A framework for argument-based task synchronization with automatic detection of dependencies

Carlos H. González*, Basilio B. Fraguela

Depto. de Electrónica e Sistemas. Universidade da Coruña. Facultade de Informática, Campus de Elviña, S/N. 15071. A Coruña, Spain

Abstract

Synchronization in parallel applications can be achieved either implicitly or explicitly. Implicit synchronization is typical of programming environments that provide predefined, and often simple, patterns of parallelism such as data-parallel libraries and languages and skeletal operations. Nevertheless, more flexible approaches that allow to express arbitrary task-level parallel computations without a predefined structure request in turn that the user explicitly specifies the synchronization needed among the parallel tasks.

In this paper we present a library-based approach that enables arbitrary patterns of parallelism with minimal effort for the user. Our proposal is the first generic approach to express parallelism we know of that requires neither explicit synchronizations nor a detail of the dependencies of the parallel tasks. Our strategy relies on expressing the parallel tasks as functions that convey their dependencies implicitly by means of their arguments. These function arguments are analyzed by our library, called DepSpawn, when a parallel task is spawned in order to enforce its dependencies. Our experiments indicate that DepSpawn is very competitive, both in terms of performance and programmability, with respect to a widespread high-level approach like OpenMP.

Keywords: Parallel programming, synchronization, out-of-order execution, libraries, dependencies, programming models

1. Introduction

One of the outstanding difficulties of the development of parallel applications is the synchronization of the different units of execution. Different parallel programming models and tools provide different synchronization techniques, which can be implicitly or explicitly invoked by the user. The data-parallel paradigm [1, 2, 3], for example, provides implicit synchronization points after each pair of parallel computation and assignment, being this one of the reasons why this paradigm is one of the most effective in terms of intuitiveness for the programmer. Unfortunately, the patterns of parallelism supported by this paradigm are too restrictive for many parallel applications. It is also the case that applications that greatly benefit from data-parallelism can further increase their performance when they are enriched with the possibility of exploiting more ambitious out-of-order scheduling [4, 5, 6] for the tasks in which their computations can be decomposed. Applications that benefit from complex patterns of parallelization and scheduling have usually to resort to paradigms and tools where explicit synchronization mechanisms are available in order to ensure that tasks satisfy their dependencies [7, 8, 9]. The implied required study of the dependencies among the tasks, and the subsequent specification of the corresponding synchronizations, result in an increased programming complexity, which is the cost to pay for the flexibility in the patterns of parallelism and dependencies among such tasks.

*Corresponding author. Tel: +34 981 167000 ext. 1376; fax +34 981 16 71 60

Email addresses: cgonzalezv@udc.es (Carlos H. González), basilio.fraguela@udc.es (Basilio B. Fraguela)

Fostered by the growing importance of exploiting the increasing number of cores and accelerators available in current computers, there have been several proposals to alleviate the aforementioned difficulties. Some of them are libraries that do not define a specific programming model and whose scope of application is restricted to a limited area [4, 5, 6]. Other projects [10, 11, 12, 13] have developed or improved already existing families of compiler directives so that users can annotate which portions of their applications can be run as parallel tasks, and which are their dependencies. Based on these annotations, a compiler adds the suitable code to ensure that at runtime tasks are only submitted to execution once the dependencies stated by the programmer are fulfilled. While these annotations are a big step forward, the user is still preoccupied with the analysis of the tasks in order to identify not only their inputs and outputs, but also the exact shape and extent of the region accessed in the case of arrays, which are by far one of the most frequent data structure found in task dependencies. Also, the requirement to use a specific compiler, at least while these directives are not widely adopted, places more restrictions on development and portability than the simple use of libraries.

In this paper we introduce a practical library-based approach to enable the expression of arbitrary patterns of parallel computation while avoiding explicit synchronizations. Our proposal is almost as flexible as explicit synchronization but without the complexity it brings to parallel programming. It also frees programmers from having to explicitly indicate which are the dependencies for their tasks. Our proposal, which has been implemented in the widely used C++ language, relies on the usage of functions to express the parallel tasks, the dependencies among the functions being solely provided by their arguments. This way, an analysis of the type of the arguments of a function whose execution has been requested, coupled with a comparison of these arguments with those of the functions that have already been submitted to execution, suffices to enforce the dependencies among them.

The rest of this paper is organized as follows. The next section explains the principles of our proposal. Then Sect. 3 describes the algorithms used to test the library in Sect.4 both in terms of performance and programmability. Sect. 5 discusses the related work, and finally Sect. 6 is devoted to our conclusions and future lines of work.

2. DepSpawn : An argument-based synchronization approach

As we have anticipated, our proposal is based on expressing the parallel tasks as functions that only communicate through their arguments. The types and memory positions of the arguments of the functions that define each parallel task are analyzed by our framework, called DepSpawn, to detect and enforce the dependencies among those tasks. Our library has three main components:

- The kernel is the function `spawn`, which requests a function to be run as a parallel task once all the dependencies on its arguments are fulfilled.
- The template class `Array`, which allows to conveniently express a dependency on a whole array or a portion of it.
- A few explicit synchronization functions to wait for the completion of all the pending tasks, to wait only for those that access some specific arguments, and to allow a task to early release dependencies.

We now describe these components in turn, followed by a small discussion on the implementation of the library.

2.1. Spawning parallel tasks

Function `spawn` accepts as first argument the name of the function whose execution as a parallel task is requested, followed by the comma-separated list of the arguments to the function. Since all the communications between parallel tasks must take place through their arguments, the function should have return type `void`, that is, act as a procedure. Functions returning a value are also accepted by `spawn`, but the returned value will be lost, as `spawn` does not return any value. It is worth mentioning that our library does not make any assumption regarding global variables. Thus, if the user wants to track dependencies on these variables,

Table 1: Behavior of `spawn` for each parameter type. `A` is any arbitrary data type. Modifiers in brackets do not change `spawn`'s behavior.

Type	Short description	Interpretation of `arg`
`[const] A arg`	Argument by value	input
`A& arg`	Argument by reference	input and output
`const A& arg`	Argument by constant reference	input
`[const] A* arg`	Pointer by value	input (not `*arg`)
`[const] A*& arg`	Pointer by reference	input and output (not `*arg`)
`[const] A* const & arg`	Pointer by constant reference	input (not `*arg`)

she should pass them as parameters to the corresponding functions. The type of the corresponding function parameter will indicate whether the variable will only be read or can be also modified.

Table 1 summarizes the interpretation that `spawn` makes of each parameter or formal argument of a function depending on its type. Arguments passed by value are necessarily only inputs to the function, since any change made on them in the function is actually performed on a local copy. As for arguments passed by reference, which are marked in C++ by preceding the parameter name by a '`&`' symbol, the situation depends on whether the reference is constant or not. Constant references, which are labeled with the `const` modifier, do not allow to modify the argument, so they can only be inputs. Non-constant references, however, can modify the argument, therefore they are regarded as inputs and outputs. Table 1 reflects that pointers are regarded as inputs or inputs and outputs following the same rules as any other data type, and that the dependency is defined on them, not on the memory region they point to. This behavior has been chosen for two reasons: first, it is consistent with the treatment of the other data types; second, the library cannot make reasonings on the dependencies generated through the usage of a pointer, as it is impossible to know the extent of the memory region that will be accessed through a pointer from its declaration.

Given the aforementioned interpretation associated to each function argument, and using the information on its address and length, `spawn` learns the data dependencies between the tasks submitted to parallel execution. It then guarantees that each task is run only once all its dependencies with respect to the previously spawned tasks are satisfied. In this regard, it must be outlined that our library releases the dependencies of a task only when the task itself and all its descendants (i.e. the tasks directly or indirectly spawned from it) finish. This way the release of a dependency by a task implies that all what would have been part of the serial execution of the task has been executed.

In order to formally detail the semantics of the programming model provided by our library, we make the following definitions:

- We say that a task T is *requested* when, during the execution of the program, the corresponding `spawn` invocation is made. A requested task can be waiting for its dependencies to be fulfilled, executing, or finished.
- A task T is an *ancestor* of task U if T requests U or, recursively extending this definition, it requests an ancestor of U.
- A task T is a *descendant* of a task U if U is an ancestor of T.
- We call *sequential execution* of the program the one that takes place if the spawn requests are replaced with standard function calls.
- A task T *logically precedes* task U if during the sequential execution of the program T is executed before U.
- A task T *dynamically precedes* task U if during the execution of the program T is requested before U (by the same or another thread of execution).
- A task T is said to have a *dependency* with a preceding task U if there is at least one memory position in common in their arguments that at least one of them could modify.

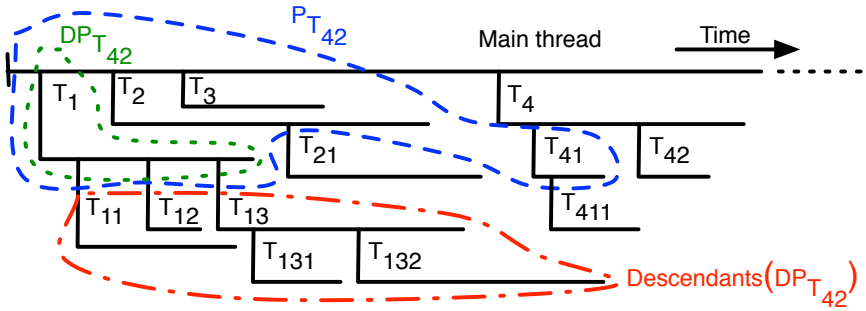

Figure 1: Preceding tasks $P_{T_{42}} = \{T_1, T_2, T_3, T_{41}\}$ for task T_{42}. Assuming that out of them only T_1 generates dependencies with T_{42} (i.e. $DP_{T_{42}} = \{T_1\}$), Descendants$(DP_{T_{42}}) = \{T_{11}, T_{12}, T_{13}, T_{131}, T_{132}\}$ is also depicted.

In our framework a requested task T waits for the completion of a task U if

a. U dynamically precedes T, U is not an ancestor of T, and T has a dependency with U; or
b. U is a descendant of a task that fulfills condition (a).

Condition (b) is derived from the fact that in our framework a task does not release its dependencies until all its children finish their execution. Let us now analyze the implications of these conditions. Condition (a) indicates that a new task respects all the dependencies with the tasks that were requested before during the dynamic execution of the program, excluding of course its ancestors. In order to provide a formal definition of the guarantee provided by this condition, we will name tasks according to their order of execution in the sequential execution of the program. This way, we will call T_i the i-th task initiated during the sequential execution of the program that is not nested inside another task, T_{ij} will be the j-th task initiated by task T_i that is only nested in task T_i, and so on. Using this nomenclature, condition (a) only guarantees that task $T = T_{x_0 x_1 \ldots x_n}$ complies with the dependencies with the preceding tasks $P_T = \{T_{x_0 x_1 \ldots x_{i-1} j}\ , \forall 0 \leq i \leq n, 1 \leq j < x_i\}$. The reason is that these are the only tasks that we know for sure that are requested before T in any execution. Among them, the focus in on the subset of those that actually generate dependencies with T, defined as $DP_T = \{U \in P_T \ / \ T \text{ has a dependency with } U\}$.

Additionally, condition (b) guarantees that T is executed after the tasks that have an ancestor in DP_T, that is, Descendants(DP_T), where Descendants$(S) = \{T_{x_0 x_1 \ldots x_m x_{m+1} \ldots x_n} \ / \ T_{x_0 x_1 \ldots x_m} \in S, n > m\}$. Notice that both the tasks in DP_T and Descendants(DP_T) necessarily logically precede T.

Figure 1, in which time runs from left to right and the spawn of new tasks is represented as a vertical line that leads to a new horizontal line of execution, helps illustrate these ideas. The tasks are labeled using the naming just defined, and it shows a concrete temporization in a parallel execution of an application. Following our definition, $P_{T_{42}} = \{T_1, T_2, T_3, T_{41}\}$ for task T_{42}, as these are the tasks that are not ancestors of T_{42} that will be requested before it no matter which is the exact length or temporization of execution of the different tasks. In this figure we assume that out of these four tasks, T_{42} only has dependencies with respect to T_1, thus $DP_{T_{42}} = \{T_1\}$. Finally, the figure also depicts Descendants$(DP_{T_{42}})$, which are all the descendants of T_1. According to the rules just described, T_{42} will not start running until T_1 and all its descendants finish. Nevertheless, since T_2, T_3 and T_{41} are guaranteed to be analyzed before T_{42} is requested, and the example assumes that T_{42} has no dependencies with them, they can run in order, including in parallel, with respect to T_{42}.

This programming model does not give guarantees of relative order of execution with respect to the tasks that logically precede T that do not belong to the sets we have defined. Such tasks are all the descendants of the tasks in P_T that do not generate dependencies with T, that is, Descendants$(P_T - DP_T)$. The relation guaranteed with respect to these tasks is of mutual exclusion if there is at least a common element of conflict, i.e., one piece of data modifiable by either T or the considered task according to their respective lists of arguments. Going back to our example in Fig. 1, the tasks in this situation with respect to T_{42} are T_{21} and T_{411}. In the execution shown in the figure they are requested before T_{42}, but in other runs any of these two tasks, of both, could be requested after T_{42}. This way, the relative order in which these three tasks will run

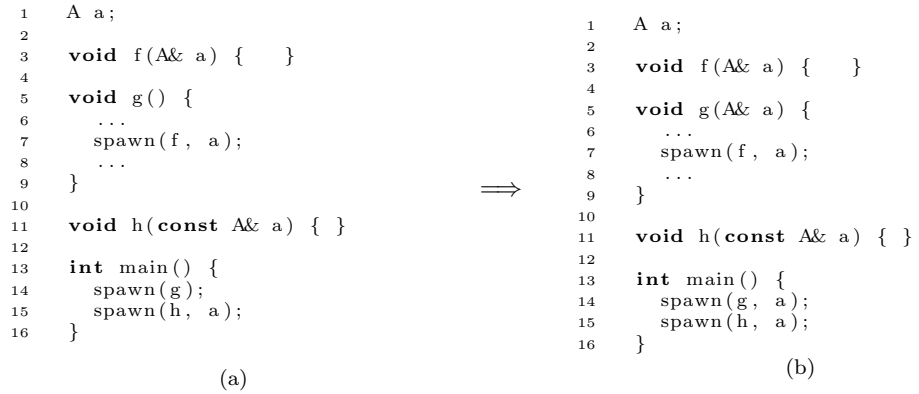

Figure 2: Enforcing dependencies between tasks (a) Wrong (b) Right

cannot be predicted, but we are guaranteed that no pair of them with a conflict in their arguments will run in parallel.

The result of the described semantics is a simple rule of thumb for the programmer. Namely, a task T is guaranteed to fulfill dependencies with respect to the tasks requested before it, inside the same task, or before any of its ancestors, in the respective tasks where they were requested (condition(a)), which is an informal definition for P_T. Task T also respects its dependencies with the tasks that are descended from those tasks in P_T that presented dependencies with T, DP_T (condition(b)). For any other task U, it is guaranteed that U will be not be run in parallel with T if there is any memory position common to the arguments of T and U in which one of them writes. This behavior suffices in many situations. If it is not the case, the user has two possibilities to enforce the ordering between U and T. The most natural one is to express the dependency in an ancestor of U that belongs to P_T so that it becomes part of DP_T. The other one is to use one of the explicit synchronization mechanisms provided by our library, which are detailed in Sect. 2.3.

Figure 2 exemplifies how to enforce a proper ordering between tasks. In Fig. 2(a) the master thread creates two tasks, `g` and `h` that carry no dependencies between them according to their arguments, so they can be run in parallel. During the execution of `g`, a new task `f` is requested that writes in a global object `a` used by task `h`. The system guarantees that `f` and `h` will not run in parallel, but it does not enforce any specific order of execution. Since `f` logically precedes `h` we probably want to make sure it runs before. This is achieved in a natural way by expressing the dependencies generated by `f` in the arguments of any ancestor spawned before `h`. Here such ancestor is `g`, giving place to the code in Fig. 2(b).

2.2. Array support

The library behavior has been explained using generic data types, which can be standard types, user defined classes or arrays. The analysis performed by our library each time a parallel task is spawned treats all data types equally, checking the starting memory position and the size of each argument for overlaps with other variables. This permits expressing any kind of parallel computation, serializing tasks that access the same object when at least one of them writes to it. This raises an important question. Some objects are actually aggregates or containers of other objects, and the fact that multiple parallel tasks operate on them does not imply there are actually data dependencies among those tasks. For example, many parallel algorithms make use of arrays whose processing is distributed among different tasks that read and write to disjoint regions of these arrays. Therefore, just checking the full object or array is not a flexible strategy, as this would serialize these actually independent tasks. One solution would be to distribute the data in the original array in smaller independent arrays so that each one of them is used by a different task, but this introduces non-negligible programming (and sometimes performance) overheads, and depending on the algorithm it is not always possible. Another solution, which is the one we have implemented, is to provide a data type that allows to express these arrays, to efficiently define subsections of them without copying

```
// Two dimensional matrix of 64x64 floats
Array<float, 2> array(64, 64);

// Subarray from position (10,0) to position (20, 30)
Array<float, 2> subarray = array(Range(10, 20), Range(0, 30));
```

Figure 3: Example of definition of an array and a subarray.

```

  void mxm(Array<float, 2>& result,
           const Array<float, 2>& a,
           const Array<float, 2>& b)
  {
    const int nrows = result.rows();
    const int ncols = result.cols();
    const int kdim = a.cols();

    for(int i = 0; i < nrows; i++) {
      for(int j = 0; j < ncols; j++) {
        float f = 0.f;
        for(int k = 0; k < kdim; k++)
          f += a(i, k) * b(k, j);
        result(i, j) = f;
      }
    }
  }

  ...
  for(int i = 0; i < N; i += N / BLK) {
    int limi = (i + N / BLK) >= N ? N : (i + N / BLK);
    Range rows(i, limi - 1);
    for(int j = 0; j < N; j += N / BLK) {
      int limj = (j + N / BLK) >= N ? N : (j + N / BLK);
      Range cols(j, limj - 1);
      spawn(mxm, result(rows, cols), a(rows, Range::all()), b(Range::all(), cols));
    }
  }
  ...
```

Figure 4: Usage of the `Array` class to enable the parallel processing of independent tasks.

data, and which is known to our dependencies analysis framework so it can retrieve the range of elements an array of this class refers to in order to check for overlaps.

In order to provide this support, we have developed a modified version of the `Array` class of the Blitz++ library [14]. Blitz++ implements efficient array classes and operations for numeric computations. Its `Array`, illustrated in Fig. 3, provides multiple indexing schemes that allow to define sub-arrays that reference a bigger matrix (i.e. they point to the same data). Our `Array` class, derived from the one provided by Blitz++, enables our task spawn framework to check for overlapping subarrays that reference the same block of memory.

An example of a typical usage of the `Array` class is shown in Fig. 4. This code subdivides the multiplication of two square matrices of `N`×`N` elements in `BLK`×`BLK` parallel tasks, each one being in charge of computing one portion of the output array `result`. The multiplication itself is performed in function `mxm`, which retrieves the dimensions of the arrays using their interface and accesses their scalar elements using operator `()`. The aim of this example is to illustrate the simple and powerful interface offered by `Array`. A high-performance implementation should obtain the pointers from the arrays involved in the multiplication and invoke a specialized function such as `gemm` from BLAS. This is in fact the way we developed the applications used in the evaluation in Sect. 4.

2.3. Explicit synchronization facilities

DepSpawn also provides three functions to control the synchronization process at a lower level so that the programmer can make some optimizations:

- `void wait_for_all()` makes the current thread wait until all the spawned tasks finish. Its intended use is to serve as a barrier where the main program can wait while the spawned processes do their operations.
- `void wait_for(Type vars...)` provides a more fine grained synchronization, where the current thread only waits for the tasks that generate dependencies on the variables specified as arguments to finish. There can be an arbitrary number of these variables and they can be of different types.
- `release(Type vars...)` can be used by a spawned task to indicate that the processing on some variables has ended, so that the dependent tasks can begin to run before this task actually finishes its execution.

2.4. Implementation details

A first decision in any software project is choosing the implementation language. While there are more widespread languages such as C, DepSpawn was implemented in C++ mainly for four reasons:

- The nuclear idea of our library is to represent the parallel tasks by means of functions that express all their dependencies through their arguments. As a result, the function outputs should be provided through their arguments. This requires either resorting to pointers, which is the only option in C, or the ability to pass arguments by reference. Since it in unfeasible to automatically analyze dependencies among pointers using a library for the reasons explained in Sect. 2.1, we had to choose a language that provides pass by reference, which is the case of C++.
- We wanted our library to be as automated and to require as minimal user intervention as possible. This way, we preferred to use a language such as C++, with metaprogramming capabilities that allow a library to automatically analyze the types of the arguments of a function. In a language without this ability, the user would have to explicitly indicate to the library the number of arguments, as well as their intention and size, for each function to spawn.
- C++ templates allow to move many computations from runtime to compile time, particularly those related to types, which play an important role in this library, resulting in improved performance.
- An object oriented language that allows operator overloading is required to implement a class such as `Array` providing a nice notation. If the language further enables template classes, as C++ does, a single implementation of the class allows to provide support for generic arrays of any underlying type and different numbers of dimensions.

Let us notice that the interoperability between C and C++ is very strong, so existing C code can be easily used in applications written with DepSpawn. A good example of this is the usage of the standard BLAS (FORTRAN with a C interface) and GotoBLAS2 libraries (written in C) in the codes used in this paper (see Sects. 3 and 4), which was straightforward.

Once we opted for C++, we exploited its advantages to the fullest, including core language functionality improvements provided by the new C++11 standard. For example, the new variadic templates enabled us to implement `spawn` so that it accepts an arbitrary number of arguments.

Another important feature of our library is that it does not give place to busy-wait situations. Rather, tasks with pending dependencies are stored so that they will be automatically launched to execution once all their dependencies are fulfilled. After storing such tasks, the corresponding threads of execution that encountered those requests proceed to the next instruction following the `spawn` invocation without further delays.

The data associated to the tasks and their dependencies are stored in structures that are internal to the library and which are updated each time a spawn is made or a tasks finishes. These structures are shared by all the threads so that all of them can keep track of the current set of dependencies to abide by and to remove the ones generated by each task when its execution finishes. This way, there is not a control thread in charge of these structures and deciding the tasks to spawn in each moment. Rather, all the threads

```
1  barnes-hut {
2    Bodies[N];
3    while final time not reached {
4      spawn(create_quad-tree, Bodies);
5      for each block from Bodies:
6        spawn(compute_forces, block)
7      spawn(update, Bodies)
8    }
9    wait_for_all();
10 }
```

Figure 5: Pseudocode of the parallel implemenation using `spawn` of the Barnes-Hut algorithm

operate on these structures, of course with proper synchronization, and launch new tasks to execution when their dependencies are satisfied.

Regarding the threading and load-balancing mechanism, our library is built on top of the low level API of the Intel Threading Building Blocks (TBB) [8] library[1]. One of the main benefits of this approach is that our library enjoys the smart load balancing and task affinity strategies provided by TBB. For example, instead of a global task queue that could be a source of contention, each thread has its local pool of tasks. This besides promotes locality because the tasks spawned by a given thread are more likely to be run by this thread, which increases the potential to reuse data accessed by the ancestor tasks in the caches of the same core. TBB also provides a clever task stealing mechanism that allows to perform load balancing between threads in a completely transparent way. Idle threads steal the least recently generated tasks of busy threads, which are the tasks with less locality and potentially more work. Finally, TBB also helps deal with the memory model of the underlying hardware under the control of its API, which provides accesses with the release and acquire semantics as well as sequentially consistent accesses that act as memory fences. Our library uses this API to guarantee the correct execution both of its internal code as well as the user functions under any memory model.

3. Tested algorithms

We first tested the correctness of our implementation with synthetic toy programs that covered all the possible combinations of task dependencies based on their input and output arguments. Then we have implemented several algorithms to test its performance and programmability. The next subsections briefly explain these algorithms.

3.1. N-body simulation using Barnes-Hut

This force-calculation algorithm employs a hierarchical data structure, called an quadtree, to approximately compute the force that the n bodies in a system induce upon each other. The algorithm hierarchically partitions the plane around the bodies into successively smaller cells. Each cell forms an internal node of the quadtree and summarizes information about the bodies it contains. The leaves of the quadtree are the individual bodies. This hierarchy reduces the time to calculate the force on the bodies because, for cells that are sufficiently far away, it suffices to perform only one force calculation with the cell instead of performing one calculation with each body inside the cell.

The algorithm has three main phases that run in sequence for each time step, until the desired ending time is reached. These phases are: (1) creating the quadtree, (2) computing the forces acting on each body, (3) updating the state of the system. The main computation load is in phase 2, whose computations can be done in parallel. Although phase 3 can be parallelized too, this is usually not considered because it is very lightweight.

[1]TBB is available on the FreeBSD, Linux, Solaris, Mac OS X, and Microsoft Windows operating systems and run on top of the x86/x64 (both from Intel and AMD), IA64 (Itanium family) and MIC (new Intel Xeon Phi) processors. There are also ports for the Xbox 360 and PowerPC-based systems. This way, TBB supports the vast majority of current computers.

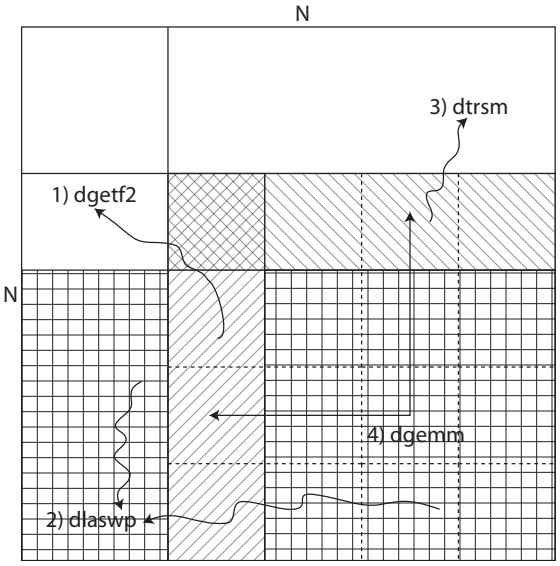

Figure 6: Blocked algorithm for computing the LU decomposition of a square matrix

```
1  for (j = 1; j < num_blocks; j ++) {
2    dgetf2(A(j:num_blocks, j), P);
3
4    update_pivots(P);
5    dlaswp(A(1:num_blocks, 1:j-1), P);
6    dlaswp(A(1:num_blocks,
7             j+1:num_blocks), P);
8
9    for_range(i in j+1:num_blocks) {
10     spawn(dtrsm, A(j, j), A(j, i));
11   }
12
13   for_range(i1 in j+1:num_blocks,
14             i2 in j+1:num_blocks) {
15    spawn(dgemm, A(i1, j),
16                 A(j, i2),
17                 A(i1, i2));
18   }
19   wait_for_all();
20 }
```

Figure 7: Pseudocode of the LU decomposition

The parallelization of this algorithm with `spawn` is quite simple: it is only needed to distribute the bodies in a block fashion and spawn the computation methods, as shown in the pseudocode in Fig. 5. It must be noted, however, that load balancing, such as the one provided by our library, may play an important role on the performance of the parallel implementations of this algorithm. The reason is that the traversal of the quadtree performed in stage (2) of the algorithm does not have the same cost for each body. Namely, the traversal is deeper and requires more computations for the bodies located in more densely populated regions. For this reason, our implementations overcompose the parallel loop in more tasks than available cores and let the underlying framework perform load balacing between the available cores. Notice also how the other stages are automatically synchronized thanks to our library, i.e., they only run when they are free of conflicts with preceding tasks.

3.2. LU decomposition

LU decomposition factorizes a matrix as the product of a lower triangular matrix and an upper triangular matrix. It is a key part of several numerical algorithms as the resolution of linear equations or computing the determinant of a matrix. The LAPACK [15] library contains a blocked implementation of this algorithm, represented in Fig. 6 with its pseudocode in Fig. 7, in which `A(i,j)` refers to the block in row `i` and column `j` in which the input matrix `A` has been divided. The algorithm progresses through the main diagonal of the matrix, and in each step it performs four operations: (1) the LU decomposition of the diagonal block is computed using the unblocked version of the algorithm (`dgetf2`) (line 2), (2) it swaps the rows of the matrix from the diagonal to the end, according to the pivots retuned by the previous step (line 4), (3) it computes the solution of $A \times X = B$, being A the block on the diagonal and B the rest of the row to the end of the matrix (line 9), and (4) it multiplies the row and column blocks to obtain the next square submatrix (line 13).

The parallelization strategy for this algorithm consists in subdividing the most expensive operations in blocks, in this case `dtrsm` but mainly `dgemm`. These blocks are represented by the dashed lines in Fig. 6. Each block is assigned to a different task to perform the required operation. The `for_range` construction in lines 9 and 13 represents a template function provided by our library whose purpose is to automatically divide an input range of blocks in smaller subblocks with the optimal size, so neither too many nor too few spawns are called.

```
1  for (j = 1; j < num_blocks; j ++) {
2    dsyrk(A(1:num_blocks,j), A(j, j));
3    dpotf2(A(j, j));
4
5    for_range(i in j:num_blocks) {
6      spawn(dgemm, A(1:j-1, j)^t, A(1:j-1, i), A(j, i));
7    }
8
9    for_range(i in j+1:num_blocks) {
10     spawn(dtrsm, A(j, j), A(j, i));
11   }
12   wait_for_all();
13 }
```

Figure 8: Pseudocode of the Cholesky decomposition

3.3. Cholesky decomposition

Cholesky decomposition takes a matrix and computes its factorization as a lower triangular matrix and its conjugate transpose. The blocked version of the algorithm follows a structure similar to LU. First, the diagonal block is computed with the non-blocked version of the algorithm. Then, the remaining matrix is computed multiplying the resulting blocks. Thus, the parallelization pattern is similar to the one used for LU, spawning tasks for the different sub-blocks of these operations.

The pseudocode of this algorithm is shown in Fig. 8, in which again `A(i,j)` refers to a block. The algorithm has two basic steps: in lines 2 and 3, the block of the diagonal is processed with an unblocked version of Cholesky, and from line 5 to the end the remaining matrix is computed and prepared for the next iteration.

3.4. Sylvester equations resolution

The Sylvester equation [16], commonly found in control theory, is the equation of the form $AX+XB = C$ where A, B, C, X are $n \times n$ matrices, being X the unknown. We use $X = \Omega(A, B, C)$ to represent the solution to the equation. In particular, we focus on the triangular case, where both A and B are upper triangular matrices. The solution of the triangular case arises as an intermediate subproblem in the Sylvester equation solver described in [16]. FLAME derives a family of blocked algorithms [17]. The result of X is stored in C. The algorithm is a hybrid of iterative and recursive algorithms. In this case, each of the blocks can be assigned to a different task, and the dependency detection system will take care of the order of execution to provide the correct result.

The computation of the solution for the Sylvester equation is done multiplying and recursively solving the blocks of the matrix. Fig. 9 shows this process: the algorithm divides the matrices in 9 blocks, which change sizes as the algorithm progresses; then, for each block, the required operation is invoked, `syl` for recursively solving or `mul` to multiply two blocks. This algorithm has a complex dependency graph, shown in Fig. 10, and it is the least scalable of the examples we tested.

4. Evaluation

In order to evaluate our approach we parallelized the algorithms described in the preceding Section using both our library and OpenMP [9], a standard high-level tool for the development of parallel applications in multicore systems. The performance tests were performed in a PC with an Intel i7 950 processor (with 4 cores and Hyperthreading) and 6GB of RAM, as well as in a server with 2 Intel Xeon E5620 quad-core processors and 16 GB of RAM. The compiler used was g++ v. 4.6.3 using -O3 optimization level.

For the N-body simulation we used a system of 100 000 bodies and simulated 1000 iterations. The speedups obtained in these experiments are shown in Figs. 11 and 12. Figures 13 to 18 show the performance of the considered linear algebra algorithms, using different combinations of hardware, libraries, and parallelization methods. Our library is compared with OpenMP and a purely sequential optimized version of the algorithms, using both the standard BLAS implementation [18] and the GotoBLAS2 library [19].

```
1  while(size(A22) > 0) {
2    divide matrix;
3
4    spawn(syl, A11, B00, C10);   // 1
5    spawn(syl, A22, B11, C21);   // 2
6    spawn(mul, C11, A12, C21);   // 3
7    spawn(mul, C11, C10, B01);   // 4
8    spawn(syl, A11, B11, C11);   // 5
9    spawn(mul, C00, A01, C10);   // 6
10   spawn(mul, C01, A01, C11);   // 7
11   spawn(mul, C01, A02, C21);   // 8
12   spawn(mul, C22, C21, B12);   // 9
13   spawn(mul, C12, C10, B02);   // 10
14   spawn(mul, C12, C11, B12);   // 11
15 }
```

Figure 9: Pseudocode of the Sylvester equations solver

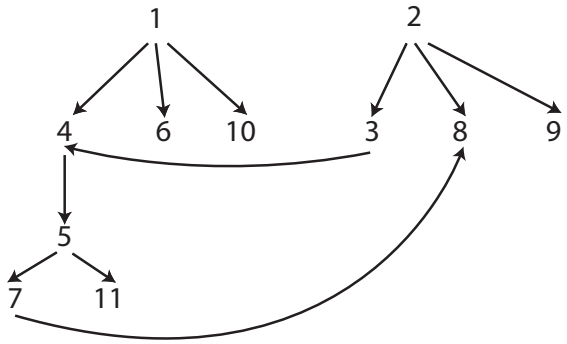

Figure 10: Dependencies of the Sylvester equation solver. Nodes refer to calls in Fig. 9

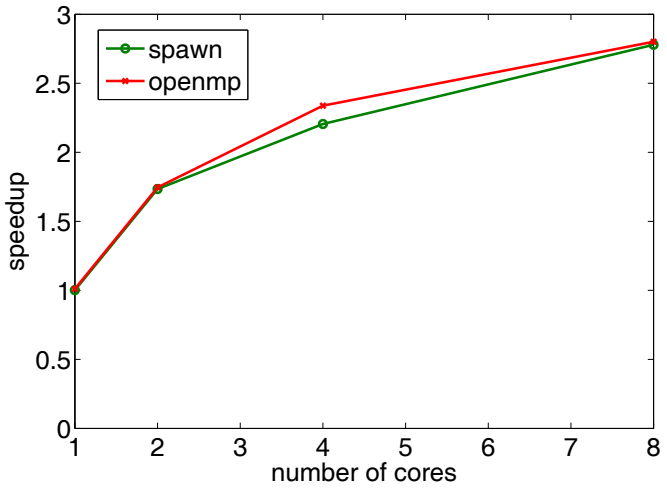

Figure 11: N-body simulation with Barnes-Hut algorithm in the i7 system

The rank of the double-precision matrices used in these tests is 8192. The i7 results for 8 cores actually correspond to the usage 4 cores with 2 threads per core thanks to the hyper-threading. It is well-known that the second thread per core provided by hyper-threading typically only provides between 5% and 20% of the performance of a real core. In fact this additional thread decreases performance for many applications due to the conflicts between the threads working sets in the core caches when large data sets are manipulated. This is the reason for the reduced performance for LU and Cholesky using 8 cores in the i7. As a matter of fact, the lack of optimizations in the usage of the memory hierarchy is a critical reason behind the poor performance, as well as the lack of scalability in the i7, of the implementations based on the standard BLAS distribution. In the case of Sylvester, given the large number of stages of the algorithm that have no parallelism and the limited maximum number of parallel tasks available (see Fig. 10), the small scalability was to be expected for any implementation.

The `spawn`-based version usually matches or outperforms the OpenMP version. We find two reasons for this. First, our library allows a task to run exactly as soon as its dependences have been satisfied. In OpenMP it is impossible to specify with such a fine grain the dependencies between tasks in different loops. Rather, synchronizations such as global barriers are required, which results in threads being idle more often. The other reason is that the task creation, scheduling and load balancing mechanisms provided by the OpenMP implementation can be less optimized and sophisticated than the ones that TBB provides to our library [20].

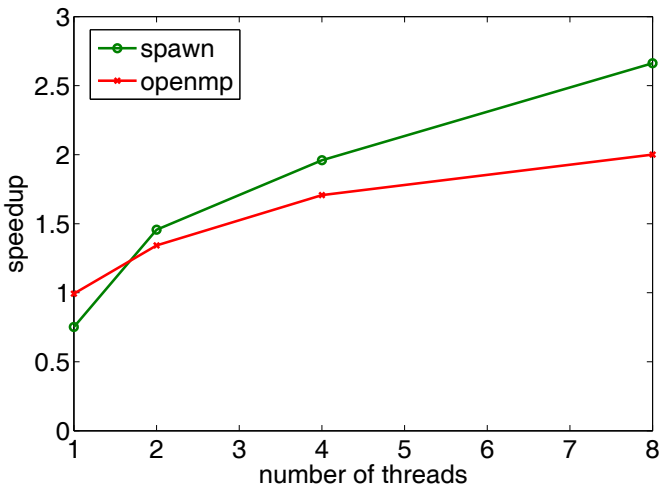

Figure 12: N-body simulation with Barnes-Hut algorithm in the Xeon system

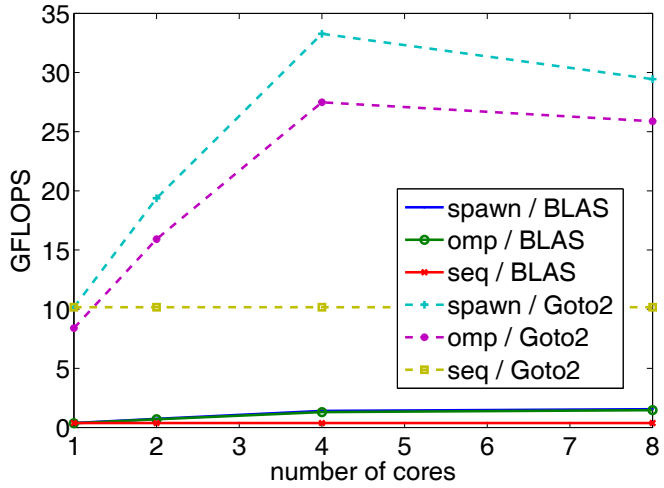

Figure 13: LU decomposition in the i7 system

We have also performed a comparison in terms of programmability of the codes parallelized with our library and OpenMP using three metrics whose results are shown in Table 2. The first metric is the number of Source Lines of Code, which counts all the lines in the program excluding the comments and the empty lines. The second one is the programming effort [21]. This metric considers a program as a string of tokens, which can be divided into two groups: *operators* and *operands*. The *operands* are variables or constants. The *operators* are symbols or combinations of symbols that affect the value or ordering of operands. We denote η_1, the number of unique operators; η_2, the number of unique operands; N_1, the total occurrences of operators; and N_2, the total occurrences of operands. The program level *Lvl*, which measures the level of abstraction at which the program is coded, and the program volume *Vol*, which represents the number of bits required to encode the program in an alphabet with one character per operand or operator, are values defined as functions of η_1, η_2, N_1 and N_2 (see [21]). The programming effort E can be obtained using the estimate of *Lvl* and *Vol* as $E = Vol/Lvl$. It is assumed that the programming effort required to implement an algorithm is proportional to E. Finally, the cyclomatic number [22] is defined as $V = P + 1$, where P

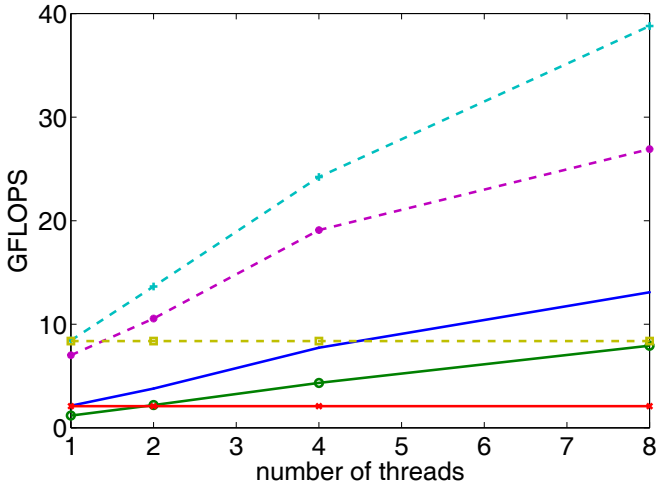

Figure 14: LU decomposition in the Xeon system

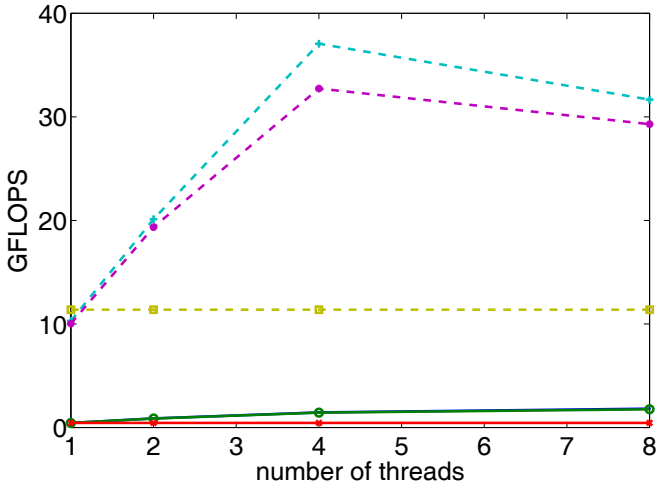

Figure 15: Cholesky decomposition in the i7 system

is the number of decision points or predicates in a program, i.e., the number of conditionals in the program flowchart. The smaller V, the smaller the complexity of the program is. In C++, P amounts to the number of constructions that include a predicate on which the control flow depends. These are the `if`, `for`, `while`, and `case` keywords, the `?:` operator, and the conditional preprocessor macros.

Usually, our library achieves better results than OpenMP for any programmability metric. An important reason is that our `Array` class greatly simplifies the programming of array-based algorithms compared with the manual management of pointers and dimensions required by the standard approach. And this is the case even when we have counted the functions that unpack the pointers and matrices dimensions from the `Array`s to make the proper calls to the underlying BLAS implementations as part of the programming effort in Table 2. Since these are very typical matrix operations, they could well be included as part of our library, therefore further strongly improving all these metics. The programmability also benefits from the notation required by our library, which is way terser than OpenMP. With `spawn`, one simply adds this word in the line of a procedure invocation that should be run as a parallel task, while OpenMP often requires separate

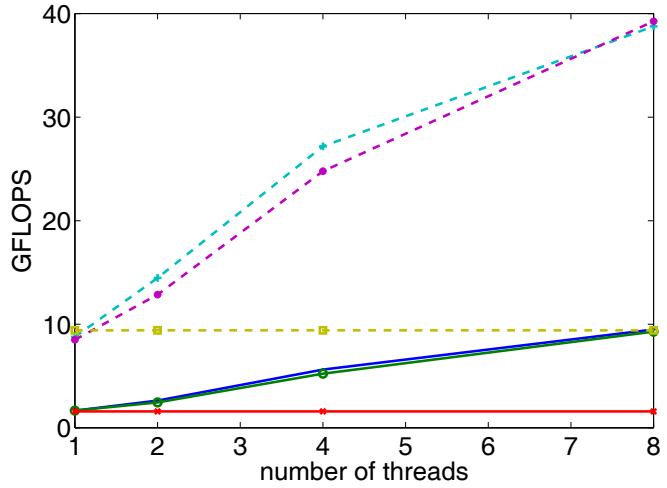

Figure 16: Cholesky decomposition in the Xeon system

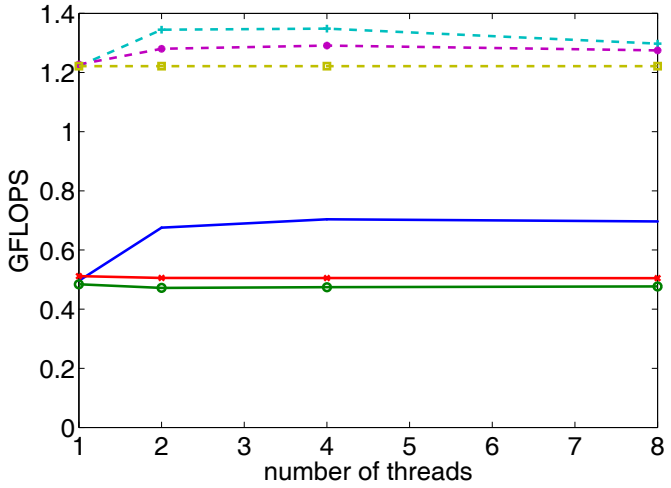

Figure 17: Sylvester equations in the i7 system

multiword directives for the creation of parallel regions and individual tasks. In Barnes-Hut, however, our approach yields somewhat worse programmability statistics. The reason is that in this application the computational functions subject to parallel invocations were not generic functions, but class methods, and our current implementation of `spawn` does not support them in a straightforward way. This is simply solved adding functions that receive the object and the method arguments and which perform the method invocation, so that `spawn` is applied to these auxiliary functions.

5. Related work

The need for explicit synchronizations is proportional to the flexibility in the patterns of computation and parallelism supported by a programming paradigm or tool. Functional programming, by avoiding state and mutable data, can provide referential transparency, so that the result of a function evaluation only depends on its arguments. This property allows in principle to evaluate in parallel any subexpression,

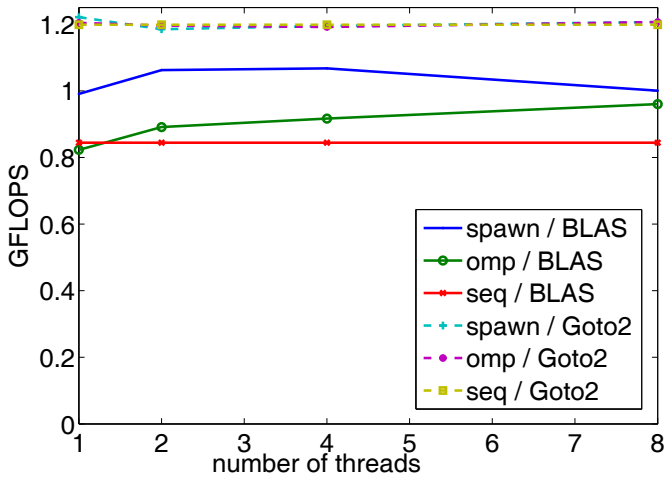

Figure 18: Sylvester equations in the Xeon system

Table 2: Programmability metrics

Application	version	SLOCs	Programming Effort	Cyclomatic #
Barnes-hut	spawn	214	919 352	36
	OpenMP	210	817 543	36
LU	spawn	94	349 770	12
	OpenMP	95	685 945	17
Cholesky	spawn	115	620 087	12
	OpenMP	116	796 005	16
Sylvester	spawn	98	789 667	3
	OpenMP	101	849 096	6

as any order of evaluation yields the same result. Nevertheless, the exhaustive exploitation of all this implicit parallelism would involve much overhead. For this reason, in practice functional languages provide mechanisms to explictly label those computations whose parallel evaluation can be advantageous [23]. While some approaches [24, 25, 26] lead to explicit communications and synchronizations, in others the user just identifies the parallel tasks, letting the compiler or runtime take care of the low level details.

Data-parallelism, which applies a single stream of instructions in parallel to the elements of one or several data structures, is the basis of some of the implicitly synchronized functional proposals [27, 28]. Unfortunately, this strategy is too restrictive for many applications either semantically or in terms of performance. A greater degree of flexibility is provided by parallel skeletal operations [29, 30, 31, 32, 33, 34], which specify the dependencies, synchronizations and communications between parallel tasks that follow a pattern observed in many algorithms. Their applicability is restricted thus to computations that fit the predefined patterns they represent.

Interestingly, there are also proposals [31, 35, 36, 32] that, like DepSpawn, allow to express tasks with arbitrary patterns of dependencies while avoiding explicit synchronizations. The fact that they are implemented in purely functional languages with lazy evaluation makes their programming strategy and the difficulties faced by the programmer very different from those of DepSpawn, which is integrated in an imperative language with the usual semantics. This way, their users have to deal with laziness, which hinders effective parallelization, and they have often to enforce sequential ordering, which are problems inexistent in our environment. Also, their functions cannot modify their inputs; rather they just return a result, whose usage expresses the dependency on the function, and which cannot be overwritten by other functions due

```
void g(A& a, A& b, A& c, A& d) {
   ...
}

void f(const A& x, A& y) {
   ...
}

spawn(g, a, b, c, d);
spawn(f, a, b);
spawn(f, c, d);
spawn(f, b, d);
spawn(f, c, a);
wait_for_all();
```

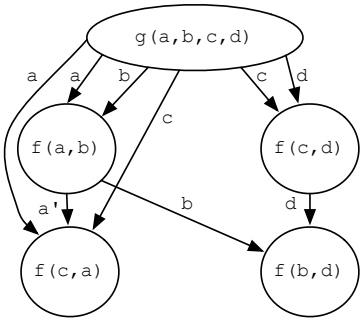

Figure 19: Small DepSpawn code and associated dataflow graph.

to the immutability of data. This implies that once a function finishes, all the subsequent tasks that use its result can run in parallel, as they can only read it, there being no need to track potential ulterior modifications of its value. On the contrary, DepSpawn functions express their dependencies only through their arguments, which can be inputs, outputs or both, while the return value is either inexistent or disregarded. Also, since the function arguments can be both read and/or written by arbitrary functions, the result(s) of a task can be overwritten by other tasks, leading to more complex patterns of dependencies on each data item than in a purely functional language. Finally, those approaches do not provide explicit synchronization facilities or classes similar to `Array` within their implicitly synchronized frameworks.

Dataflow programming [37, 38] is another paradigm with links to our proposal. The reason is that it models programs as directed graphs whose nodes perform the computations and whose edges carry the inputs and outputs of those computations, thus interconnecting data-dependent computational nodes. This view promotes the parallel execution of independent computations, which are only run when their data dependencies are fulfilled, very much like DepSpawn tasks. Under this paradigm a DepSpawn task can be viewed as a node with input edges for its input arguments, and output edges for its output arguments. The node would only be triggered when there were data in all its input edges, and it would generate a result in each one of its output edges. Arguments that can be both inputs and outputs would be represented with edges both entering and leaving the node. If several tasks had a dependency on a given output of a task, the node of this task would be connected to each one of the dependent nodes with a separate output edge labeled with the output argument name. If the communication between DepSpawn tasks took place by means of individual copies, mimicking the behavior of edges that carry the data, this is all that would be needed. However, communication actually takes place through shared modifiable variables. This implies that if we want to represent a DepSpawn program with a dataflow graph whose tasks have the same set of legal schedules, we must prevent tasks that can write to a variable from beginning their execution while any preceding reader has not finished, even when there is no actual flow of data between them. This is achieved by connecting tasks that only read a given argument with output edges associated to it that link them with the next task that can modify that argument. This way the writer cannot be triggered until the reader finishes and puts a value in the connecting edge, even if the value read from the edge is discarded. Figure 19 illustrates all the situations described above with a small piece of code and its corresponding dataflow graph. The edge `a'` has been marked with an apostrophe to indicate that there is no actual flow of data, but a signal to prevent `f(c,a)` from modifying `a` while `f(a,b)` is still working on it. The figure also illustrates why a node must be used per invocation/task rather than by function, as, for example, otherwise it would be impossible to run parallel instances of a function.

While all of them have in common that data in the input edges of a node trigger its computation, which in turn generates new data in its output edges, there are several models of dataflow networks with different assumptions and semantics [39]. A DepSpawn application in which each task is deterministic and tasks only communicate through their arguments can be modeled as a Kahn Process Network (KPN) [40], which is a network composed by deterministic sequential processes that communicate through one-way unbounded FIFO channels. In fact, since data are always consumed and produced by means of a single argument, which

is of an aggregate type such as `Array` when several data items are involved, a network of tasks generated by DepSpawn can be more accurately modeled as a Synchronous Dataflow (SDF) [41], a restriction of KPN in which nodes consume and produce a fixed number of data items per firing in each one of their edges. Furthermore, DepSpawn dataflow graphs are homogeneous SDFs, as all nodes produce or consume a single sample on each input or output arc when invoked. The properties of the associated computation graphs have been analyzed in [42, 43, 44].

Notice that the aforementioned characterization is also valid when tasks spawn children tasks, provided that these children are also deterministic and only communicate with other tasks through their arguments. This is thanks to the fact that DepSpawn tasks wait for all the preceding tasks with which they have dependencies and their descendants.

Programs developed under the usual imperative languages can also avoid explicit synchronizations by resorting to data-parallel approaches [45, 1, 3, 46] and parallel skeletons [47, 48, 8, 49, 33], at the cost again of restricting the patterns of parallelization. Because of their limitations, users are often forced to resort to lower level approaches [50, 51, 52, 7, 8, 53, 54, 9] seeking more flexibility. The downside is that these tools hurt productivity, being explicit synchronizations one of the reasons. Some of them [51, 8, 53, 54, 9] have in common with DepSpawn that they allow to build tasks expressed as functions, although often with a more convoluted syntax, and that they let the runtime manage the execution of these tasks using advanced techniques such as work-stealing. In fact, as explained in Sect. 2.4, DepSpawn in built on top of TBBs [8]. However, none of them can automatically track and enforce the dependencies among the tasks. This way, they all require explicitly synchronizing the tasks by linking each task with its successors or predecessors, establishing synchronization points, or accessing futures attached to variables that represent the tasks. Achieving a behavior similar to DepSpawn, in terms of performance, minimal interface and automatic fulfillment of data dependencies by means of these tools is a non trivial task.

The idea of exploring arbitrary out-of-order execution of tasks by relying on the dependences among them has already been explored in the context of libraries and compilers. This way, Supermatrix [4, 5, 6] provides a library exclusively focused on linear algebra algorithms that is able to execute its parallel tasks out-of-order respecting the serial semantics by means of a task dependency analysis. Since, contrary to our library, its aim is not general, it does not define any particular programming model either.

As for compiler-based approaches, the Star Superscalar (StarSs) programming model, with implementations for different devices such as the Cell broadband engine [10] or general SMP computers [11], similarly to our library, seeks to provide general out-of-order execution of tasks based on data dependencies. Nevertheless, this paradigm, which has led to the proposal of extensions to OpenMP [12][13], requires the user to explicitly annotate such dependencies by means of compiler directives in the code. This involves not only analyzing the code to establish which are the inputs and outputs of each task, but also which is the exact size of the area of the arrays pointed by the pointers used in these tasks, as well as the extension of the region accessed by each pointer. Our library provides elegant solutions to these problems by directly extracting the information from the function parameters or from the `Array` objects provided, resulting in clearer and less error-prone codes.

There are also important differences in the programming model of both approaches. Namely, in the programming model supported by these compiler directives, dependencies are only detected inside the scope of the same parent task. This way if there is any piece of data on which there can be carried dependencies that need to be considered by a task, it must be explicitly annotated in its parent. Obviously, this also implies in turn annotating all the ascendants up to the level where the potential dependency is generated. And these annotations also imply that those ascendants will have to abide by those dependencies, even when they are actually only needed for the bottom-level task we were initially considering. Nevertheless, under the programming model provided by our library, tasks automatically fulfill any dependencies generated not only in their parent task, but also in all of their ancestors, there being no need to apply those dependencies to any of those ancestors if they do not need them. This significatively increases the amount of parallelism that can be exploited in many situations. Other distinctive features of our library that improve this aspect are the possibility of releasing (some) dependencies before a task finishes, or blocking a task at some arbitrary point in order to wait for a specific set of variables.

6. Conclusions

In this work we have presented DepSpawn, a new library for parallel programming that provides very flexible patterns of parallelism without the need of explicit synchronizations. Using advanced features of C++11, our library is able to analyze the parameters of arbitrary functions and detect dependencies between them. With this information it schedules their parallel execution while respecting their dependencies. We have also provided a clear description of the programming model enabled by our library, as well as a comparison with a standard high-level approach to parallelize applications in multicore systems like OpenMP. The results obtained are very satisfactory, both in terms of performance and programmability.

As future work we plan to extend the support for functions that return values using the concept of futures. Namely, for these functions `spawn` would return a special object to hold the value returned by the function. Reading this object would conform an implicit synchronization point. DepSpawn is publicly available under an open-source license at http://depspawn.des.udc.es.

Acknowledgements

This work was supported by the Xunta de Galicia under project INCITE08PXIB105161PR and program 2010-06, by the Ministry of Science and Innovation, cofunded by the FEDER funds of the European Union, under the grant TIN2010-16735, and by the FPU Program of the Ministry of Education of Spain (Reference AP2009-4752). Finally, we thank the anonymous reviewers for their suggestions, which helped improve the paper.

References

[1] S. Hiranandani, K. Kennedy, C.-W. Tseng, Compiling Fortran D for MIMD distributed-memory machines, Commun. ACM 35 (1992) 66–80.

[2] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, L. Snyder, W. D. Weathersby, C. Lin, The case for high-level parallel programming in ZPL, IEEE Comput. Sci. Eng. 5 (3) (1998) 76–86.

[3] B. Fraguela, G. Bikshandi, J. Guo, M. Garzarán, D. Padua, C. von Praun, Optimization techniques for efficient HTA programs, Parallel Computing 38 (9) (2012) 465–484. doi:10.1016/j.parco.2012.05.002.

[4] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn, Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures, in: Proc. 19th ACM symp. on Parallel algorithms and architectures, SPAA'07, 2007, pp. 116–125.

[5] E. Chan, F. G. Van Zee, E. S. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn, Satisfying your dependencies with Supermatrix, in: Proc. 2007 IEEE Intl. Conf. on Cluster Computing, CLUSTER'07, 2007, pp. 91–99.

[6] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, R. van de Geijn, Supermatrix: a multi-threaded runtime scheduling system for algorithms-by-blocks, in: Proc. 13th ACM SIGPLAN Symp. on Principles and practice of parallel programming, PPoPP'08, 2008, pp. 123–132.

[7] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, K. Warren, Introduction to UPC and language specification, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences (1999).

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

[9] OpenMP Architecture Review Board, OpenMP Program Interface Version 3.1, http://www.openmp.org (July 2011).

[10] J. Planas, R. M. Badia, E. Ayguadé, J. Labarta, Hierarchical task-based programming with StarSs, Int. J. High Perform. Comput. Appl. 23 (3) (2009) 284–299.

[11] J. Perez, R. Badia, J. Labarta, A dependency-aware task-based programming environment for multi-core architectures, in: 2008 IEEE intl. conf. on Cluster Computing, 2008, pp. 142–151.

[12] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, J. Labarta, A proposal to extend the OpenMP tasking model with dependent tasks, Intl. J. Parallel Program. 37 (3) (2009) 292–305.

[13] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. González, F. D. Igual, D. Jiménez-González, J. Labarta, Extending OpenMP to survive the heterogeneous multi-core era, Intl. J. Parallel Program. 38 (5-6) (2010) 440–459.

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

[15] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, D. Sorensen, LAPACK: a portable linear algebra library for high-performance computers, in: Proc. 1990 ACM/IEEE conf. on Supercomputing, Supercomputing '90, 1990, pp. 2–11.

[16] R. H. Bartels, G. W. Stewart, Solution of the matrix equation AX + XB = C [F4], Commun. ACM 15 (9) (1972) 820–826.

[17] J. A. Gunnels, F. G. Gustavson, G. M. Henry, R. A. van de Geijn, FLAME: Formal Linear Algebra Methods Environment, ACM Trans. Math. Softw. 27 (4) (2001) 422–455.

[18] National Science Foundation, Department of Energy, BLAS, http://www.netlib.org/blas/ (2011).
[19] K. Goto, GotoBLAS2, `http://www.tacc.utexas.edu/tacc-projects/gotoblas2`, accessed: 16 Feb 2013.
[20] S. Olivier, J. Prins, Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs, Intl. J. Parallel Program. 38 (5-6) (2010) 341–360.
[21] M. H. Halstead, Elements of Software Science, Elsevier, 1977.
[22] McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (1976) 308–320.
[23] P. Totoo, H.-W. Loidl, Parallel Haskell implementations of the N-body problem, Conc. and Comp.: Practice and Experience. To appear.
[24] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2007.
[25] P. Haller, M. Odersky, Scala actors: Unifying thread-based and event-based programming, Theor. Comput. Sci. 410 (2-3) (2009) 202–220.
[26] S. Marlow, R. Newton, S. Peyton Jones, A monad for deterministic parallelism, SIGPLAN Not. 46 (12) (2011) 71–82.
[27] S. Peyton Jones, Harnessing the multicores: Nested data parallelism in Haskell, in: Proc. 6th Asian Symp. on Programming Languages and Systems, APLAS'08, 2008, pp. 138–138.
[28] A. Prokopec, P. Bagwell, T. Rompf, R. Odersky, A generic parallel collection framework, in: Proc. 17th intl. conf. on Parallel Processing, Euro-Par'11, 2011, pp. 136–147.
[29] M. Cole, Algorithmic skeletons: structured management of parallel computation, MIT Press, 1991.
[30] F. A. Rabhi, Abstract machine models for highly parallel computers, Oxford University Press, 1995, Ch. Exploiting parallelism in functional languages: a "paradigm-oriented" approach, pp. 118–139.
[31] P. W. Trinder, K. Hammond, H.-W. Loidl, S. L. Peyton Jones, Algorithm + strategy = parallelism, J. Funct. Program. 8 (1) (1998) 23–60.
[32] R. Loogen, Y. Ortega-mallén, R. Peña marí, Parallel functional programming in Eden, J. Funct. Program. 15 (3) (2005) 431–475.
[33] H. González-Vélez, M. Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers, Softw., Pract. Exper. 40 (12) (2010) 1135–1160.
[34] C. Smith, Programming F# 3.0, O'Reilly Media, 2012.
[35] S. Marlow, S. Peyton Jones, S. Singh, Runtime support for multicore Haskell, SIGPLAN Not. 44 (9) (2009) 65–78.
[36] S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, P. Trinder, Seq no more: better strategies for parallel Haskell, SIGPLAN Not. 45 (11) (2010) 91–102.
[37] J. B. Dennis, First version of a data flow procedure language, in: B. Robinet (Ed.), Programming Symposium, Vol. 19 of Lecture Notes in Computer Science, Springer-Verlag, 1974, pp. 362–376.
[38] J. B. Dennis, Data flow supercomputers, Computer 13 (11) (1980) 48–56.
[39] E. Lee, T. Parks, Dataflow process networks, Proceedings of the IEEE 83 (5) (1995) 773 –801.
[40] G. Kahn, The semantics of a simple language for parallel programming, in: IFIP Congress, 1974, pp. 471–475.
[41] E. Lee, D. Messerschmitt, Synchronous data flow, Proceedings of the IEEE 75 (9) (1987) 1235–1245.
[42] R. M. Karp, R. E. Miller, Properties of a model for parallel computations: Determinacy, termination, queueing, SIAM J. Appl. Math. 14 (6) (1966) 1390–1411.
[43] R. Reiter, Scheduling parallel computations, J. ACM 15 (4) (1968) 590–599.
[44] F. Commoner, A. W. Holt, S. Even, A. Pnueli, Marked directed graphs, J. Comput. Syst. Sci. 5 (5) (1971) 511–523.
[45] L. Snyder, The design and development of ZPL, in: Proc. 3rd ACM SIGPLAN conf. on History of programming languages, HOPL III, 2007, pp. 8–1–8–37.
[46] A. de Vega, D. Andrade, B. B. Fraguela, An efficient parallel set container for multicore architectures, in: intl. conf. on Parallel Computing, ParCo 2011, 2011, pp. 369–376.
[47] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, Parallel Computing 30 (2004) 389–406.
[48] J. Falcou, J. Sérot, T. Chateau, J.-T. Lapresté, Quaff: efficient C++ design for parallel skeletons, Parallel Computing 32 (7-8) (2006) 604–615.
[49] C. H. González, B. B. Fraguela, An algorithm template for domain-based parallel irregular algorithms, International Journal of Parallel Programming, in press. doi:10.1007/s10766-013-0268-3.
[50] D. R. Butenhof, Programming with POSIX Threads, Addison Wesley, 1997.
[51] K. H. Randall, Cilk: Efficient multithreaded computing (1998).
[52] R. W. Numrich, J. Reid, Co-array Fortran for parallel programming, SIGPLAN Fortran Forum 17 (2) (1998) 1–31.
[53] D. Leijen, W. Schulte, S. Burckhardt, The design of a task parallel library, SIGPLAN Not. 44 (10) (2009) 227–242.
[54] C. Campbell, R. Johnson, A. Miller, S. Toub, Parallel Programming with Microsoft .NET - Design Patterns for Decomposition and Coordination on Multicore Architectures, Microsoft Press, 2010.