A Comparison of Task Parallel Frameworks based on Implicit Dependencies in Multi-core Environments

Basilio B. Fraguela
Universidade da Coruña, A Coruña, Spain
Email: basilio.fraguela@udc.es

***Abstract*—The larger flexibility that task parallelism offers with respect to data parallelism comes at the cost of a higher complexity due to the variety of tasks and the arbitrary patterns of dependences that they can exhibit. These dependencies should be expressed not only correctly, but optimally, i.e. avoiding over-constraints, in order to obtain the maximum performance from the underlying hardware. There have been many proposals to facilitate this non-trivial task, particularly within the scope of nowadays ubiquitous multi-core architectures. A very interesting family of solutions because of their large scope of application, ease of use and potential performance are those in which the user declares the dependences of each task, and lets the parallel programming framework figure out which are the concrete dependences that appear at runtime and schedule accordingly the parallel tasks. Nevertheless, as far as we know, there are no comparative studies of them that help users identify their relative advantages. In this paper we describe and evaluate four tools of this class discussing the strengths and weaknesses we have found in their use.**

***Keywords*-programmability; task parallelism; dependencies; programming models**

I. Introduction

Many applications require the exploitation of task parallelism to benefit from all the parallelism they can expose, and sometimes, just to be parallelized at all. With the need to express different parallel tasks comes the requirement to properly schedule and synchronize them according to the arbitrary patterns of dependences that they can exhibit. There is also of course the option to resort to speculation under Transactional Memory [20] or Thread Level Speculation [28]. However, speculation typically has non-negligible costs and a wide range of applications can be in fact successfully parallelized by properly ordering the execution of their tasks so that their dependencies are fulfilled. While this has been done using low level approaches since the early days of parallel computing, the growing need to parallelize every kind of application together with the large availability of parallel systems, particularly since the appearance of multi-core processors, led to the proposal of high-level approaches that facilitate this task. Some of the most advanced programming tools in this category are those in which the users just declare, as implicitly as possible, the dependencies of their tasks, but without specifying how they must be met, letting instead the underlying compiler and/or runtime automatically manage them. The fact that these tools require minimum effort from the developers, while they allow to build extremely complex task graphs and provide maximum parallelism thanks to the graph scheduling algorithms they implement, makes them particularly interesting.

Despite the relevance of this approach, we have not found comparative studies of the existing practical alternatives of this kind beyond some performance comparisons [36]. For this reason this paper tackles this issue describing and comparing some of the high-level approaches available nowadays with a particular focus on their semantics and ease of use, our main aim being to help identify the best tool for each problem at hand. Also, while there are proposals that extend these ideas to clusters and hybrid systems, this study is restricted to the widely available multi-core systems that are ubiquitous nowadays, so that the problems related to heterogeneity and distributed-memory are not considered. This way, we set some basic criteria to choose the programming environments for our comparison:

- Allow users to implicitly declare the task dependencies, automatically taking care of the concrete implied synchronizations and scheduling.
- Provide a high-level API (i.e. oriented to final programmers), and be both publicly available and well documented.
- Be usable in some programming language among the most widely used in the literature/research on parallel applications, mainly in the fields of scientific and engineering computing. Besides, for fairness the comparison should use the same language for all the tools. This implicitly discards proposals based on new languages, which besides being more complex to compare fairly, tend to have lower rates of adoption than compiler directives and libraries, as these latter alternatives better facilitate the reuse of existing codes.
- Do not require the usage of concepts or APIs related to distributed computing, which would distort the purely shared-memory based comparison.

Based on these criteria, four tools were chosen. A popular strategy for the declaration of task dependences is the annotation of the inputs and the outputs of each task together with a valid order of execution, implicitly given by the sequential execution of the tasks. Since the inclusion of

URI: http://hdl.handle.net/10125/41914
ISBN: 978-0-9981331-0-2

dependencies for tasks in OpenMP 4.0 [26] this standard is the most widespread option that supports this paradigm, and thus the first we study. A very active and related project, which in fact pushed for the adoption of dependent tasks in OpenMP [11], is OmpSs, which integrates features from the StarSS family of programming models [27]. While OpenMP and OmpSs cover well the space of compiler directives in our scope of interest, the area of libraries is much more sparse. Because our focus here is on the semantics and the programming style, we will discuss the two libraries that fit our criteria and that we find to be more original in their approach. Both of them are based on C++, which is not surprising given the excellent properties of this language to enable high performance coupled with ease of use. In fact, the first library, DepSpawn [16], heavily relies on the properties of this language to minimize the effort of the programmer. The second library is Intel® CnC (Concurrent Collections) [6], [8], whose main interest lies in its very original approach for the specification of the dependences.

The rest of this paper is organized as follows. Section II describes the main characteristics of the frameworks analyzed. They are then compared in terms of performance and programmability in Section III. This is followed by a discussion on related work in Section IV, and the last Section is devoted to our conclusions.

II. Frameworks

The frameworks analyzed are now described in turn. In all the cases we only center on the creation of dependent tasks, skipping other possible functionalities, even if they are needed to exploit this paradigm. A clear example are explicit synchronizations, which are required even if only to make sure that all the parallel tasks have completed before leaving the program. Also, since the libraries tested are based on C++, making it the natural language for our comparison, the specific problems found in the use of the compiler directives in this language will be described.

A. OpenMP

The well-known OpenMP standard extended in [26] the `task` construct introduced in version 3.0 [2] with support for task dependences by means of the `depend` clause. The clause allows to define lists of data items that are only inputs, only outputs, or both input and outputs using the notation `depend(`*`dependence-type`*` : `*`list`*`)` where *`dependence-type`* is `in`, `out` or `inout` and the data items can be variables or array sections. The annotated task will be scheduled for execution only when the dependences expressed by those data items are satisfied with respect to preceding tasks in the same task region, i.e., with the same parent.

Two of the limitations of this approach have been discussed in [36]. For example, this mechanism alone does not allow to use dependent tasks for reduction processes, forcing to use other OpenMP functionalities such as atomic operations or critical sections, or resort to other high level constructs such as `section` or `for`. The other issue is related to the specification of multi-dimensional array regions when instead of a static or variable length array we have a pointer, although this can be solved casting the pointer.

We find much more relevant other limitations. For example, the standard requires the dependent items to have either identical or disjoint storage. This means that if array sections appear in the lists of dependences, they must be either identical or disjunct. Similarly, the standard explicitly states that a variable that is part of another variable (such as a field of a structure) but is not an array element or an array section cannot appear in a depend clause, which again points to the unavailability of tests on partial overlaps of objects. In fact OpenMP does not allow data members in its clauses, which restricts the usefulness of this framework in programs with composite data types other than arrays.

A problem of OpenMP that is specific to C++ is its treatment of references. While references are implemented under the hood by means of pointers, this is never exposed to the programmers, who just see and use them as alias to existing data items. OpenMP however considers references as what they are actually for the compiler (pointers) and in fact prohibits reference types for `private`, `firstprivate` and `threadprivate` variables, because the compiler would privatize the reference, not the object accessed through it, giving place to a wrong behavior. For the same reasons, references, which are heavily used in C++ applications both for performance and programmability reasons, are not suited to express dependences for OpenMP tasks.

B. OmpSs

Dependent tasks are the core of the OmpSs programming model [27], [3]. Task creation follows the OpenMP syntax with a slight difference in the specification of dependencies. Namely, they are provided using three different clauses `in(`*`list`*`)`, `out(`*`list`*`)` and `inout(`*`list`*`)`, which express input, output and both input and output dependences on the memory positions provided in the list, respectively. The dependences are enforced with respect to preceding tasks built in the same task or outside any task code, as OmpSs programs assume that their execution begins with a master thread that can spawn tasks to other threads at any point. A more interesting difference is that OmpSs allows to annotate a function declaration or definition with a `task` pragma that specifies its dependencies on its formal parameters. This automatically turns every invocation of the function into a dependent task, the dependency on each actual argument being the one specified for its associated formal parameter.

Contrary to OpenMP, OmpSs tasks support the `reduction` clause and their lists of dependencies allows to specify data members as well as expressions that

provide data items (e.g. `*ptr` or `b.mx[i][j]`). Another advantage is that it allows to express dependencies on C++ references with the intuitive and natural semantics that the dependence is on the object aliased by the reference. A restriction in common with OpenMP is though that both tools only correctly support dependences on array sections, and in general data items, that completely overlap, as although overlapped memory region detection has been implemented for clusters in [5], the currently available compiler (15.06) and public specification [3] do not provide it. This way, the support of data members in its clauses is dangerous, as tasks that operate on a member of an object can run in parallel with other tasks that modify the whole object, including that member.

Although this is not a fundamental but a technological issue, it is worth mentioning that relevant limitations of OmpSs with respect to the other approaches are that the current version only works in Unix and that it does not support the increasingly adopted C++11 and C++14 standards. This is unfortunate as these new standards largely help in the development of every kind of applications, and in particular scientific and parallel applications, thanks to a large number of critical improvements such as the definition of a multithreading memory model, rvalue and move semantics (which avoid costly copying processes), lambda functions, etc.

C. DepSpawn

C++ is one of the languages more heavily used for parallel computing given the high performance it can provide, its flexibility, the large number of existing libraries and the rich support of different programming paradigms it provides. The DepSpawn library [16] takes advantage of these properties, particularly the template metaprogramming, to simplify to the maximum the creation of dependent tasks. This proposal requires the parallel tasks to be function calls and it infers their dependencies from the types of the formal parameters of the functions, so that no user intervention is required. Namely, arguments passed by value are inherently only inputs, as the function cannot modify the original argument provided. Similarly, arguments passed by reference are both inputs and outputs, because the function can both read and modify the argument provided by the caller. An exception are arguments passed by constant reference (`const&`), which indicate that although the function can access the argument provided by the caller, it will not modify it, which makes the argument read-only, and thus only input to the function. Users only need a keyword, `spawn`, to generate tasks. Namely, `spawn(f, a, b, ...)` generates a parallel task for the execution of `f(a, b, ...)` that respects the dependencies on the arguments provided according to the nature of their corresponding formal parameters in `f`.

DepSpawn supports dependencies on reference arguments following the same semantics as OmpSs, that is, establishing the dependence on the object aliased by the reference. Also, because it tracks the actual arguments provided by the caller, the task arguments can be arbitrarily complex C++ expressions. Contrary to the compiler directives described before, DepSpawn tasks respect dependencies with respect to any task spawned before them, no matter they come from the same parent or not. Because the order of execution with respect to subtasks from other parents is unknown, what this means in practice is that tasks satisfy the data dependencies with respect to preceding tasks of the same parent, as we would expect in any environment of this kind, and they are mutually exclusive with other parallel tasks when at least one of them writes in a common memory position. This allows to use dependent tasks for reduction processes and critical sections. Another important difference is that DepSpawn detects partial overlaps on the memory regions of the task arguments. In the case of generic subarrays that do not occupy a consecutive memory region, or where their size cannot be inferred from their data type (e.g. a pointer to a array), the library requires the arrays to be handled by an `Array` class it provides that is derived from the Blitz++ library [35]. With this mechanism, it supports dependent tasks on arbitrary (sub)arrays of up to 10 dimensions.

Finally, this library relies on the Intel TBB [29] for its internal threading systems and requires a compiler that supports C++11, being thus portable to the vast majority of parallel computers and main operating systems.

D. Intel® Concurrent Collections (CnC) for C++

The CnC approach to build general task based-applications following a declarative model for the dependencies is very different from the previous ones [6], [8]. Instead of annotating or learning the inputs and outputs of each task, and their correct ordering in a sequential execution, CnC builds a graph of tasks, called *steps*, that communicate with each other by means of data *items* and control *tags*. These three kinds of elements are organized in sets, called *collections*, in which each individual element is identified by means of a unique *tag*, which is a tuple of *tag components*; typically integers. In the case of control tags, they are their own identifier. The steps produce and/or consume data items by pushing and/or retrieving them from their collections, respectively, always identifying them by their associated tags. The role of tag collections is to *prescribe* the execution of steps. This means that they specify whether a given step instance, which is an individual concrete execution of a step/task identified by a control tag, can proceed or not, but not when this will be done. Each tag collection can prescribe one or several step collections. The prescription is made by pushing into the tag collection the tag associated to that/those step(s). All the operations on collections take place during the execution of the steps under the control of the programmer, and the runtime executes a step instance, at some point, after it has been prescribed.

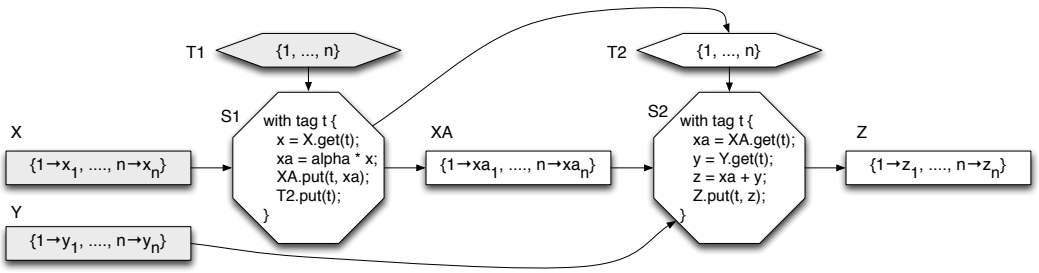

Figure 1. CnC SAXPY example

Figure 1 shows schematically a possible SAXPY ($y_i = y_i + \alpha x_i$) CnC implementation where the shadowed collections are initialized by the environment while the other ones are filled by the algorithm, and the result is stored in a different collection Z to simplify the example. All the tags are integers and $a \rightarrow b$ means that tag a is associated to item b. Each tag t in T1 triggers the execution of a step S1 with that tag, which computes αx_t and inserts it in collection XA. By inserting the tag t in T2, it enables the execution of a step S2 with that tag. Such step adds αx_t to y_t, putting the result in the output collection Z.

While the use of the elements just described suffices to build any CnC program, the library offers a series of *tuners* (https://icnc.github.io/api/index.html) that allow programmers to specify additional information to the runtime to optimize the execution. For example, they can specify which are the exact data items required by a step instance, so that it is only launched when they are all available. Another example is to specify how many times will the program use a given data item so that it can be deallocated after the last usage, as otherwise the runtime cannot know when it is no longer needed.

A very interesting property of CnC is that, unlike the previous approaches, it supports distributed-memory environments[1], although this requires additional coding. The threading is based on Intel TBB [29], so similar comments about portability as in DepSpawn apply, except that C++11 support is not required.

III. Evaluation

As in this study we are mostly interested in the extent to which these approaches simplify the development of applications based on dependent tasks, a programmability comparison will be first made, followed by a performance evaluation. By far, the most common codes found in evaluations of tools of this kind are matrix operations performed by blocks [8], [16], [36]. This is not surprising, as this class of algorithms can present very complex patterns of task dependencies that are both the largest challenge and the largest opportunity for performance improvement for these tools, particularly when compared to approaches that cannot support arbitrary patterns of dependencies. As a result, our analysis is also based on algorithms of this kind that can be parallelized with any of the four tools.

Another preoccupation for the evaluation is of course fairness. While the author is well versed in the use of the first three tools presented, which are besides quite straightforward to use, his degree of expertise with CnC, particularly with its tuners, was smaller. For this reason, in order to ensure the quality and fairness of the comparison, two optimized (i.e. with tuners) codes distributed with the Intel CnC were used. The original codes included elements related to distributed computing, which were carefully cleaned so that the baselines only include code to run in shared-memory environments. The two codes are highly optimized. In fact one of them was used in [33] for a deep analysis of the performance of CnC using the same structure and tuners. The other three versions were developed from these cleaned codes in an analogous style, so that the differences among them can only be attributed to the tool used for the parallelization. This also implies that since DepSpawn and CnC are based on C++, all the versions are written in this language, so that language differences do not affect the comparison either. The two codes are a right-looking Cholesky decomposition of a lower triangular matrix and an in-place matrix inversion based on Gauss-Jordan elimination. In both algorithms, written in terms of tiles in Fig. 2, the basic linear functions (syrk, gemm) are provided by calls to a external BLAS library.

A. Programmability

Measuring and analyzing programmability is a very complex task with many aspects involved. The ideal strategy is probably to study it based on the development times, quality of the code written, opinions, etc. of teams of programmers [34], preferably with a degree of experience in the approaches tested that is as similar as possible among the developers involved, but unfortunately such teams are seldom available. For this reason, an approach followed by many authors is to rely on objective metrics extracted from the source code, as they are reproducible and avoid subjective impressions. We follow this strategy measuring

[1]As seen in [5], OmpSs supports clusters, but that version is not publicly available at the moment of writing this paper.

Table I
COMPLEXITY METRICS FOR THE BASELINES AND ABSOLUTE INCREASES FOR THE DIFFERENT PARALLELIZATION STRATEGIES. SL STANDS FOR SLOCS AND PE FOR PROGRAMMING EFFORT (IN THOUSANDS).

Algorithm	Baseline		OpenMP		OmpSs-inv		OmpSs-decl		DepSpawn		CnC		CnC-untuned	
	SL	PE	ΔSL	ΔPE	ΔSL	ΔPE	ΔSL	ΔPE	ΔSL	ΔPE	ΔSL	ΔPE	ΔSL	ΔPE
Cholesky	148	1022	9	170	6	166	6	69	3	53	125	1136	107	996
Inversion	255	1673	11	153	8	124	8	83	8	78	147	2324	108	1381

```
for(i = 0; i < dim; i++) {
  A[i][i] = potrf(A[i][i]);
  for(r = i+1; r < dim; r++) {
    A[r][i] = trsm(A[i][i], A[r][i]);
  }
  for(j = i+1; j < dim; j++) {
    A[j][j] = dsyrk(A[j][i], A[j][j]);
    for(r = j+1; r < dim; r++) {
      A[r][j] = A[r][j] + A[r][i] * A[j][i];
    }
  }
}
```

(a) Cholesky factorization

```
for(i = 0; i < dim; i++) {
  A[i][i] = inverse(A[i][i]);
  for (j = 0; j < dim; j++) {
    if (j != i)
      A[i][j] = A[i][i] * A[i][j];
  }
  for(r = 0; r < dim; r++) {
    if(r != i) {
      for (j = 0; j < dim; j++) {
        if (j != i)
          A[r][j] = A[r][j] - A[r][i] * A[i][j];
      }
      A[r][i] = - A[r][i] * A[i][i];
    }
  }
}
```

(b) Matrix inversion

Figure 2. Algorithms tested, where `dim` is the number of tiles per dimension

two metrics in the codes. The first one are the source lines of code excluding comments and empty lines (SLOCs). The second one is the Halstead programming effort [18], which estimates the development cost of a code by means of a reasoned formula based on the number of unique operands, unique operators, total operands and total operators found in the code. For this, the formula regards as operands the constants and identifiers, while the symbols or combinations of symbols that affect the value or ordering of operands constitute the operators. In our opinion the programming effort approximates better the complexity faced by programmers than the SLOCs, as it is well known that lines of code can widely vary in complexity.

Table I shows the value of the complexity metrics for the baseline sequential versions, which include a generic class to represent tiles, and their absolute increases when they are parallelized with each one of the tools discussed. As explained in Sect. II-B, in the case of OmpSs, tasks associated to function invocations can be annotated either in the function invocation or in the function declaration, which automatically turns all the invocations to the function into parallel tasks. Both approaches, which we will call in the rest of the paper OmpSs-inv and OmpSs-decl, respectively, have been measured for completeness in Table I. While both alternatives require the same number of SLOCs in our codes, OmpSs-decl reduces the programming effort because its dependency clauses are based on the formal parameters rather than on the actual arguments, which usually have a more complex form. The problem with this approach is that the fact that the function becomes a parallel task wherever it appears in the application can be dangerous and undesired. In the case of CnC, since the programming cost of the tuners is non-negligible, we also developed an untuned version by cleaning everything related to tuners from the codes. CnC requires users to define many classes and objects that do not exist in other programs, such as the collections or the tags. It also enforces new protocols such as the access to data by means of retrievals from collections or the execution of new steps by adding new tags to control collections. These activities and concepts require much more coding than the other alternatives, on which we focus now.

The usage of OpenMP, OmpSs and DepSpawn is illustrated in Fig. 3 by means of analogous excerpts of our implementation of the matrix inversion code. In the case of OmpSs, the OmpSs-inv version is shown, so that the differences appear in the same places and the codes are more directly comparable. In these codes the matrix is stored in the same way as in the CnC codes, namely in a unidimensional dynamically allocated array called `tiles`, where each element is an object of the tile class mentioned before. The matrix tiles are stored by rows in the array, so that if the matrix has `dim`×`dim` tiles, the tile (`i`,`j`) is located in the position `dim`×`i`+`j`. The examples show the same tendencies as the global programmability metrics in Table I, where OpenMP is the most expensive, particularly in terms of programming effort, followed by OmpSs. One reason for the larger complexity of OpenMP are its restrictions on what can be specified in the dependency lists, such as the

```
1  #pragma omp parallel
2  #pragma omp single
3    {
4      for (int i = 0; i < dim; i++) {
5        tile* const pivot = &tiles[dim*i+i];
6
7  #pragma omp task depend(inout:pivot[0])
8        pivot[0] = inverse(pivot[0]);
9
10       ...
11
12       for (int r = 0; r < dim; r++) {
13         if (r == i) continue;
14
15         tile* const tin = &tiles[dim*r+i];
16
17         for (int j = 0; j < dim; j++) {
18           if (j == i) continue;
19
20 #pragma omp task depend(inout:tiles[dim*r+j])
         depend(in:tin[0], tiles[dim*i+j])
21             multiply_subtract_in_place(tiles[dim*r+j],
                 tin[0], tiles[dim*i+j]);
22
23         }
24
25         ...
26       }
27
28     }
29   } // end omp single, omp parallel
```

(a) OpenMP

```
1    for (int i = 0; i < dim; i++) {
2      tile& pivot = tiles[dim*i+i];
3
4  #pragma omp task inout(pivot)
5      pivot = inverse(pivot);
6
7      ...
8
9      for (int r = 0; r < dim; r++) {
10       if (r == i) continue;
11
12       tile& tin = tiles[dim*r+i];
13
14       for (int j = 0; j < dim; j++) {
15         if (j == i) continue;
16
17 #pragma omp task inout(tiles[dim*r+j])
         in(tin, tiles[dim*i+j])
18         multiply_subtract_in_place(tiles[dim*r+j],
             tin, tiles[dim*i+j]);
19       }
20
21       ...
22     }
23   }
24
25 #pragma omp taskwait
```

(b) OmpSs

```
1    for (int i = 0; i < dim; i++) {
2      tile& pivot = tiles[dim*i+i];
3
4      spawn( [] (tile &t) { t = inverse(t); },
5             pivot);
6
7      ...
8
9      for (int r = 0; r < dim; r++) {
10       if (r == i) continue;
11
12       tile& tin = tiles[dim*r+i];
13
14       for (int j = 0; j < dim; j++) {
15         if (j == i) continue;
16
17         spawn(multiply_subtract_in_place,
18               tiles[dim*r+j], tin, tiles[dim*i+j]);
19       }
20
21       ...
22     }
23   }
24
25   wait_for_all();
```

(c) DepSpawn

Figure 3. Excerpts from the matrix inversion code implementations in OpenMP, OmpSs and DepSpawn.

non-support of C++ references or expressions that provide data to access other than scalars and array sections. For example, as shown in Fig. 2(b), in the matrix inversion code the tile (`i`,`i`) is used in three different places, and we have measured that the SLOC increase associated to defining a reference to it (one line) is much less, even in relative terms, than the Halstead programming effort cost due to selecting three times this tile from the `tiles` array. This also matches the intuition that naming an entity that is used several times makes the code more readable and maintainable. As a result the OmpSs and DepSpawn versions define a reference called `pivot` to this tile in line 2 of their codes in Fig. 3. In OpenMP the best we can do is to use a pointer, defined in line 5, which must be indexed to play the same role. While according to our measurements this is still an improvement with respect to the repeated access to the underlying tile, the solution is clearly less elegant and introduces more complexity than the use of a reference. OpenMP also needs to enclose the parallel algorithms in a parallel region and a top-level sequential task (lines 1-3 and last line in Fig. 3(a)) that are not needed in OmpSs and DepSpawn. In exchange, these two latter alternatives require an explicit synchronization in their last line to make sure that all the tasks have finished, while for OpenMP this is implicit once we exit the parallel region.

Other important difference among these approaches is that OpenMP and OmpSs require to annotate with a separate pragma the dependencies of a task, which involves more specifications and analysis than the use of `spawn` in DepSpawn. But while they can mark as tasks individual statements or code blocks, DepSpawn requires its tasks to be function calls. The fact that this library supports any kind of function, including the convenient C++11 lambda functions, ameliorates this restriction, although the result is arguably less readable than the use of directives. Our metrics reflect this, as depending on the concrete situation the programming effort of these lambdas goes from similar to noticeably larger than that of directives, particularly those of OmpSs. Our example in Fig. 3 includes both a task whose most natural expression is not a function call and another one that is a function invocation in order to help to illustrate these differences. The first task is the replacement of the pivot tile by its inverse (including the computation of such inverse). This is naturally expressed by an assignment to the pivot tile of the result of the function call that computes its inverse. In OpenMP and OmpSs we label this assignment as a task for which the pivot is both the input and the output. DepSpawn requires a function call, so we could isolate this block as a separate traditional function or, as in the example, we could express it as a C++ lambda function. Notice

how the non-constness of the function formal parameter indicates that it can both read and modify its argument, which is the pivot. The second task is an invocation to a function `multiply_subtract_in_place(a,b,c)`, which computes `a=a-b×c`. Here OpenMP and OmpSs follow the same pattern as before, while DepSpawn benefits from the existence of the function, which makes the `spawn` much simpler. Regarding the OmpSs-decl style, if this were the only place where this function is used in the application, or the user added the compiler directives necessary to allow all its other uses to run safely in parallel, the OmpSs version could avoid the annotations in the invocations of this function just by declaring it as

```
#pragma omp task inout(a) in(b, c)
void multiply_subtract_in_place(tile &a, const tile& b, const tile& c);
```

Comparing this directive to line 17 in Fig. 3(b) explains the noticeable decrease of the programming effort of OmpSs-decl with respect to OmpSs-inv in Table I. This justifies the annotation of function declarations as a worthwhile feature of OmpSs. Still, making a function automatically parallel wherever it appears has dangers, such as if a user forgets its parallel nature in another context, and it implies synchronization requirements in all the uses of the function. Both issues should not be underestimated, particularly in large software projects and libraries.

Altogether, the tendency observed is that the best metrics are achieved by DepSpawn, followed by OmpSs-decl, OmpSs-inv, and OpenMP in this order, the differences between them being medium to small. Focusing on the nature of the codes, when function calls predominate in them, which is the case in Cholesky, DepSpawn offers the best programmability for the two metrics measured. But when it is more natural to express a meaningful portion of the tasks as statements or blocks, which is the style in which the matrix inversion code is written, things can be different. For example, according to Table I, in this algorithm the programming effort advantage of DepSpawn with respect to the other alternatives was smaller and it matched OmpSs in terms of SLOCs, the reason being the (lambda) function declarations not required by the compiler directives for code blocks.

B. Performance

We used two Linux systems in our tests. The first one has 2 Intel Xeon E5-2660 Sandy Bridge processors at 2.2Ghz with 8 cores and 20MB L3 cache each, and 64GB of DDR3 memory. The second one has 2 Intel Xeon E5-2680 Haswell processors at 2.5GHz, with 12 cores and 30MB L3 cache each, and 64GB of DDR4 memory. The compilers used are g++ 4.9.2 and 4.9.1, respectively, with optimization flag `O3`. These compilers provide the OpenMP support tested. It must be noted that, being a standard, there are many implementations of OpenMP. The implementation tested, belonging to the ubiquitous gcc tool chain, is one of the best supported and most popular and widely used ones, being thus very representative. For OmpSs we installed its compiler and runtime version 15.06, while the DepSpawn and Intel CnC versions are 1.0 and 1.0.100, respectively. The BLAS functions used were provided by the high quality OpenBLAS library v0.2.14 and all the experiments were made using double-precision floating point data. We found no performance differences between OmpSs-inv and OmpSs-decl, so they are discussed jointly in this Section.

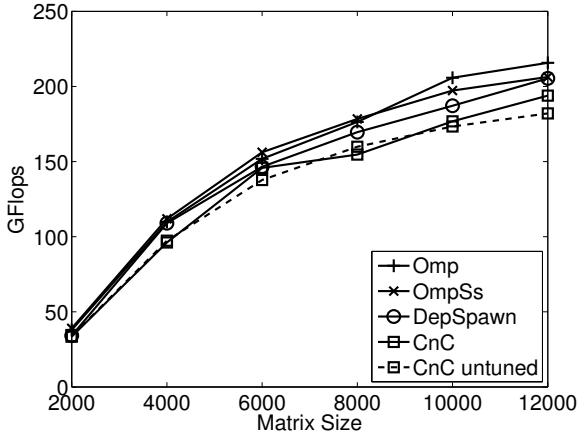

Figure 4. Cholesky performance in the Sandy Brigde platform

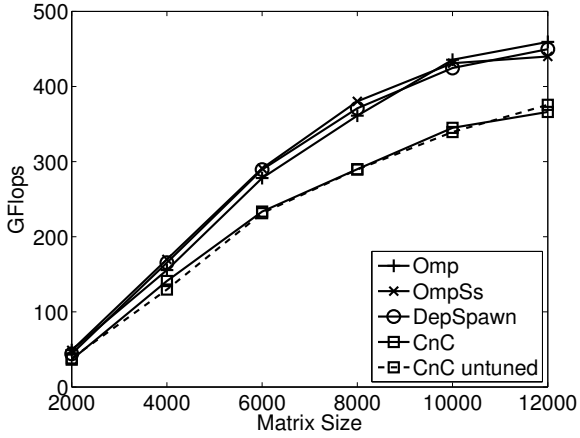

Figure 5. Cholesky performance in the Haswell platform

Figures 4 and 5 show the performance that our five versions of the Cholesky factorization achieve in the two platforms tested as a function of the matrix size when using all the cores they provide, 16 in the Sandy Bridge system and 24 in the Haswell system. Each experiment was run using tiles of size 200×200, 250×250, 400×400 and 500×500, and repeating the measurement three times. The figure reports at each point the best performance achieved for any tile size. Figures 6 and 7 show the performance of the matrix inversion algorithm obtained following the same

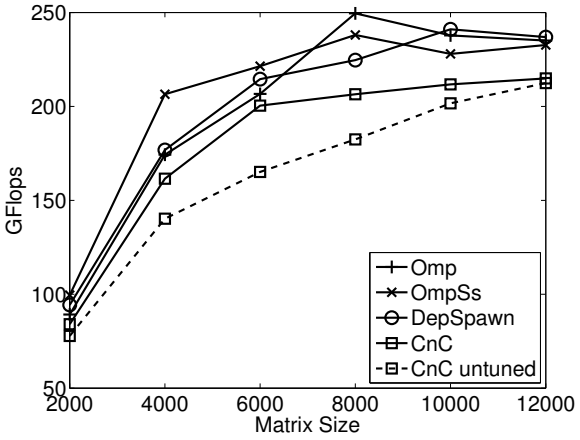

Figure 6. Matrix inversion performance in the Sandy Brigde platform

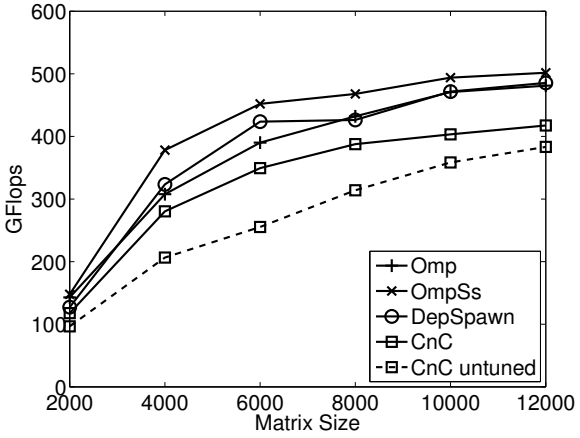

Figure 7. Matrix inversion performance in the Haswell platform

methodology. Interestingly, while the tuners played no clear role for CnC in the first algorithm, they were crucial in the matrix inversion. But even with them, CnC tends to be in the low range of performance, because of the larger complexity of its programs, which require additional operations.

As mentioned before, a deep analysis of the performance and overheads of CnC in the matrix inversion code can be found in [33]. Although it is not possible to directly compare their results with ours due to the differences in hardware, software versions and linear algebra kernels used, there are some similarities. For example, the overhead they measure for the tuned CnC with respect to the underlying TBB (12%-15%) is similar to the speed difference we measured for this code with respect to the also TBB-based DepSpawn, 7%-13% in the Sandy Bridge and 8%-21% in the Haswell. In our experiments, however, the tuners increased the performance of this CnC code by 11% in the Sandy Bridge and 23% in the Haswell, while the improvement in [33] was a more modest 6%. We attribute this latter fact mostly to the larger number of cores we use (16 and 24 vs 8) and the higher quality of the very optimized BLAS dgemm routine we use, which accounts for the vast majority of the runtime (all the computations except `inverse` in Fig. 2(b)), as both facts stress much more the runtime system of the parallelizing solution used.

There is not a clear performance relation among the other alternatives. The maximum performance difference among them for Cholesky happens for the smallest matrix size, and it is 14% in the Sandy Bridge and 11% in the Haswell. Also, while DepSpawn is always the slowest one in the Sandy Bridge for this algorithm, there is a virtual match among the three versions in the Haswell, with OpenMP achieving the lowest performance for half of the matrix sizes. The maximum performance difference in the matrix inversion between the slowest and the fastest alternative in this group grows to 18% and 23% in the Sandy Bridge and Haswell platforms, respectively, and they happen for the 4000×4000 matrix. Although there is not a clear pattern among OpenMP, OmpSs and DepSpawn for this algorithm in the Sandy Bridge, this is not the case in the Haswell. Here, OpmpSs consistently offers the best performance and DepSpawn usually has an intermediate, or almost identical position with respect to OpenMP.

IV. Related work

A line of research related to this work are parallel skeletons [9], [17], as they are higher order functions that implement the flow of execution, parallelism, synchronization and data communications of typical strategies for the parallel resolution of problems. This way, the use of a given skeleton implicitly expresses all the dependences among the parallel tasks that conform it and it is responsible for all the details of the underlying parallelization. Nevertheless, because skeletons implement fixed collections of preset patterns, they are not general enough to optimally express any task graph, which is an essential part of our motivation.

CnC was compared to other approaches in [8], but only in terms of performance, and it did not include other programming tools based on implicit dependences and synchronizations. More recently, the performance of different runtimes for OpenMP dependent tasks and multithreaded libraries was compared in [36]. As for the programmability issues, the general ease of use of OpenMP has been discussed in [23], [15], and lists of typical mistakes and good practices have been compiled in [32]. A motivation, design description and performance evaluation of OpenMP 3.0 tasks is found in [2]. The manuscript precedes the inclusion of the `depend` clause in OpenMP 4.0, and thus it does not cover the implicit dependencies it enables, critical for our work. This latter observation also pertains to [25], which compares task parallelism under several parallel frameworks based on explicit synchronizations, including OpenMP 3.0 and TBB, which is the backend for the two libraries we have tested.

There are many other tools that support the paradigm studied in this paper. For example, there has been extensive research on this subject in the field of functional [21], [22] and dataflow [10] programming, but we have focused on imperative languages, as they are more commonly used in parallel applications. While some imperative languages were proposed based on this approach [30], [13], most of the research on data-flow task programming developed libraries with semantics similar to those of the tools discussed in this paper. Examples are [7], exclusively focused on linear algebra, [37], used in the backend of the PLASMA library, or [1], particularly focused on heterogeneous computing. There are also projects that provide both a library to build tasks that express their dependencies by means of its API datatypes [14] and a runtime for OpenMP [4].

Out of the scope of this work is the large set of proposals to facilitate the creation and management of tasks that require users to explicitly handle the dependencies and synchronizations between them, at least when the tasks dependencies do not form a simple pattern. Examples of this kind of tools are [29], [12], [31], [24] to name a few. The mechanisms used to specify the synchronizations and dependencies between the tasks include language constructs, keywords, library functions, etc.

V. Conclusions

Declaratively specifying the abstract dependencies of the parallel tasks that conform our application and letting a runtime automatically detect and enforce the actual dependencies generated during the execution is a very attractive alternative. This strategy not only requires less programmer time and gives place to more maintainable codes that lower-level more explicit alternatives, but it is widely applicable and it allows an optimal exploitation of the underlying hardware depending on the quality of the scheduling algorithms of that runtime. While there are performance evaluations of tools of this kind, we have not found discussions on the relative advantages and limitations of proposals that support this paradigm. We have thus tackled this issue in this paper.

Two families of compiler directives, OpenMP and OmpSs, and two C++ libraries, DepSpawn and CnC, were analyzed. As a result, this study mostly focused on this language, extensively used in parallel computing. Our findings can be summarized as follows. CnC was the option that clearly required more programming effort and offered a more modest performance, even when the applications were optimized with tuners. It has though the unique advantage that it is the only one that (publicly, at least) provides a path to execute its applications in distributed-memory systems, although this requires some additional coding. Therefore this is the tool advised for hybrid and distributed memory environments, at least while the OmpSs support for clusters is not publicly available.

There was not a clear performance relation among the other tools, and the differences between them were small in most cases. In fact the average performance difference between the slowest and the fastest implementation among them in our tests was 8.8%. Regarding programmability, DepSpawn was the best positioned, and OpenMP the worst one, although there were no large differences either. In view of this, OpenMP seems the default advised approach, given its large support and portability, although the impossibility to use references or expressions in the dependencies makes it less friendly than OmpSs and DepSpawn. Another meaningful shortcomings of OpenMP are the lack of support for data members in its clauses and for annotating function declarations. Finally, DepSpawn is the only option in this group when we need to respect dependencies on arbitrarily overlapping data items, including array regions, as it is the only one that supports this feature.

Among the most interesting lines of development for these tools are the support of distributed-memory systems for those that are missing it, and the possible inclusion of futures [19] as another mechanism of implicit synchronization.

Acknowledgements

This work was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the European Union (Project TIN2013-42148-P) as well as by the Xunta de Galicia under the Consolidation Program of Competitive Reference Groups (Ref. GRC2013/055). The author also wants to thank Dr. Thierry Gautier for his help and CESGA (Galicia Supercomputing Center, Santiago de Compostela, Spain) for providing access to the Intel Xeon E5-2680 Haswell platform.

References

[1] 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.

[2] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, March 2009.

[3] *OmpSs Specification*, Barcelona Supercomputing Center, Dec 2015.

[4] F. Broquedis, T. Gautier, and V. Danjean, *OpenMP in a Heterogeneous World: 8th Intl. Workshop on OpenMP, (IWOMP 2012)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, ch. libKOMP, an Efficient OpenMP Runtime System for Both Fork-Join and Data Flow Paradigms, pp. 102–115.

[5] 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.

[6] M. G. Burke, K. Knobe, R. Newton, and V. Sarkar, "The concurrent collections programming model," Dept. of Computer Science, Rice University, Tech. Rep. TR 10-12, 2010.

[7] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn, "SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks," in *13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP'08, 2008, pp. 123–132.

[8] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *2010 IEEE Intl. Symp. on Parallel Distributed Processing (IPDPS)*, 2010, pp. 1–12.

[9] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1991.

[10] J. B. Dennis, "Data flow supercomputers," *Computer*, vol. 13, no. 11, pp. 48–56, Nov 1980.

[11] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta, "A proposal to extend the OpenMP tasking model with dependent tasks," *Intl. J. Parallel Program.*, vol. 37, no. 3, pp. 292–305, 2009.

[12] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *ACM SIGPLAN 1998 Conf. on Programming Language Design and Implementation*, ser. PLDI '98, 1998, pp. 212–223.

[13] F. Galilee, G. Cavalheiro, J.-L. Roch, and M. Doreille, "Athapascan-1: On-line building data flow graph in a parallel language," in *1998 intl. conf. on Parallel Architectures and Compilation Techniques (PACT'98).*, Oct 1998, pp. 88–95.

[14] T. Gautier, J. Lima, N. Maillard, and B. Raffin, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IEEE 27th Intl. Symp. on Parallel Distributed Processing (IPDPS)*, 2013, pp. 1299–1308.

[15] R. Gonçalves, M. Amaris, T. K. Okada, P. Bruel, and A. Goldman, "OpenMP is not as easy as it appears," in *49th Hawaii Intl. Conf. on System Sciences (HICSS 2016)*, jan 2016, pp. 5742–5751.

[16] 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.

[17] S. Gorlatch and M. Cole, "Parallel skeletons," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 1417–1422.

[18] M. H. Halstead, *Elements of Software Science*. New York, NY, USA: Elsevier, 1977.

[19] R. H. Halstead, Jr., "MULTILISP: A language for concurrent symbolic computation," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 4, pp. 501–538, Oct. 1985.

[20] M. Herlihy, J. Eliot, and B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *20th Annual Intl. Symp. on Computer Architecture*, 1993, pp. 289–300.

[21] R. Loogen, Y. Ortega-Mallén, and R. Peña Marí, "Parallel functional programming in Eden," *J. Funct. Program.*, vol. 15, no. 3, pp. 431–475, May 2005.

[22] S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. Trinder, "Seq no more: better strategies for parallel Haskell," *SIGPLAN Not.*, vol. 45, no. 11, pp. 91–102, Sep 2010.

[23] T. G. Mattson, "How good is OpenMP," *Sci. Program.*, vol. 11, no. 2, pp. 81–93, Apr. 2003.

[24] S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical workstealing on manycore clusters," in *Fifth Conf. on Partitioned Global Address Space Programming Models (PGAS 2011)*, Oct 2011.

[25] S. Olivier and J. Prins, "Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs," *Intl. J. Parallel Program.*, vol. 38, no. 5-6, pp. 341–360, 2010.

[26] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," July 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[27] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *2008 IEEE Intl. Conf. on Cluster Computing*, Oct 2008, pp. 142–151.

[28] L. Rauchwerger and D. Padua, "The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 2, pp. 160–180, 1999.

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

[30] M. Rinard, D. Scales, and M. Lam, "Jade: a high-level, machine-independent language for parallel programming," *Computer*, vol. 26, no. 6, pp. 28–38, 1993.

[31] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove, "X10 language specification," IBM, Tech. Rep., Dec 2015. [Online]. Available: http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf

[32] M. Süß and C. Leopold, "Common mistakes in OpenMP and how to avoid them: A collection of best practices," in *2005 and 2006 Intl. Conf. on OpenMP Shared Memory Parallel Programming*, ser. IWOMP'05/IWOMP'06, 2008, pp. 312–323.

[33] P. Tang, "Measuring the overhead of Intel C++ Concurrent Collections over Threading Building Blocks for Gauss-Jordan elimination," *Concurrency and Computation: Practice and Experience*, vol. 24, no. 18, pp. 2282–2301, 2012.

[34] C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan, "Evaluation of UPC programmability using classroom studies," in *3rd Conf. on Partitioned Global Address Space Programing Models*, ser. PGAS '09, 2009, pp. 10:1–10:7.

[35] T. L. Veldhuizen, "Arrays in Blitz++," in *2nd Intl. Scientific Computing in Object-Oriented Parallel Environments (ISCOPE98)*. Springer-Verlag, 1998, pp. 223–230.

[36] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier, *10th Intl. Workshop on OpenMP (IWOMP 2014)*. Springer International Publishing, 2014, ch. Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite, pp. 16–29.

[37] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users guide: Queueing and runtime for kernels," Dept. of Computer Science, University of Tennessee, Tech. Rep. TR ICL-UT-11-02, 2011.