability of the code. This way, depending on the size of the initial code, the coding style and the desire to maximize the readability of the code, one can expect increases between 15% and 70% of the number lines. However, even with this additional coding, this value will still be at about one half of the number of lines required by the equivalent PTG strategy.

5 Related work

There are several strategies for developing data-flow applications in distributed memory environments. One approach involves requiring the user to explicitly specify beforehand all the tasks and the dependencies among them. This is the case of the Parameterized Task Graph (PTG) model [11], on which the PaRSEC framework [13] used in this paper is based. On the one hand, this static availability of the dependences allows the framework to generate offline the portion of the TDG that is of interest for each process, achieving results similar to the ones of the optimization presented in this paper. On the other hand, the requirement to know in advance all the dependencies in the code preclude the application of this approach to problems whose dependencies can only be known at runtime.

The most common approach in the literature consists in explicitly enforcing the dependencies by means of programming mechanisms and objects manipulated by the user. Indeed, a large number of synchronization mechanisms such as locks [8], clocks [10], full/empty bits [9], synchronized blocks [25], synchronization variables [12] and futures [3], to name a few, have been proposed.

Another alternative is to express the dependences implicitly by specifying the inputs and outputs of each task and providing an apparently sequential version of the algorithm based on those tasks. During the execution of this version a runtime builds the TDG and dynamically schedules the tasks according to the dependencies inferred. This requirement to observe in sequence all the tasks in the application is a potential bottleneck that can severely limit the scalability, which is the problem tackled in this paper. In this family of solutions we find centralized approaches in which a single process is responsible for building and managing the TDG, while the other processes are servers that run the tasks assigned [22, 7]. These approaches can limit the TDG overheads by generating smaller numbers of tasks of a coarser grain, which in [7] can be further parallelized within a node in a second level of parallelism. Other approaches generate the TDG in parallel in all the processes involved, this being the case of our library and StarPU [2, 1], which lacks the multithreaded nature of our solution within each process and operates on top of MPI. The strategy of generating a partial TDG in each process makes sense for these approaches, and is in fact mentioned as future work in [1]. There are projects such as Legion [4], which has moved from a library-based implementation to a new language and related compiler [21]. This improvement enabled static analyses that are impossible at runtime and moved offline part of the cost, in addition to achieving a clearer notation. Other projects based on implicit parallelism were based on languages and compilers that generate the low level data-flow application from the high level user specification right from their inception [23].

6 Conclusions

UPC++ DepSpawn is a library that allows to exploit task-based dataflow computing in hybrid shared/distributed memory systems by applying both process-level and thread-level parallelism. Two of the most distinctive properties of the library are its integration on a PGAS environment such as UPC++, and the reliance on the data types of the formal parameters of the functions that constitute the tasks to execute to infer the data dependencies. This gives place to a quite terse an natural notation.

In this paper we have improved the initial implementation of the library by adding the ability to restrict the generation of the TDG in each process to only the portion that is actually of interest to that process. The reduction of runtime overhead due to this optimization allowed to improve the execution time of the Cholesky factorization, the LU decomposition, the sparse LU decomposition and the Gauss-Seidel stencil by 8.6%, 6.4%, 3.9% and 18.9% on average, respectively, in executions ranging from 2 to 32 nodes with 24 cores each.

Our future work includes the development of a version of UPC++ DepSpawn on top of the new version of UPC++ presented in [3] and exploring the usage of heterogeneous computing in the tasks managed by the library.

Acknowledgements

This research was supported by the Ministry of Science and Innovation of Spain (TIN2016-75845-P and PID2019-104184RB-I00, AEI/FEDER/EU, 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 2017/04). 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. We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of their computers.

References

1. 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) (2014)
2. 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)
3. 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 (2019)

4. 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, SC '12, pp. 66:1–66:11 (2012)
5. 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 (2011). DOI 10.1109/IPDPS.2011.299
6. 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). DOI 10.1016/j.parco.2011.10.003
7. 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, ICS '13, pp. 359–368 (2013)
8. Burke, M.G., Knobe, K., Newton, R., Sarkar, V.: UPC language specifications, v1.2. Tech. Rep. LBNL-59208, Lawrence Berkeley National Lab (2005)
9. Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the Chapel language. Int. J. High Perform. Comput. Appl. **21**(3), 291–312 (2007). DOI 10.1177/1094342007078442
10. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: 20th Annual ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '05, pp. 519–538 (2005)
11. 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 (1995). DOI 10.1109/HICSS.1995.375471
12. Cray Inc: Chapel language specification version 0.984 (2017)
13. Danalis, A., Jagode, H., Bosilca, G., Dongarra, J.: PaRSEC in practice: Optimizing a legacy chemistry application through distributed task-based execution. In: 2015 IEEE Intl. Conf. on Cluster Computing, pp. 304–313 (2015). DOI 10.1109/CLUSTER.2015.50
14. Fraguela, B.B.: A comparison of task parallel frameworks based on implicit dependencies in multi-core environments. In: 50th Hawaii Intl. Conf. on System Sciences, HICSS'50, pp. 6202–6211 (2017)
15. Fraguela, B.B., Andrade, D.: Easy dataflow programming in clusters with UPC++ DepSpawn. IEEE Transactions on Parallel and Distributed Systems **30**(6), 1267–1282 (2019)
16. 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)
17. Koniges, A., Cook, B., Deslippe, J., Kurth, T., Shan, H.: MPI usage at NERSC: Present and future. In: 23rd European MPI Users' Group Meeting, EuroMPI 2016, pp. 217–217 (2016). DOI 10.1145/2966884.2966894. URL https://www.nersc.gov/assets/Uploads/MPI-USAGE-at-NERSC-poster.pdf
18. Nieplocha, J., Palmer, B., Tipparaju, V., Krishnan, M., Trease, H., Aprà, E.: Advances, applications and performance of the global arrays shared memory programming toolkit. Intl. J. of High Performance Computing Applications **20**(2), 203–231 (2006). DOI 10.1177/1094342006064503
19. Numrich, R.W., Reid, J.: Co-array Fortran for Parallel Programming. SIGPLAN Fortran Forum **17**(2), 1–31 (1998). DOI 10.1145/289918.289920
20. Pugh, W.: The Omega test: A fast and practical integer programming algorithm for dependence analysis. In: 1991 ACM/IEEE Conf. on Supercomputing, Supercomputing '91, p. 4–13. Association for Computing Machinery, New York, NY, USA (1991). DOI 10.1145/125826.125848
21. 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, SC '15, pp. 81:1–81:12 (2015). DOI 10.1145/2807591.2807629
22. 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)

23. 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 (2013). DOI 10.1109/CCGrid.2013.99
24. Yelick, K., Bonachea, D., Chen, W.Y., Colella, P., Datta, K., Duell, J., Graham, S.L., Hargrove, P., Hilfinger, P., Husbands, P., Iancu, C., Kamil, A., Nishtala, R., Su, J., Welcome, M., Wen, T.: Productivity and performance using partitioned global address space languages. In: Proc. 2007 Intl. Workshop on Parallel Symbolic Computation, PASCO '07, pp. 24–32 (2007). DOI 10.1145/1278177.1278183
25. Yelick, K.A., Graham, S.L., Hilfinger, P.N., Bonachea, D., Su, J., Kamil, A., Datta, K., Colella, P., Wen, T.: Titanium. In: Encyclopedia of Parallel Computing, pp. 2049–2055. Springer US (2011)
26. 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 (2014)