A framework for argument-based task synchronization

Carlos H. González and Basilio B. Fraguela

Universidade da Coruña, Galiza
{cgonzalezv,basilio.fraguela}@udc.es

Abstract. Synchronization in parallel applications can be achieved implicitly or explicitly. For example, the semantics of data-parallel libraries and languages implicitly synchronize the parallel threads of execution after each parallel computation. Skeletal operations also define implicitly the synchronization of the tasks defined by the used. On the other hand, more flexible approaches that allow to express arbitrary task-level computations without a defined structure always request in turn that the user specifies explicitly the synchronization needed among the parallel tasks.
In this paper we present an approach that enables arbitrary patterns of parallelism without requiring explicit synchronizations. Our strategy relies on expressing the parallel tasks as functions that convey their dependences implicitly by means of their arguments. These arguments can be then analyzed when a parallel task is spawned in order to enforce its dependences. A library-based prototype that implements this approach is then evaluated.

Keywords: Parallel programming, synchronization

1 Introduction

One of the most common difficulties when programming parallel algorithms is the synchronization of the different units of execution (processes, threads, tasks...) Different parallel programming models and tools provide different synchronization techniques, which can be implicitly or explicitly invoked by the user.

The data-parallel paradigm [10, 4, 9] 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. The applications that benefit from more complex patterns have usually to resort to paradigms and tools where synchronizations are explicit [3, 13]. This way, programming complexity due to the specification of the synchronization requirements among the tasks is the cost to pay for the flexibility in the patterns of parallelism and dependences among tasks.

```
void f1(const int&, int&);
void f2(const int&);
void f3(const int&, const int&, const int&);

int a, b, c;
spawn(f1, a, b);
spawn(f2, b);
spawn(f3, a, b, c);
```

Fig. 1. Simple example of spawn invocation.

In this paper we propose a practical approach to enable the expression of arbitrary patterns of parallel computation while avoiding explicit synchronizations. This way our proposal is almost as flexible as explicit synchronization but without the complexity it brings to parallel programming. Our proposal relies on the usage of functions to express the parallel tasks, being the dependencies among the functions solely provided by their arguments. This way, a comparison of the arguments to a function whose execution has been requested with the arguments to the functions that have already been submitted to execution suffices to enforce the dependences among them. We demonstrate the feasibility and convenience of this approach by means of a C++ library which exploits the metaprogramming capabilities of this language to analyze at compile time the types of the parameters of the functions, and which uses that information to detect possible risks.

The rest of his paper is organized as follows. The next section explains the principles of our proposal. Section 3 describes two examples used to test the library, while Sect. 4 shows some preliminary performance results. In Section 5 there is a discussion on some of the related work, and Section 6 is devoted to the conclusions of this work.

2 An Argument-based Synchronization Approach

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 the parallel tasks are analyzed by the framework to detect the dependences among those tasks. The types are used to detect risks, which are caused by writeable elements, namely C++ non-const references, which allow to pass parameters by reference in C++. Knowing these types, the system stores in an internal structure the address and size of all the parameters along with whether they are writeable or not (i.e. whether they are non const references) by the function. Const references are also tracked because, while they do not allow to modify the input argument in the function, they can cause anti-dependencies if a posterior call needs to write in the object they refer to. This information is then used each time a new task is spawned, so that dependent functions wait until all their dependences are free. Given two functions spawned one after another, the second one will always wait if they share at least one parameter and one of them writes on it.

Table 1. State of the internal structures after the first spawn in Fig. 1

Position	Writeable	Dependencies
a	No	-
b	Yes	-

Table 2. State of the internal structures after the second spawn in Fig. 1

Position	Writeable	Dependencies
a	No	-
b	Yes	f2

Table 3. State of the internal structures after the third spawn in Fig. 1

Position	Writeable	Dependencies
a	No	-
b	Yes	f2, f3
c	No	

The code in Fig. 1 will be used to explain this behavior. It defines three functions with different arguments, which are spawned as parallel tasks in the last three lines. The code uses the actual syntax of our framework, which provides function `spawn` to launch to execution a parallel task. The task is specified by means of a function to evaluate, which is the first argument to `spawn`, followed by the arguments with which the function must be evaluated. When the first `spawn` is called there are no previous risks, so the table of dependencies is initialized with its two parameters, `a` and `b`, being `a` marked as non writeable and `b` as writeable (Table 1). The second `spawn` detects that the function `f2` is going to use `b` again and, since it is marked as a risk, it must wait. Thus it is added as a dependent of `b` as seen in Table 2. The third call represents a similar situation, so `f3` is added to the wait list of `b` (Table 3). When `f1` ends running, the system wakes the dependent functions. In this case neither of them writes on `b`, so they can run in parallel.

Figure 2 shows an example of a spawn done inside of another spawned function. The `main` routine spawns `g` and `h`, which causes a risk because both use `b` as a parameter and `g` writes on it, so `h` must wait for `g`. But `g` in turn spawns a new function `f` that does some process on `b` too. Our library supports this kind of flow in the following way: since `f` operates on `b`, which is also a parameter of its caller function, `g` will wait for its spawned children to end before releasing its parameters, and thus `h` will wait for the completion of both `g` and `f`. If this were not done this way, `h` could start to run just after `g` has finished its execution, even if `f` were still running or had not even began its execution, therefore altering the sequential semantics of the code.

```
void f(B& b) {       }

void g(A& a, B& b) {
    spawn(f, b);
 }

void h(const B& b) {       }

main() {
  spawn(g, a, b);
  spawn(h, b);
}
```

Fig. 2. Example of nested tasks with dependences.

This system works well for standard data types, structs and classes, which have a defined starting memory address and size. When the functions use pointers to arbitrary allocated memory, the library cannot detect overlaps between different blocks of memory (although the process is done for the pointer itself).

2.1 Array Support

In the previous sections the library 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 dependences 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 adds a big overhead and depending on the algorithm it is not always possible. Another solution, which is the one we have implemented in our framework, is to provide a data type that allows to express these arrays, to efficiently define subsections of them without copying data, and which is known to our dependences analysis framework so it can retrieve the range of elements an array of this class refers to in order to check for overlaps.

To provide this support, we use a modified version of the Array class of the Blitz++ library [15]. This C++ library implements efficent array classes and operations for numeric computations, and provides methods to define subarrays that reference a bigger matrix (i.e. they point to the same data), using the convenient syntax shown in Fig. 3. Our `Array` class, derived from the one provided by Blitz++, enables our task spawn framework to check for overlapping subarrays which reference the same block of memory.

```
// Two dimensional matrix of size 64x64
Array array(64, 64);

// Subarray from position (10,0) to position (20, 30)
Array subarray(Range(10, 20), Range(0, 30));
```

Fig. 3. Example of the definition of a subarray using Blitz++.

```
void addreduce(int& result, const Array& values) {
  for(int i = 0; i < values.size(); i++)
    result += values(i);
}

Array r(100), a(100);
for(i=0; i<100; i+=10)
  spawn(addreduce, r(i/10), a(Range(i,i+9)));

spawn(addreduce, result, r);
```

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

An example of the utility of the `Array` class is shown in the code in Fig. 4, which adds all the elements in array `a` into `result` in two stages. First, the `for` loop spawns tasks that work on blocks of ten elements of `a`, so that the range $\mathtt{a}(\mathtt{i} : \mathtt{i} + 9)$ is reduced into the temporary $\mathtt{r}(\mathtt{i}/10)$. The array regions accessed by these tasks do not overlap, so they can be run in parallel. The next `spawn` gets the total result using all the elements of `r`. In this case the library detects the overlap between `r` and the portions of this vector that are written in each one of the tasks spawned in the preceding loop, making thus sure that the last `addreduce` waits until all these tasks have finished.

```
wait_for_all()
wait_for(Type vars...)
release(Type vars...)
```

Fig. 5. Low level synchronization methods.

2.2 Explicit synchronization facilities

Our library also provides some methods to control at a lower level the synchronization process so that the programmer can make some optimizations. These methods are shown in Fig. 5. The `wait_for_all` function makes the current thread wait until all the spawned processes end. Its intended use is to serve as a barrier where the main program can wait while the spawned processes do their operations. `wait_for` provides a more fine grained synchronization, where the current thread only waits for the tasks that generate dependences on the variables specified as arguments to finish. There can be an arbitrary number of these variables and they can be of different types. Finally, `release` can be used by a spawned process 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.

2.3 Implementation details

As we have seen, our proposal for automatic synchronization relies basically on the template function `spawn` that can be used to invoke arbitrary C++ procedures which will run in parallel in different threads. The library is built using the thread facilities of the Boost C++ library [11]. Using the C++ capabilities for template metaprogramming and variadic templates of the C++0x standard proposal, our framework can analyze at compile time the types of the parameters of the procedure, thus detecting which of them are references and thus may create risks. It must be noted that only parameters passed by reference cause dependencies. When a parameter is passed by value, it is copied in a new memory position owned by the new thread. Thus, the analysis process only checks and stores information of by-reference arguments.

It must be noted that `spawn` supports C++ functions with an arbitrary number of arguments. The return value of the spawned functions is ignored, by `spawn`. As a result, these functions should typically have return type `void`, that is, they should behave as procedures. Notice that these routines can still provide output results by means of the arguments associated to their non constant reference parameters. There are no restrictions on the types of the arguments of these functions.

3 Examples

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 two algorithms to test its performance and programmability. The next subsections briefly explain these algorithms.

3.1 Merge

The merge algorithm takes two ordered lists of numbers and produces a new list with all the numbers that is still ordered. The input lists can be of different sizes. For example, given the inputs

```
5, 11, 12, 18, 20
2, 4, 7, 11, 16, 23, 28
```

the output would be

```
2, 4, 5, 7, 11, 11, 12, 16, 18, 20, 23, 28
```

This algorithm can be implemented using a divide-and-conquer approach. A simple pseudocode is shown in Fig. 6.

The parallelization of this algorithm consists simply in spawning the recursive `merge` calls until the inputs are small enough to be processed efficiently in a

```
merge(list1, list2, output) {
  if(list1 longuer than list2)
    exchange list1 and list2 values
  if(size of list1 == 0) return
  g1 = middle position of list1
  q2 = position of list1[q1] in q2
  q3 = q1 + q2
  output[q3] = list1[q1]

  // Recursive calls
  merge(list1[0:q1-1], list2[0:q2-1], output[0:q3-1])
  merge(list2[q1+1:end], list2[q2:end], output[q3+1:end])
}
```

Fig. 6. Pseudocode for the merge algorithm.

sequential manner. The lists of numbers are stored in our `Array` class so that our framework knows that merge operations on separate regions of the arrays can proceed in parallel.

```
1  void merge(const Array<int, 1>& a1, const Array<int, 1>& a2,
2                       Array<int, 1>& out) {
3    if(len1 > len2)
4      exchange(a1, a2);
5
6    if(a1.size() == 0) return;
7
8    int q1 = (a1.size()-1) / 2;
9    int q2 = find(a1(q1), a2, 0, a2.size()-1);
10   int q3 = q1 + q2;
11   out(q3) = a1(q1);
12
13   if(a2.size() > LIMIT)
14     spawn(merge, a1(Range(0, q1-1)), a2(Range(0, q2-1)),
15                                  out(Range(0, q3-1)));
16   else
17     merge(a1(Range(0, q1-1)), a2(Range(0, q2-1)),
18                                  out(Range(0, q3-1)));
19   merge(a1(Range(q1+1, a1.size() - 1)),
20              a2(Range(q2, a2.size() - 1)),
21              out(q3+1, a1.size()+a2.size()-1)));
22 }
```

Fig. 7. Implementation of the merge algorithm using `spawn`

Figure 7 shows the implementation of a parallel version of the merge algorithm using our library. Line 4 exchanges the two arrays so they conform the condition of the second array being longer than the first. Line 6 is the base case of the recursion: when the first array has no elements the merging process is finished. Lines 8 to 11 insert the appropiate elements in their corresponding position of the output array. The `spawn` function is called in Line 14 if the arrays are not small enough. If they are the process continues sequentialy, thus not creating new tasks.

3.2 N-body simulation using Barnes-Hut

Barnes-Hut's force-calculation algorithm employs a hierarchical data structure, called an quadtree, to approximately compute the force (e.g., gravitational, electric, or magnetic force) that the n bodies in the system induce upon each other.

With n bodies, $O(n^2)$ interactions need to be considered. The Barnes-Hut algorithm hierarchically partitions the plane around the n bodies into successively smaller cells. Each cell forms an internal node of the quadtree and summarizes information about the bodies it contains, in particular their combined mass and center of gravity. The leaves of the quadtree are the individual bodies. This hierarchy reduces the time to calculate the force on the n bodies to $O(n \log n)$ 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.

```
barnes-hut {
  Bodies[N];
  while final time not reached {
    create quad-tree
    for each block from Bodies:
      spawn(compute_forces, block)
    spawn(update, Bodies)
}
```

Fig. 8. Pseudocode of the parallel implemenation using `spawn` of the Barnes-Hut algorithm

The algorithm has three main phases that run one after each other 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, where complex computations are performed to calculate the forces. These computations can be done in parallel as they do not modify the data required for the formulas (namely, the position of the bodies). The result of these computations is the velocity of each body, which is used in phase 3 to update its position. This can be done in parallel too, although its impact in the global performance is lower, as only two sums and two multiplications are done for each body. Thus, parallelizing 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. 8.

4 Preliminary Evaluation

From our tests we conclude that the dependency detection system can be used to effectively order the execution of the different threads taking into account the risks these dependencies create and avoiding them. The performance tests were done in the development system, with an Intel i7 950 processor (with 4 cores and Hyperthreading) and 6GB of RAM, to check that the possible overheads compensated with reasonable speedups and justified the development of the library. The compiler used was g++ v. 4.5 using -O3 optimization level.

For the merge algorithm we used two input integer lists with 80000000 elements each. Our version using `spawn` achieves a speedup of 4.15x with respect to a sequential version. We must remember also that we are using the Blitz++ library on both the parallel and sequential version. Comparing to another implementation of the algorithm, which uses Intel Treading Building Blocks and the C++ standard library, our version is 1.80 times faster.

The case of the Barnes-Hut algorithm is more complex, and it achieves a worse speedup. Using 2 threads and an input size of 10000 bodies it runs 1.66 times faster than the sequential version. Although the algorithm is easily parallelizable, this speedup decreases when using more threads in our current implementation of the framework. This is due to the way threads are created in this work-in-progress version of the library. The spawn method creates a new thread in each call, and it is destroyed when it ends. Given the nature of a simulation algorithm, where threads are created and destroyed at each timestep, this can cause a big overhead. We plan to use a thread pooling system to overcome this problem.

5 Related Work

The need for explicit synchronization operations is usually proportional to the flexibility in the patterns of task parallelism supported by a programming paradigm or tool. This way, data-parallel approaches [14, 10, 9] rely on applying a single stream of instructions in parallel to the elements of one or several data structures, usually arrays, therefore giving place to sets of identical parallel tasks that operate on different data. Each one of these parallel operations must be completed before the next one begins, which synchronizes implicitly each two consecutive groups of parallel tasks with a barrier between them. This largely simplifies the development of applications under this paradigm, which has proved to be one of the most beneficial ones for parallel programmer's productivity. Unfortunately the restriction to sets of parallel identical tasks that must finish before the next set begins its execution can be very restrictive for many applications either semantically or in terms of performance.

A greater degree of flexibility is provided by parallel skeletons, which are high-level models for parallel operations that build on programming patterns. Parallel skeletal operations, usually provided by libraries [7, 5, 13, 6] specify the dependences, synchronizations and communications between parallel tasks that follow a pattern observed in many algorithms. A skeleton provides implicitly the synchronization among the tasks involved in a parallel computation, which can have a certain degree of irregularity, but still it must have a predefined structure so it can fit the predefined pattern represented by the skeleton.

As a result of these limitations, users are very often forced to resort to lower level approaches which provide the flexibility they need. The downside is that these tools hurt their productivity, being explicit synchronizations one of the reasons for this. This way, tools such as [2, 1, 12, 13, 3] support richer and more diverse task-parallel applications than the pure data-parallel and skeleton-based approaches. The tasks of these programs must be explicitly synchronized using traditional mechanisms provided by these tools such as barriers or locks that serialize the access to data when several tasks need to read or write on them and this cannot be done simultaneously.

Finally, many approaches allow to mix explicit and implicit synchronizations. For example [13] provides both parallel skeletons as well as mutexes, locks and a low level framework for the specification of dependences among tasks. The message-passing paradigm, typical of distributed memory systems, deserves special mention. In this paradigm explicit synchronization operations such as barriers usually coexist with implicit synchronizations such as those provided by blocking communications between processes. The synchronization comes from the fact that the receiving process waits until the message arrives and this arrival implies that the sender process reached the point of its code where the message was sent. Unfortunately message-passing can lead to unstructured codes and the distributed view of computation and data its usual implementations offer [8] do not favor programmability. Therefore the replacement of explicit synchronizations by blocking communications seldom represents an improvement in terms of programmability.

6 Conclusions

In this work we have presented a new library for parallel programming that provides more flexibility than implicit synchronization without the need of explicit constructions. Using the advanced features of C++0x for variadic templates and template metaprogramming, our library is able to analyze the parameters of arbitrary functions and thus detect dependencies between them, and uses this information for the synchronization. We have tested the correctness of the library and its performance, and the results justify the further development and optimization of the library.

As future work we want to improve the support for functions that return values using the concept of futures. Instead of having to write the output on a parameter passed by reference, the return value of spawn would be assigned to a special object, whose reading would conform an implicit syncronization point.

Acknowledgements

This work was supported by the Xunta de Galicia under project INCITE-08PXIB105161PR 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).

References

1. OpenMP Architecture Review Board: OpenMP Program Interface Version 3.0 (May 2008)

2. Butenhof, D.R.: Programming with POSIX Threads. Addison Wesley (1997)
3. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., Warren, K.: Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences (1999)
4. Chamberlain, B., Choi, S.E., Lewis, E., Snyder, L., Weathersby, W., Lin, C.: The case for high-level parallel programming in ZPL. Computational Science Engineering, IEEE 5(3), 76 –86 (jul-sep 1998)
5. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput. 30, 389–406 (March 2004)
6. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: efficient C++ design for parallel skeletons. Parallel Computing 32(7-8), 604–615 (2006)
7. Gonzalez, C., Fraguela, B.: A Generic Algorithm Template for Divide-and-Conquer in Multicore Systems. In: High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on. pp. 79 –88 (sept 2010)
8. Gropp, W., Lusk, E., Skjellum, A.: Using MPI (2nd ed.): portable parallel programming with the message-passing interface. MIT Press, Cambridge, MA, USA (1999)
9. Guo, J., Bikshandi, G., Fraguela, B.B., Garzaran, M.J., Padua, D.: Programming with tiles. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 111–122. PPoPP '08, ACM, New York, NY, USA (2008)
10. Hiranandani, S., Kennedy, K., Tseng, C.W.: Compiling Fortran D for MIMD distributed-memory machines. Commun. ACM 35, 66–80 (August 1992)
11. Karlsson, B.: Beyond the C++ Standard Library: An Introduction to Boost. Pearson Education (2005), http://books.google.com/books?id=lFfuYJ0OeZkC
12. Randall, K.H.: Cilk: Efficient Multithreaded Computing (1998)
13. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly (July 2007)
14. Snyder, L.: The Design and Development of ZPL. In: HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages. ACM (2007)
15. Veldhuizen, T.L.: Arrays in Blitz++. In: In Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE98). pp. 223–230. Springer-Verlag (1998)