13th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2020)
Porto, Portugal
July 9-10, 2020

A Divide-and-conquer Parallel Skeleton for Unbalanced and Deep Problems

Millán A. Martínez · Basilio B. Fraguela ·
José C. Cabaleiro

Abstract The Divide-and-conquer (D&C) pattern appears in a large number of problems and is highly suitable to exploit parallelism. This has led to much research on its easy and efficient application both in shared and distributed memory parallel systems. One of the most successful approaches explored in this area consists in expressing this pattern by means of parallel skeletons which automate and hide the complexity of the parallelization from the user while trying to provide good performance. In this paper we tackle the development of a skeleton oriented to the efficient parallel resolution of D&C problems with a high degree of unbalance among the subproblems generated and/or a deep level of recurrence. Our evaluation shows good results achieved both in terms of performance and programmability.

Keywords Algorithmic skeletons · Divide-and-conquer · Template metaprogramming · Load balancing

1 Introduction

Divide-and-conquer [1], hence denoted D&C, is a strategy widely used to solve problems whose solution can be obtained by dividing them into subproblems, separately solving those subproblems, and combining their solutions to compute the one of the original problem. In this pattern the subproblems have the same nature as the original one, thus this strategy can be recursively applied to them until base cases are found. The independence of the subproblems

Millán A. Martínez · Basilio B. Fraguela
Universidade da Coruña, CITIC, Computer Architecture Group. 15071. A Coruña. Spain
Tel: +34-881 011 219
E-mail: {millan.alvarez, basilio.fraguela}@udc.es

José C. Cabaleiro
Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Universidade de Santiago de Compostela. 15782. Santiago de Compostela, Spain
Tel: +34 881 816 421
E-mail: jc.cabaleiro@usc.es

allows exploiting parallelism in this pattern, and the fact that it has a well defined structure allows expressing it by mean of algorithmic skeletons [6], which automate the management of typical patterns of parallelism [19]. Since skeletons hide the implementation details and the difficulties inherent to parallel programming from the user, they largely simplify the development of parallel versions of these algorithms with respect to manual implementations. In fact several parallel skeletons for expressing D&C problems have been proposed in the literature, either restricted to shared memory systems [18,10,8] or supporting distributed memory environments [7,3,9,15,5,11]. In addition to the large number of problems that exhibit a D&C pattern, properly designed D&C skeletons can be used to express or implement other very common patterns such as map or reduce [12]. This wide applicability makes it extremely interesting in our opinion to develop highly optimized skeletons for this pattern.

In this paper we present a novel C++ parallel skeleton for the resolution of D&C problems in shared memory environments. Our proposal improves upon the existing solutions when considering applications with a large degree of unbalance among the subproblems generated and/or a large depth in the recursion of the algorithm. As we will see, these situations can sometimes lead the existing skeletons to either provide suboptimal performance or even be inapplicable. Our skeleton, called *parallel_stack_recursion* is an evolution of the *parallel_recursion* skeleton proposed in [10] after a complete redesign and reimplementation. Our new implementation is available at https://github.com/fraguela/dparallel_recursion together with the material published in [10] and [11].

The rest of this paper is organized as follows. The next section discusses the related work. Then, Section 3 reviews the key aspects and main problems of the D&C skeleton *parallel_recursion*. Our solution to these problems is presented in Section 4, where the new skeleton and its implementation details are explained. The evaluation of the performance and programmability of the new implementation is presented in Section 5. Section 6 is devoted to our conclusions and future work.

2 Related Work

The divide-and-conquer parallel pattern is supported by a number of skeletons in the literature. Like ours, some of them are restricted to shared memory systems, the advantage with respect to the distributed system counterparts being the reduced communication and synchronization costs as well as the easier load balancing. In this category we find the Java-based *Skandium* library [18], which follows a producer-consumer model in which the tasks are pushed and popped from a shared ready queue by the participating threads. The C++ *DAC* parallel template [8] supports three different underlying runtimes (OpenMP [4], Intel TBB [22] and FastFlow [2]) who take the responsibility of balancing the workload among the threads. These two proposals have in common that the skeleton only needs to be fed the input data and functions for identifying base cases, splitting non-base cases, combining partial results and solving base cases, which are indeed the basic components of a D&C algorithm.

Our work derives from the C++ *parallel_recursion* skeleton [10], explained in detail in Section 3, which relies on the Intel Threading Building Blocks (TBB) runtime [22] for tasking and load balancing. This skeleton allows to optionally provide a partitioner object that helps the runtime to decide when the resolution of a non base problem must be solved sequentially or in a parallel fashion. This is in contrast with the previously discussed approaches, which always apply parallelism to the resolution of every non base case. They can, though, mimic a similar behavior at a higher programming cost by identifying as base cases also those non basic problems whose parallelization is not worthy and including a sequential D&C implementation for their resolution in the function that takes care of the base cases. As we will see, even with this higher degree of control, *parallel_recursion* is not well suited to problems that exhibit a large degree of unbalance. The partitioner concept of [10] is inspired by the TBB, which offers several higher order functions, many of which support partitioners, as well as lower level mechanisms to exploit D&C parallelism, although it lacks a specific skeleton for this parallel pattern.

In the distributed memory realm we find *eSkel* [7], which provides parallel skeletons for C on top of MPI, including one for D&C. The API is somewhat low-level because of the limitations of the C language, which leads for example to the exposure of MPI details. A template library for D&C algorithms without this problem is *Muesli* [5], which is designed in C++ and built on top of MPI and OpenMP, although the latter has only been applied to data-parallel skeletons, so that its D&C skeleton is only optimized for distributed memory.

Lithium [3] is a Java library specially designed for distributed memory that provides, among others, a parallel skeleton for D&C. The implementation is based on macro data flow instead templates and extensively relies on runtime polymorphism. This is in contrast with *Quaff* [9], whose task graph must be encoded by the programmer by means of C++ type definitions which are processed at compile time to produce optimized message-passing code. These static definitions mean that tasks cannot be dynamically generated at arbitrary levels of recursion and, although the library allows skeleton nesting, it has the limitation that this nesting must be statically defined.

Finally, there are D&C skeletons specifically oriented to multi-core clusters, as they combine message-passing frameworks in order to support distributed memory with multithreading within each process. This is the case of [14], which supports load balancing by means of work-stealing. The proposal though is only evaluated with very balanced algorithms and unfortunately, contrary to ours, it is not publicly available. Furthermore, their balancing operations always involve a single task, which, as we will see, can be very inefficient. Another work in this area is *dparallel_recursion* [11], an evolution of [10] in which the shared memory portion relies on [10] and offers thus the same behavior.

It is interesting to notice that while skeletons have been traditionally directly used by programmers, their scope of application is growing thanks to very promising novel research. Namely, the development of techniques and

```
template<typename T, int N>
struct Info : Arity<N> {
  bool is_base(const T& t) const; //base case detection
  int num_children(const T& t) const; //number of subproblems of t
  T child(int i, const T& t) const; //get i−th subproblem of t
};

template<typename T, typename S>
struct Body : EmptyBody<T, S> {
  void pre(T& t); //preprocessing of t before partition
  S base(T& t); //solve base case
  S post(T& t, S ∗r); //combine children solutions
};
```

Listing 1: Class templates with pseudo-signatures for the info and body objects used by `parallel_recursion`

tools to identify computational patterns and refactor the codes containing them [16,17] not only simplifies the use of skeleton libraries by less experienced users but it can even lead to the automatic parallelization of complex codes on top of libraries of skeletal operations.

3 The *parallel_recursion* skeleton

In this section we will describe the D&C algorithm template *parallel_recursion*, including the limitations that led us to propose a new alternative in this field.

3.1 Syntax and semantics

Specifying a D&C algorithm requires providing functions to decide whether a problem is a base case or, on the contrary, it can be subdivided into subproblems, to subdivide a non-base case, to solve a base case, and to obtain to solution of a non-base problem by combining the solutions of its subproblems. The analysis performed in [10] noticed that the two first functions are mostly, and often exclusively, related to the nature of the input data structure to process, while the two latter ones more strongly relate to the computation being performed. For this reason, *parallel_recursion* relies on two separate objects to provide these two groups of elements. We now describe in turn the requirements for these objects, which are modeled by the C++ class templates `Info` and `Body` shown in Listing 1.

The object that describes the structure of the problem is called the *info* object and it must belong to a class that provides the member functions `is_base(t)`, which indicates whether a given problem `t` is a base case or not, `num_children(t)`, which gives the number of subproblems in which a non-base problem can be subdivided, and finally `child(i, t)`, which returns the `i`-th child of the non-base problem `t`. As shown in Listing 1, the class `Info` for this object must derive from a class `Arity<N>` provided by the library, where `N` is either the number of children of every non-base case of the problem, when it is fixed, or the identifier `UNKNOWN` when this value is not a constant. In the first case, `Arity<N>` automatically provides the `num_children` function member so that users do not need to implement it.

We call *body* object the one that provides the computations. Its class must provide the functions of the class template `Body` shown in Listing 1. Here, `base(t)` provides the solution for a base case `t`, while `post(t, r)` receives in the pointer `r` the array of solutions to the subproblems in which a non-base problem `t` was subdivided so that combining them, maybe with some additional information from `t`, it can compute the solution to the parent problem `t`. The object class must also support a function member `pre(t)` that allows performing computations on the problem `t` before even checking whether it is a base problem or not, as it was found to be useful for some D&C problems. The library provides a class template `EmptyBody<T,S>` that can be used as base case for the body object classes, where `T` is the type of the problems and `S` is the type of the solutions, although this it not required. The main advantage of `EmptyBody` is that it provides empty implementations of all the body functions required, so that deriving a class from it avoids writing unneeded components.

Besides the input problem and the two aforementioned objects, the skeleton accepts a fourth optional argument called the *partitioner*. Its role is to indicate when parallelism should be applied during the execution of the skeleton. The partitioner can be of three different classes. If the programmer provides a partitioner from the `simple_partitioner` class, the skeleton will parallelize the resolution of any non-base problem. This behavior is the only possible one in the other shared-memory skeletons we know of [18,8]. Creating, scheduling and synchronizing parallel tasks for every level of a D&C recursion tree may be very inefficient, particularly when the tree contains many computations of small size. For this reason, two other alternatives are supported. If the partitioner belongs to the `auto_partitioner` class, the skeleton will apply some heuristics in order to try to generate a number of parallel tasks that keeps busy the threads available while allowing for some load balancing, in case the tasks were not of an identical size. In general it is impossible for the skeleton to know in advance the size of the tasks, and even the number of children of each subproblem. Therefore a third possibility is to use a `custom_partitioner`, in which the user controls when to apply parallelism by means of a `do_parallel(t)` member function that she must provide in the *info* object. The function must return a boolean that indicates whether a problem `t` should be solved using parallelism or not.

Listing 2 illustrates the use of the skeleton in a problem consisting in adding the value `val` stored in a tree of nodes of type `tree_t` in which each node has a variable number of children whose pointers are stored in a `std::vector<tree_t*>` called `children`. The base cases are identified by the `is_base` member function of the info object, whose class is `TreeAddInfo`,

```
struct TreeAddInfo: public Arity<UNKNOWN> {
  bool is_base(const tree_t *t) const { return t->children.empty(); }

  int num_children(const tree_t *t) const { return t->children.size(); }

  tree_t *child(int i, const tree_t *t) const { return t->children[i]; }
};

struct TreeAddBody: public EmptyBody<tree_t *,int> {
  int base(tree_t * t) { return t->val; }

  int post(tree_t * t, int *r) { return std::accumulate(r, r + t->children.size(), t->val); }
};

int r = parallel_recursion<int>(root, TreeAddInfo(), TreeAddBody(), auto_partitioner());
```

Listing 2: Reduction on a tree using `parallel_recursion`

as those nodes whose vector of children is empty. These nodes just contribute their stored value `val` to the reduction, as shown in the member function `base` of the body object, whose class is `TreeAddBody`. The `num_children` member function of the info object provides the correct variable number of children of each node, each child being obtained by means of the `child` member function of the same object. Finally, the `post` member function of the body object uses the `std::accumulate` function to add the values returned by all the children of the node together with the value `val` stored in the node itself. The last line in Listing 2 illustrates the invocation of the skeleton with the root of the tree and applying an `auto_partitioner` to take the decisions on parallelization.

3.2 Implementation and limitations

The implementation of `parallel_recursion` heavily relies on templates and static parallelism, which are resolved at compile time, in order to avoid the costs associated to runtime polymorphism. As for parallelism, it uses the low level API of the Intel TBB [22] to build and synchronize the parallel tasks. This also means that the library relies on the TBB scheduler to balance the workload among the threads, which is achieved by means of work-stealing.

At the top level, the skeleton proceeds generating new parallel tasks to solve the children subproblems of each non-base problem as long as the partitioner in use decides that parallelism must be applied. However, when the partitioner decides that a given problem must be solved sequentially, the skeleton assigns the solution of that problem to a purely sequential highly optimized code that relies on the info and body objects to perform the computation, but which never checks again the possibility of creating new parallel tasks within the resolution of that problem. As a result, the task graph generated by the skeleton takes the form of a tree that grows with new tasks as long as the partitioner in use recommends doing so, and which once this is not the case, reaches leaf tasks. Each leaf task recursively solves in a sequential fashion an independent D&C problem.

The aforementioned strategy is very successful in many situations, but it presents two main limitations. First, there is the issue of load balancing. As seen in [10], the skeleton can provide good performance for some unbalanced problems by generating more tasks than threads and letting the TBB scheduler balance them. However, sometimes this does not work well because the unbalance among tasks generated at high levels of the D&C tree may be too big to keep all the threads busy, while generating parallel tasks down to the level needed to attain this balance could be very detrimental to performance. In addition, even if we wished to assert as much control as possible by means of a custom partitioner, it might not be possible to estimate whether it is interesting or not to parallelize a given problem with the information available.

The second limitation is related to the implementation of the serial computations performed by the skeleton when it decides not to parallelize a D&C problem. They follow a very efficient and simple recursive strategy that relies on stack memory for the recursive calls. Unfortunately, stack memory is much more limited than other kinds of memory, and if this recursion is very deep it can be easily exhausted, breaking the program. As in the case of the load balancing problem, this can be solved by reducing the size of the sequential tasks applying parallelism up to deeper levels in the D&C tree. However, often this does not solve the stack memory problems either, as this limitation also exists in the case of the parallel computations of this skeleton. The reason is that the frame in which a parallelized problem is considered remains in the stack until the lower level tasks it generates finish and return their results, which also implies continuous growth in the depth of the stack memory as deeper and smaller parallel tasks are generated. Furthermore, the excessive parallelization may have important additional costs due to the overheads associated to the creation, scheduling and synchronization of the tasks.

4 A new D&C algorithm template

Given the nature of the problems of the `parallel_recursion` skeleton described in Section 3.2, our first approach to solve them was to try to minimize the changes required following an incremental strategy. Namely, we designed new partitioners that allowed spawning new parallel tasks from tasks that such partitioners had decided to run sequentially at some point, something that the original skeleton did not support. Unfortunately, the results obtained were unsatisfactory, which led us to consider a complete redesign and reimplementation in a form of a new algorithm template which we call `parallel_stack_recursion`. We now discuss in detail the strategy followed by this parallel skeleton together with its interface and a very useful auto-tuning feature for its most critical parameter.

4.1 Implementation strategy

We observed that the cost of the creation and management of parallel tasks and the decision on when to build them in order to balance the workload could imply large overheads, particularly in algorithms in which the core computation was relatively lightweight. As a result we decided to build our new algorithm template so that it would have a single parallel task per thread, and to base the load balancing on the ability of such tasks to steal pending work from other tasks, which should be cheaper. This largely simplified the structure of the parallel execution, in particular avoiding the requirement of `parallel_recursion` to perform an efficient scheduling of parallel tasks, which this skeleton obtained by relying on the excellent scheduler of the Intel TBB framework. As a result, the parallelization of `parallel_stack_recursion` was just based on the C++11 facilities for multithreading, thus eliminating the dependency on Intel TBB. Since the skeleton uses a single task per thread, both words will be used interchangeably in what follows.

In order to enable the load balancing among the threads, the library could simply rely on a shared queue where all the threads could place and retrieve problems to process such as the one proposed in [18]. However that strategy implies the need for synchronizations on the queue every time a thread requests a new problem to process or tries to insert new pending problems. For this reason we designed a data structure in which each thread has its own container of pending problems, where it places the new problems it generates and from which it obtains the problems it processes. The load balancing is achieved in this structure by stealing pending work from the containers of other threads when the current thread cannot find work in its own container. Such steals of course must be performed with proper synchronization on the container of the victim thread.

The use of the proposed containers to keep the problems also solves the second issue of `parallel_recursion` related to the limitation of the stack memory, as the data of our containers are stored in the heap and there are no longer recursive calls within the skeleton. Rather, each thread works in a simple cycle in which, once a new problem is obtained, it is processed in a single step if it is a base case, while non base problems can be also subject to an optional processing, after which they are decomposed in children problems that are stored in the container to be considered later. In either case, the problem is then deallocated and the skeleton proceeds to obtain a new problem to process.

As for the problem containers, given the recursive nature of D&C algorithms, and in order to enhance locality, using stacks seemed the most natural and performant option. As a result of this selection, since each thread will be always pushing and popping problems from the top of its stack, it was clear that work stolen by another thread should be taken from the opposite part of the stack, namely its bottom. Despite this adequate design decision, if work stealing could happen at any moment in the private stack of a thread, this thread would always have to use synchronization mechanisms whenever it accessed its stack in order to make sure it suffered no conflicts with work steals. This would clearly strongly degrade the performance with respect to unsynchronized accesses. In order to avoid this problem, the work stack of each thread is divided in two dynamic sections:

– At the top we find the *local section*, which is exclusively reserved for the owner thread. Since it is only accessible by its owner, work cannot be stolen form this portion of the stack and the owner thread can therefore push and pop problems in an unsynchronized fashion there. Each thread is expected to work the vast majority of the time on this portion of the stack.
– Just below there is the *shared section*, from which other threads can steal work when they run out of it, and in particular, from the bottom of this section. As a result, and as its name implies, accesses to this region must always be synchronized.

While steals could always take place with a granularity of a single problem, there are two reasons why in general it is more beneficial to steal chunks of several problems. The first one is that the computing cost associated to a single problem may be too small, and thus stealing only one problem will often be insufficient to keep reasonably busy doing useful work the thread that initiated the process. The second one is that each steal has non negligible costs, as it involves not only examining the stacks of several threads until finding one with stealable problems in the shared section, but also locking the victim stack and transferring the stolen problems from it to the destination stack. It is therefore desirable to amortize this cost among several stolen problems. For these reasons our skeleton supports a parameter called *chunkSize* that controls the number of problems stolen in a steal process. This way, once the user defines this variable, a thread will only attempt to steal work from another thread when the shared section of its stack has at least *chunkSize* elements, and that will be the amount of problems stolen.

Our library also uses the chunk size to decide when to migrate work between the local and the shared sections of a stack. This way, when a local section has a size s of at least two chunk sizes, $(\lfloor s/chunkSize \rfloor - 1) \times chunkSize$ elements of the bottom of the local section are moved to the shared section. Conversely, if a local section becomes empty, the associated thread checks whether the shared section has elements. If this is the case, the *chunkSize* problems at the top of the shared section (notice that the size of the shared section is always a multiple of *chunkSize*), and thus just next to the just emptied local section, are moved to this latter section. If on the contrary the shared section has no data, the thread will try to steal work, namely *chunkSize* problems, from another stack. The data movements between –always consecutive– sections of the same stack are very cheap because, beyond the required synchronization, as they always affect the shared section, they do not imply any actual data movement, but rather a modification of pointers that indicate the limits of the sections on the stack.

A final issue to consider is what to do when a thread cannot find work to steal from any other thread. In this case, it spinlocks, waiting for a signal that is activated each time that any thread releases a chunk to its shared section. At that point, the thread retries the steal process. If at any time it is detected that all threads are in the spinlock waiting for work, this means that all the problems have already been processed and the execution of the algorithm finishes.

4.2 Interface

One of the aims in the design of *parallel_stack_recursion* was to minimize the changes in its interface so that it were as similar as possible to that of *parallel_recursion*. Indeed the new algorithm template can use exactly the same info objects and with the same semantics as the original *parallel_recursion* algorithm. There are some changes however in the body and partitioner objects supported, which we now explain in turn.

4.2.1 Body object

Given the implementation described in Section 4.1, a problem is destroyed as soon as its children are generated, which makes it impossible to use the `post` member function described in Section 3.1. It would have been possible to use still that interface, at the cost of more memory and CPU consumption, by storing somewhere subdivided problems until all their subproblems are processed, and by adding in the internal data structures of the library components to associate each problem with its subproblems and their solutions. However, when we analyzed the highly unbalanced problems for which the new skeleton was useful, we found that the reduction operations of their D&C algorithms were not only associative and commutative, but also that we could not find situations in which it were necessary to process together a problem with the solution of its subproblems. Therefore, although it would have been perfectly possible to use exactly the same body objects as *parallel_recursion*, for efficiency reasons we propose a new `post` function that covers all the problems we found and is in fact easier to write than the original one. Its signature is

```
void post(const S& local_result, S& global_result)
```

where `S` is the type of the results, `local_result` is a partial result such as the one obtained by processing a base problem by means of the `base` member function, and `global_result` is the global result with which the local result must be reduced. A restriction with this design is that while with the original `post` function the non base problems could contribute to the computation of the global result, this is impossible now. The reason is that since partial results are only obtained from the `base` functions applied to base problems and their reductions performed by `post` invocations, the intermediate nodes of the tree have no mechanism to contribute to the global result beyond the results of its children. While this is enough in many problems, whose results are obtained by combining only the results obtained at the leaves of the D&C recursion tree, in some algorithms the internal nodes may also have a contribution to the result. For this reason, the body objects of our skeleton support a

```
struct NewTreeAddBody: public EmptyBody<tree_t *,int, true> {
  int base(tree_t * t) { return t->val; }

  void post(int local, int& global) { global += local; }
};

int r = parallel_stack_recursion<int>(root, TreeAddInfo(), NewTreeAddBody());
```

Listing 3: Reduction on a tree using `parallel_stack_recursion`

`S non_base(const T& t)`
member function that computes a partial result from a non base node `t`. The `EmptyBody` template introduced in Section 3.1 provides a default implementation of this function that just invokes the `base` member function. Finally, since only some problems benefit from the `non_base` function, the `EmptyBody` template now supports a third optional argument which is a boolean that indicates whether this function should be used, when true, or not, when false.

Given the explanations above, the problem expressed in Listing 2 using `parallel_recursion` can be rewritten using `parallel_stack_recursion` as shown in Listing 3. The listing does not include the `TreeAddInfo` class because it is identical. As for the body class, since the internal nodes of this D&C recursion tree also contribute to the result, it derives from an instantiation of the `EmptyBody` class template whose third argument is true. The member function `non_base` is not implemented though, as its default implementation relies on the `base` member function, which suffices in this case, as both base and non base nodes contribute their `val` value to the global result. Altogether we can see that the interface is very similar and somewhat simpler than that of `parallel_recursion`.

4.2.2 Partitioner

The second change reflected in the interface pertains to the partitioner. In `parallel_recursion` this object decides when to switch from parallel computation, by generating and synchronizing new parallel tasks, to sequential computation, by entering a sequential recursive computation. In `parallel_stack_recursion` however, there is always a single task per thread that iteratively works on its container stack, sometimes stealing work from other stacks. Therefore the role of the partitioner was redefined. Namely, in this skeleton it chooses whether a given problem taken from the stack must be processed using the aforementioned strategy based on dynamic stacks described up to this point or, on the contrary, it must be solved by means of a recursive sequential computation unrelated to the stack container analogous to those offered by `parallel_recursion`. The second alternative means that all the computations are directly performed in the thread that took the problem from its stack, making impossible the stealing of portions of this D&C recursion tree by other threads. It has the advantage however that the computation can be faster because every interaction with the container stack is avoided and replaced with a direct optimized recursive execution that relies on the stack memory of the thread. This can be particularly advantageous for problems whose computations are very simple.

In `parallel_stack_recursion`, the *simple* partitioner gives place to the default behavior that relies on the container stacks for all the processing, while the *custom* partitioner allows to programmatically choose between the default behavior and the optimized sequential resolution for every problem taken from the stack. This partitioner must be used with caution, since it can cause the same problems of unbalance and excessive stack memory usage that the new skeleton intends to avoid.

Regarding the *automatic* partitioner, we must remember that the effort to develop this new skeleton derives from the impossibility to find adequate work decompositions in terms of overhead incurred and load balancing in the original skeleton for very irregular problems. As a result, it seemed of little use to support any automatic partitioner in the new skeleton, since there are no simple heuristics that allow to obtain good performance in the irregular unbalanced problems it is oriented to. This is why we can notice that Listing 3 does not use the automatic partitioner used in Listing 2.

4.3 Chunk size auto-tuning

As we have seen, our skeleton only introduces one quantitative parameter, called *chunkSize*, which controls the granularity of the steals among threads as well as the movements between the sections of a stack. This is one of the most important parameters that influences performance. Usually, a program has a set of consecutive chunk sizes that provide good performance and, as one moves away from these values, the performance begins to decrease, sometimes very quickly. The reason is that if the chunk size is too small, the threads consume too much time performing continuous steals, while if it is too large, there are fewer steals and some threads remain idle for too long. Unfortunately, there is no a universal value for this parameter that guarantees good performance for all cases, as the best value depends on many factors such as the type of D&C problem, its implementation, and even on the processor architecture where the execution is performed.

Figure 1 illustrates the comments we have just made by representing the performance of the benchmarks used in our evaluation in Section 5 when they are parallelized using different chunk sizes for `parallel_stack_recursion`. The performance is measured as the speedup achieved with respect to an optimized sequential execution when running each problem using 24 threads, i.e. one per core, in the system used in our experiments, also described in Section 5, and a simple partitioner. We can see that the performance is basically

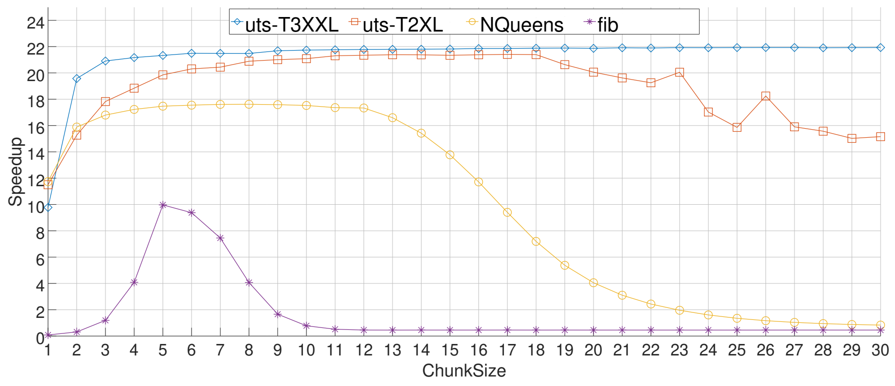


Fig. 1: Performance of several benchmarks in a 24 core system using the `parallel_stack_recursion` library with the *simple* partitioner as a function of the *chunkSize*.

Table 1: Benchmarks used. n stands for nodes and h for heights.

Name	Problem Size	Seq. Time
uts-T3XXL	Binomial tree, 2793M n, h 99049	519.867 s
uts-T2XL	Geometric [cyclic branch factor] tree, 1495M n, h 104	457.092 s
N Queens	16 x 16 board	178.696 s
fib	54st Fibonacci number	332.491 s

a concave downward curve (inverted U shape) with respect to the chunk size because of the aforementioned problems when the chunks too large or too small. Interestingly, while benchmarks such as *fib* present a very limited number of chunk sizes that provide good performance, others reach near optimal performance for a large range of values.

Given the importance of this parameter and the difficulty to predict a priori a good value for it, users should perform tests in order to choose a reasonable value for their executions. While doing this manually is not particularly difficult, it is tedious and it represents additional work that can be automated. For this reason, another contribution of our new library is an auto-tuning framework that automatically searches for the best chunk size for a given problem and environment. The framework allows to control the search process, for example, the amount of time of the search or the size of the number of tests.

5 Evaluation

Our evaluation relies on the benchmarks described in Table 1, which includes their sequential runtime in the system used in the experiments. The *uts* (Unbalanced Tree Search [21]) benchmark processes unbalanced trees of different shapes and sizes. The two main types of trees it suports are binomial and geometric. A binomial tree is an optimal adversary for load balancing strategies, since there is no advantage to be earned by choosing to move one node over another for load balance: the expected work at all nodes is identical. In geometric trees however the expected size of the subtree rooted at a node increases with proximity to the root. The *N Queens* benchmark solves the N Queens puzzle problem, which computes on how may ways can n chess queens be placed on an $n \times n$ chessboard so that no two queens threaten each other. Finally, *fib* implements the recursive algorithm to compute the *n-th* Fibonacci number. Although this is an inefficient method, it is often used in the literature of D&C and unbalanced algorithms. The enormous simplicity of its computations is particularly interesting when evaluating a parallel skeleton like ours, since it is in this kind of algorithm where the overheads of the library can be more clearly observed.

We will first analyze the performance of the skeleton, which will be followed by a study on the programmability advantages it offers. In all cases, it will be compared to the sequential version, a version developed using OpenMP, and another one based on the `parallel_recursion` algorithm template. While other approaches could not be compared for space reasons, it must be noted that `parallel_recursion` was successfully compared to other implementations in the single node experiments in [11], thus providing an approximate indirect comparison to our new proposal.

5.1 Performance evaluation

All the measurements were taken in a server with 128 GB of memory and two 2.5 GHz Intel Xeon E5-2680v3 with 12 cores each, totaling 24 cores. The codes were compiled with g++ 6.4.0 and the optimization level O3. As we will see, we measured the performance when using 6, 12 and 24 cores, always using a single thread per core. We measured separately the performance obtained using the different kinds of partitioners supported by each skeleton, tuning the user-provided function used by the custom partitioner separately for the two skeletons. The `parallel_stack_recursion` chunk size used was obtained by means of a separate auto-tuning configured to use just 10% of the runtime of the sequential version. This time is not included in the performance measurements. Further tests proved that in our experiments the performance of the chunk size obtained following this strategy was always almost identical to that of the optimal chunk size.

In all the figures, the benchmarks using `parallel_stack_recursion` will be labeled as *spar*, those using `parallel_recursion` will be known as *par*, and those that are implemented with *OpenMP* will be labeled as *omp*.

The *uts* benchmark allows generating unbalanced trees that follow different distributions and have different sizes and shapes depending on the arguments to the binary. The *T3XXL* tree is predefined in the *uts* distribution package, while *T2XL* has been added as example of a geometric tree with a circular

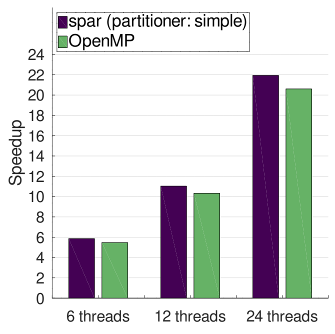


Fig. 2: Performance results of *uts-T3XXL* benchmarks

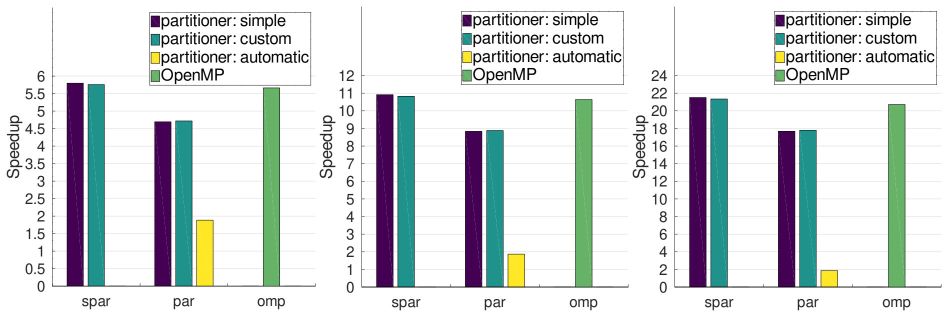
factor branch, its *uts* parameters being `-t 1 -a 2 -d 26 -b 7 -r 220`. The performance obtained on these trees is now discussed in turn.

We consider that *T3XXL* is the most challenging benchmark tried, since this binomial tree has a very large depth and extreme imbalance. Figure 2 shows the speedup achieved by our skeleton and the standard OpenMP implementation of the benchmark for 6, 12 and 24 threads, taking as baseline the standard sequential implementation. All the executions with `parallel_recursion` failed with a stack overflow error no matter the partitioner used, further confirming the interest of our new proposal. The figure only shows the results of `parallel_stack_recursion` with a *simple* partitioner because we were unable to find a custom partitioner that performed better. As we can see, our skeleton, despite strongly reducing the complexity of the code with respect to the OpenMP implementation as shown in Section 5.2, systematically offers between 6.5% and 7.1% better performance than the manually optimized OpenMP implementation.

It deserves to be mentioned that in experiments using smaller binomial trees so that `parallel_recursion` would not break, it consistently offered clearly worse performance than `parallel_stack_recursion` and OpenMP.

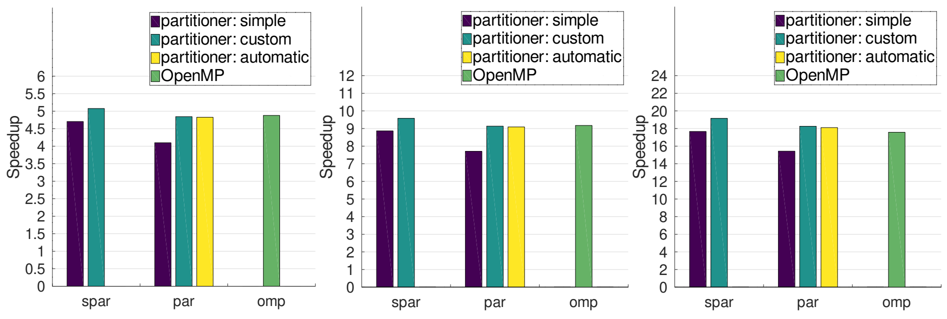
T2XL is a geometric tree with a cyclic branch factor, which makes somewhat difficult to balance its processing. As we can see in Figure 3, the use of the *simple* partitioner in `parallel_stack_recursion` is enough to obtain the best performance, there being no advantage in the use of a *custom* partitioner. The speedups obtained by `parallel_recursion` are always lower, and it is particularly interesting that the use of the *automatic* partitioner provides very bad results for this benchmark. As for OpenMP, despite the high programming cost of developing by hand this optimized code, it performs about 3% slower than our skeleton.

Figure 4 shows the results of *N Queens*. The larger complexity of the computations of this benchmark allows the *simple* partitioner to obtain quite good results, although usually worse than those of OpenMP. The exception happens at 24 threads, where fact that our new proposal always performs slightly



(a) 6 threads (b) 12 threads (c) 24 threads

Fig. 3: Performance results of *uts-T2XL* benchmarks

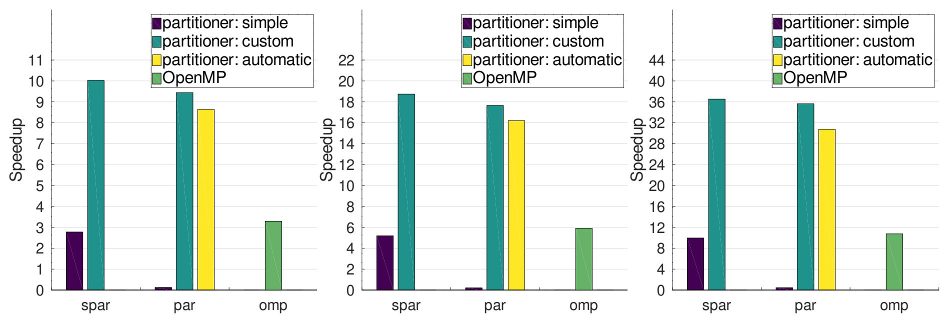


(a) 6 threads (b) 12 threads (c) 24 threads

Fig. 4: Performance results of *N Queens* benchmarks

better than `parallel_recursion` allows it to reach the same performance as OpenMP. As expected, both skeletons improve their performance when a tuned *custom* partitioner is used, our new proposal systematically offering better performance than the other implementations. This way, it is consistently $\sim 4.8\%$ faster than `parallel_recursion` and $\sim 4.4\%$ faster than OpenMP, except for 24 threads, where its advantage grows to 9%.

Figure 5 shows the performance of all the parallel implementations of *fib* developed. As mentioned before, this is a particularly challenging benchmark given the extremely lightweight nature of all the individual functions that conform it as a D&C algorithm. This is clearly reflected in the poor performance of both skeletons when a *simple* partitioner is used, as the consideration of every single Fibonacci number computation as a separate task to be managed leads to much overhead. Our new skeleton is considerably more efficient than `parallel_recursion` in this situation, being in fact 22 times faster when 24 threads are used, and reaching a performance similar to that



(a) 6 threads (b) 12 threads (c) 24 threads

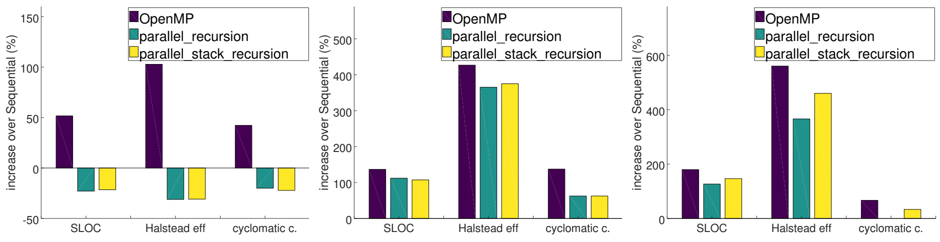
Fig. 5: Performance results of *fib* benchmarks

of OpenMP. When smarter partitioners that perform as sequential computations the calculations of Fibonacci numbers under some threshold are used, the performance of both skeletons grows considerably. In this scenario our new proposal also consistently outperforms `parallel_recursion` across all the parallel executions, although only for a small margin that decreases as the number of threads grow. Namely, the speedup of the new skeleton with respect to `parallel_recursion` goes from 6.2% for 6 threads to 2.5% for 24.

As for the absolute performance, both skeletons achieve superlinear speedups, which are in addition much higher than that of the OpenMP version, even when it also applies the strategy of computing as sequential tasks the Fibonacci numbers below a threshold for which we searched for the optimal value. Both behaviors are related to the fact, already observed and discussed in [11], that the object code that the compiler generates from our skeleton is much more efficient than the one it generates from the typical recursive implementation used by the sequential and OpenMP versions.

5.2 Programmability comparison

The best approach to measure and compare the programmability of different options is probably to rely on the observations and results from a group of programmers with a similar degree of expertise when trying to apply them. This is seldom possible, thus, our study relies on three approximate metrics of this kind. The first one is the number of source lines of code (SLOC) excluding comments and blank lines. Its value strongly depends on the programming style used and lines can widely vary in terms of complexity. A more precise metric is the Halstead programming effort [13], which estimates the complexity of the program through a formula that takes into account the number of unique operands, unique operators, total operands and total operators found in the code. The last metric computed is the cyclomatic complexity [20], defined as $V = P + 1$, where P is the number of predicates or decision points in a program.



(a) *uts* benchmark (b) *N Queens* benchmark (c) *fib* benchmark

Fig. 6: Growth of the programmability metrics of the parallel implementations with respect to the sequential one.

Figure 6, shows the relative growth of all the metrics in the parallel versions with respect to the sequential counterpart. The measurements were performed in all the cases on the whole application. As expected, given their similar API and semantics, the metrics are very similar for the two skeletons considered, which is interesting given the more complex behavior and better performance of our new proposal. The differences come basically from the changes in the `post` and `non_base` functions, although the latter one is only required in *uts*.

It is interesting that, despite relying on compiler directives, whose API is usually terser than that of libraries, the OpenMP version yields consistently worse programmability metrics than the skeletons. This way, depending on the metric used, the OpenMP version requires between 83% and 192% for more effort than `parallel_stack_recursion` for *uts*, between 28% and 72% for *N Queens* and between 14% and 25% for *fib*. An important reason for this in the case of the last two benchmarks is the need to write two versions of the algorithm in order to obtain the best performance, something which was already observed in [11]. The first version is the one invoked by the user and it contains the OpenMP directives as well as tests to decide whether the solution of a problem should rely on task parallelism or be performed as a sequential task. The second implementation is purely sequential and it takes care of these latter tasks avoiding any overhead associated to the parallelization. In the case of *uts* the sequential and OpenMP versions are much more complex than the ones based on skeletons because they have to explicitly create and manage data structures to perform the processing of the trees than in the skeleton implementations are implicitly provided by the library runtime. It is also interesting that sometimes, despite the better performance observed in Section 5.1, our skeleton offers slightly better programmability metrics than `parallel_recursion`. This is mostly associated to the simpler `post` method of `parallel_stack_recursion`, which by being restricted to a single element, avoids loops and computations on numbers of children that are required in the analogous method of `parallel_recursion`.

6 Conclusions

Divide-and-conquer is a very important pattern of parallelism that appears in many problems and can be used to implement other very relevant patterns. This makes very relevant and useful the development of tools that allow the easy and optimized implementation of this pattern, one of the best solutions being algorithmic skeletons. In this paper we have introduced `parallel_stack_recursion`, a C++ algorithmic skeleton that implements this pattern in shared memory with a focus on problems with large levels of recursion and/or high degree of unbalance. While our proposal is a complete redesign of `parallel_recursion`, a highly optimized skeleton for the same pattern, it manages to keep an almost identical interface.

Our evaluation shows that indeed the new skeleton can be applied in situations in which `parallel_recursion` breaks due to stack memory limitations, which justifies by itself its development. Furthermore, the new skeleton is on average 10.4% faster than `parallel_recursion` in the benchmarks that the latter one supports, and 4.9% faster than optimized OpenMP implementations if we disregard the *fib* benchmark, in which the compiler gives an unfair advantage to our skeletons. The maximum speedups however can go up to 22% when compared to `parallel_recursion` and 9% when compared to the OpenMP baseline, again discarding *fib*. Despite these performance advantages, the evaluation shows that the development effort associated to our new proposal is consistently similar to that of `parallel_recursion` and noticeably better than that of OpenMP. This latter observation is particularly true in the case of our largest benchmark, *uts*, in which versions not based on a skeleton have to manually define and manage data structures in order to support the highly irregular processing and the load balancing it needs to attain good performance.

As future work we plan to develop a version of this skeleton optimized for systems such as current multi-core clusters, whose optimal exploitation involves the usage of distributed and shared memory programming paradigms.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the EU (TIN2016-75845-P, TIN2016-76373-P and the predoctoral Grant of Millán Álvarez Ref. BES-2017-081320), and by the Government of Galicia (Xunta de Galicia, Spain) co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2017/04 and ED431C 2018/19) as well as under Xunta de Galicia and FEDER funds of the EU (Centro de Investigación de Galicia accreditation 2019–2022, refs. ED431G2019/01 and ED431G2019/04). We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the usage of its supercomputers.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
2. Aldinucci, M., Danelutto, M., Kilpatrick, P., Torquati, M.: FastFlow: High-Level and Efficient Streaming on Multicore, chap. 13, pp. 261–280. John Wiley & Sons, Ltd (2017)
3. Aldinucci, M., Danelutto, M., Teti, P.: An advanced environment supporting structured parallel programming in Java. Future Gener. Comput. Syst. **19**(5), 611–626 (2003)
4. Board, O.A.R.: OpenMP application program interface version 5.0 (2018)
5. Ciechanowicz, P., Kuchen, H.: Enhancing Muesli's data parallel skeletons for multi-core computer architectures. In: 12th IEEE Intl. Conf. on High Performance Computing and Communications, (HPCC 2010), pp. 108–113. IEEE (2010)
6. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press (1989)
7. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing **30**(3), 389–406 (2004)
8. Danelutto, M., De Matteis, T., Mencagli, G., Torquati, M.: A divide-and-conquer parallel pattern implementation for multicores. In: Proc. 3rd Intl. Workshop on Software Engineering for Parallel Systems, SEPS 2016, pp. 10–19. ACM (2016)
9. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: efficient C++ design for parallel skeletons. Parallel Computing **32**(7-8), 604–615 (2006)
10. González, C.H., Fraguela, B.B.: A generic algorithm template for divide-and-conquer in multicore systems. In: Proc. 12th IEEE Intl. Conf. on High Performance Computing and Communications, (HPCC 2010), pp. 79–88. IEEE (2010)
11. González, C.H., Fraguela, B.B.: A general and efficient divide-and-conquer algorithm framework for multi-core clusters. Cluster Computing **20**(3), 2605–2626 (2017)
12. Gorlatch, S., Cole, M.: Parallel skeletons. In: Encyclopedia of Parallel Computing, pp. 1417–1422. Springer (2011)
13. Halstead, M.H.: Elements of Software Science. Elsevier (1977)
14. Hosseini Rad, M., Patooghy, A., Fazeli, M.: An efficient programming skeleton for clusters of multi-core processors. Int. J. Parallel Program. p. 1094–1109 (2018)
15. Karasawa, Y., Iwasaki, H.: A parallel skeleton library for multi-core clusters. In: Proc. 2009 Intl. Conf. on Parallel Processing (ICPP'09), pp. 84–91. IEEE (2009)
16. von Koch, T.J.K.E., Manilov, S., Vasiladiotis, C., Cole, M., Franke, B.: Towards a compiler analysis for parallel algorithmic skeletons. In: Proc. 27th Intl. Conf. on Compiler Construction, CC 2018, p. 174–184 (2018)
17. Kozsik, T., Tóth, M., Bozó, I.d.: Free the conqueror! refactoring divide-and-conquer functions. Future Gener. Comput. Syst. **79**(P2), 687–699 (2018)
18. Leyton, M., Piquer, J.M.: Skandium: Multi-core programming with algorithmic skeletons. In: Proc. 18th Euromicro Conf. on Parallel, Distributed and Network-based Processing (PDP 2010), pp. 289–296. IEEE (2010)
19. Mattson, T., Sanders, B., Massingill, B.: Patterns for parallel programming. Addison-Wesley Professional (2004)
20. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering **2**, 308–320 (1976)
21. Olivier, S., Huan, J., Liu, J., Prins, J., Dinan, J., Sadayappan, P., Tseng, C.W.: UTS: An unbalanced tree search benchmark. In: Languages and Compilers for Parallel Computing (LCPC 2006), pp. 235–250. Springer Berlin Heidelberg (2006)
22. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly (2007)