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

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

Pubished online: 14 May 2021

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 of 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 imbalance among the subproblems generated and/or a deep level of recurrence. The skeleton achieves in our experiments average speedups between 11% and 18% higher than those of other solutions, reaching a maximum speedup of 78% in some tests. Nevertheless, the new proposal requires an average of between 13% and 29% less programming effort than the usual alternatives.

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

1 Introduction

Divide-and-conquer [1], hence denoted D&C, is widely used to solve problems whose solution can be obtained by dividing them into subproblems, separately

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

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 allows exploiting parallelism in this pattern, and the fact that it has a well defined structure allows expressing it by mean of algorithmic skeletons [7], which automate the management of typical patterns of parallelism [23]. 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 [21,14,9] or supporting distributed memory environments [3,8,13,6,15]. 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 [16]. In our opinion, this wide applicability makes it extremely interesting to develop highly optimized skeletons for this pattern.

A common shortcoming of the skeletons for D&C we know of are their limited mechanisms to deal effectively with problems with large degrees of imbalance among their subtasks and/or large depths of recursion. For example, Section 4 shows that very unbalanced algorithms achieve limited speedups under the optimized skeleton in [14] due to their imbalance. This is the case of *uts-T2XL*, described in Table 1, or *topsorts*, discussed in Section 4.3, which achieve speedups of 17.8 and 14.3 when using 24 threads in the evaluation platform described in Section 4, respectively. As we can see these speedups are far from the optimal values near 24 that would be desirable for these very parallel problems. There are several reasons for this inefficiency. In this regard, some skeletons only provide static load balancing, which is inadequate for irregular unbalanced problems. Most skeletons provide dynamic load balancing by turning each individual step of the algorithm into a task that can be stolen by any thread in a work-stealing scheme. However, this packaging and availability as task of every single step, together with the cost incurred by the parallel threads when competing for every task, imposes a large overhead when there are many steps or they are relatively lightweight. This problem can be ameliorated by limiting the number of tasks created and increasing their granularity. Nevertheless, in the implementations found once a task is not partitioned, it cannot be later partitioned even if this were required to achieve load balancing, thus limiting the performance. As for the problem related to deep recursions in D&C algorithms, it appears when the recursions rely on the stack, something very usual as it is natural and provides good performance. The problem appears when a deep recursion crashes the application because of the limited maximum size of the stack memory in many systems.

This paper presents a C++ parallel skeleton for the resolution of D&C problems in shared memory systems that avoids these problems. Our skeleton, called `parallel_stack_recursion`, is an evolution of the `parallel_recursion` skeleton proposed in [14] after a complete redesign and reimplementation. To solve the aforementioned issues, it provides advanced task-stealing based on stacks of tasks located in the heap, which avoids the deep recursion problems. The implementation ensures minimal contention among the threads by supporting private and shareable groups of tasks per thread as well as stealing several tasks at once. Also, it uses very light tasks, which allows the continuous creation of parallel tasks. This leads to a cheap load balancing that provides efficient executions in many scenarios where other skeletons fail or underperform. Our new implementation is available at https://github.com/fraguela/dparallel_recursion together with the material published in [14] and [15].

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

2 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.

2.1 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 the solution of a non-base problem by combining the solutions of its subproblems. The analysis performed in [14] 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

```
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`

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 a partitioner of the `simple_partitioner` class is used, the skeleton parallelizes the resolution of any non-base problem. This behavior is the only possible one in the other shared-memory skeletons we know of [21,9]. Since this partitioner breaks the resolution of every non-base problem into parallel tasks, it is the most suitable one when the cost of the subproblems generated may be very unbalanced and they have a minimum degree of granularity. This second condition is required so that the overhead of creating and scheduling every single step as a task is not counterproductive. An example of problem of this kind is the UTS [25] benchmark, whose subtasks can be extremely unbalanced, but which have a suitable minimum cost because for each subproblem various calculations of SHA1 hashes are performed to generate pseudorandom numbers.

The second kind of partitioner is the `auto_partitioner`. Under its control the skeleton applies 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 identical size. This partitioner generates sequential non-partitionable tasks once a given level of recursion of the D&C is reached in which the heuristics assume that enough tasks have been generated. For this reason it is adequate when the imbalance between the tasks is not too large and it can thus be reasonably addressed by generating a number of tasks per thread rather than a single one at the top levels of decomposition of the problem. Algorithms in which the packaging and scheduling as a parallel task of every single step would be very expensive in comparison with the cost of the step itself also benefit from this partitioner. Concrete examples of algorithms that benefit from this partitioner are the resolution of the N Queens problem, whose high level subtask are not too unbalanced, or the recursive computation of Fibonacci numbers, in which each step is very lightweight.

Finally, with the `custom_partitioner` users decide when to apply parallelism by means of a `do_parallel(t)` member function that they must provide in the *info* object. The function must return a boolean that indicates whether problem `t` should be solved using parallelism or not. This partitioner is interesting for problems with meaningful imbalance where the user can estimate whether it is worth to parallelize a problem in order to promote load balancing or, on the contrary, it is a bad idea because the problem is too small. For example, the Fibonacci computation can improve its performance with this partitioner if we fine-tune when it is actually worth to parallelize a step. Another example is the floorplan benchmark [12], which computes the optimal floorplan distribution of a number of cells. While its tasks are too small for a simple partitioner, they are too unbalanced for the automatic partitioner. Thus only a user-tuned custom partitioner can find the best balance between creating enough parallel tasks for the load balancing and avoiding excessive overhead for the task creation.

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`, 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

```
struct tree_t {
  int val;
  std::vector<tree\_t*> children;
};

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`

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.

2.2 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 [27] 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 [14], the skeleton can provide good performance for some imbalanced problems by generating more tasks than threads and letting the TBB scheduler balance them. However, sometimes this does not work well because the imbalance 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. As an example, Section 4.3 shows that in the *topsorts* application our new proposal obtains speedups between 28% and 43% larger than those of `parallel_recursion` in executions using between 6 and 24 cores. As another example, in the same execution environment, the UTS [25] benchmark with the input *uts-T2XL* described in Table 1 attains speedups 22% higher than this skeleton in the same environment. The reason in both cases is that the workload is very imbalanced and this skeleton cannot balance it among the threads as efficiently as our new proposal. And this is despite the fact that `parallel_recursion` does perform some load balancing. This way, for example its speedups for the uts-T2XL problem are between 3.5 and 9.4 times larger that those obtained in a parallel execution without any load balancing when using between 6 and 24 threads in the same system, respectively.

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. This problem is also illustrated in our evaluation in Section 4, as we will see that this skeleton cannot support the input *uts-T3XXL* from Table 1 for the reasons just stated, while our new skeleton successfully parallelizes it.

3 A new D&C algorithm template

Given the nature of the problems of the `parallel_recursion` skeleton described in Section 2.2, 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.

3.1 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. This allows us to avoid the framework limitations and achieve full control over the implementation and a much greater margin of customization. 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 [21]. 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 from 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 granularity of a single problem muy be too small. The second one is that each steal has a non negligible overhead, so it is 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.

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 of at least two chunk sizes, 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 at least *chunkSize* elements. If this is the case, the *chunkSize* problems at the top of the shared section, which are 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 *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

```
FUNCTION $ThreadExecution(threadId, body, info, result)$ :
while not($allThreadStacksEmpty()$) do
    while not($stack[threadId].localSectionEmpty()$) do
        $item \leftarrow stack[threadId].pop()$
        $body.pre(item)$
        if $info.is\_base(item)$ then
            $base\_res \leftarrow body.base(item)$
            $body.post(base\_res, result)$
        else
            $body.pre\_rec(item)$
            if $body.process\_non\_base$ then
                $non\_base\_res \leftarrow body.non\_base(item)$
                $body.post(non\_base\_res, result)$
            end if
            if $info.doParallel(item)$ then
                for $i \leftarrow 0$ to $info.num\_children(item)$ do
                    $child \leftarrow body.child(item, i)$
                    $stack[threadId].push(child))$
                end for
                $stack[threadId].releaseWorkToSharedSection()$
            else
                $computeSequentially(item, body, info, result)$
            end if
        end if
    end while
    if $stack[threadId].stealWorkFromSharedSection() == FAIL$ then
        $victimId \leftarrow findVictimWithWork()$
        $stolenWork \leftarrow stack[victimId].stealWork()$
        $stack.push(stolenWork)$
    end if
end while
END FUNCTION
```

Listing 3: Pseudocode of the thread main function of `parallel_stack_recursion`

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.

Listing 3 summarizes as a pseudocode the operation of each thread in our skeleton following the strategy just described. We can can see that the *body* object presents a somewhat different interface from the one of `parallel_recursion`. This change and its motivation is explained in Section 3.2.

3.2 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.

3.2.1 Body object

Given the implementation described in Section 3.1, a problem is destroyed as soon as its children are generated, which makes it impossible to use the `post` member function described in Section 2.1. 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

```
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 2.1 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 4. The listing does not include the `TreeAddInfo` class because

```
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 4: Reduction on a tree using `parallel_stack_recursion`

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`.

3.2.2 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

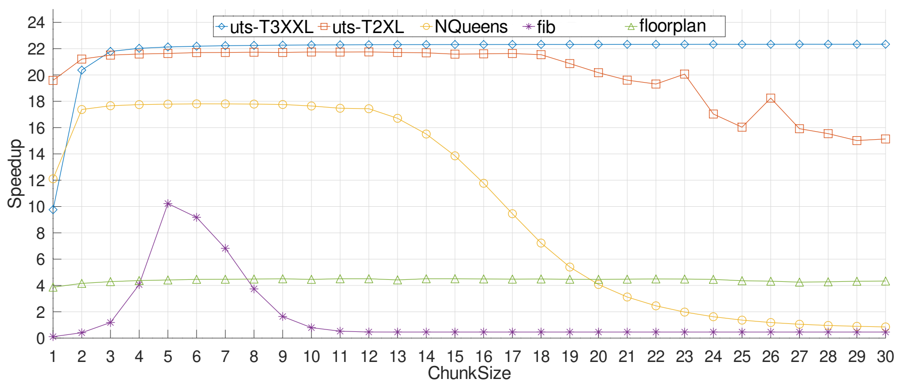


Fig. 1: Performance of several benchmarks in a 24 core system using the `parallel_stack_recursion` library with the *simple* partitioner as a function of the *chunkSize*.

same problems of imbalance 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 4 does not use the automatic partitioner used in Listing 2.

3.3 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.

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	54^{st} Fibonacci number	332.491 s
floorplan	Optimal placement of 20 cells in a floorplan	6.381 s
knapsack	44 total items	293.059 s

Figure 1 illustrates the comments we have just made by representing the performance of some benchmarks used in our evaluation in Section 4 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 4, and a simple partitioner. We can see that the performance is basically a concave downward curve (inverted U shape) with respect to the chunk size because of the aforementioned problems when the chunks are 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.

4 Evaluation

The core of 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 [25]) 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 many ways can n chess queens be placed on an $n \times n$ chessboard so that no two queens threaten each other.

The *fib* benchmark 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.

Two benchmarks from the Barcelona OpenMP Tasks Suite [12] are included, *floorplan* and *knapsack*. The *floorplan* benchmark computes the optimal floorplan distribution of a number of cells. This benchmark needs certain adaptations so that it can be treated as a D&C problem, and it is interesting because it is a memory-bound problem, where most of the runtime is making continuous memory copies, limiting its scalability. For the *knapsack* benchmark, given a set of items, each with a weight and a value, the application determines the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. These two benchmarks come with an aggressive pruning mechanism based on the best result found up to that moment to reduce the search space, which allows achieving large speedups and a high scalability. This effect is not very noticeable in *floorplan*, specially because this is a memory-bound application with a short sequential execution time, but is very noticeable in *knapsack*.

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, a version developed using Cilk and another one based on the `parallel_recursion` algorithm template. For Cilk and OpenMP, two different implementations are made, one that does not limit the creation of tasks (*simple*), and another one manually tuned to limit the number of tasks created by a cut-off mechanism (*manual*). In these manual versions the code inside a non-partitionable task is purely sequential, as in our skeleton, and we tuned each one of their executions seeking the cut-off condition that led to the best possible performance. The OpenMP versions are implemented with OpenMP tasks and they rely on custom *if* statements in the *manual* version, which turned out to be more efficient than using the OpenMP *if* clause. We are using tied tasks, and although we also tested the use of untied tasks the performance was very similar. For `parallel_recursion` and `parallel_stack_recursion` the use of the *simple* partitioner will be know as *simple* version and the use of the *custom* partitioner will be know as *manual* version. The `parallel_recursion` tests will include a third *automatic* version that relies in its *automatic* partitioner.

The only exception to the explanations above is the *uts* OpenMP benchmark, for which a single version, namely the highly optimized standard OpenMP version provided by the authors will be used. This code does not use tasks and rather relies on a single parallel section with an optimized manual work-stealing stack-based strategy similar to our new implementation.

While other approaches could not be compared for space reasons, it must be noted that `parallel_recursion` was successfully compared to the recent

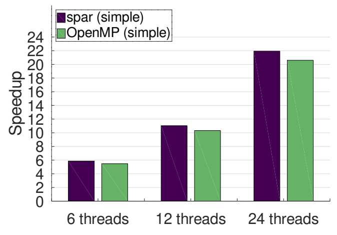


Fig. 2: Performance results of *uts-T3XXL* benchmarks.

D&C skeleton [9] in the single node experiments in [15], thus providing an approximate indirect comparison to our new proposal.

As final step, our skeleton will be evaluated in the scope of an original application consisting in the resolution of the topological sorts problem following the reverse search approach.

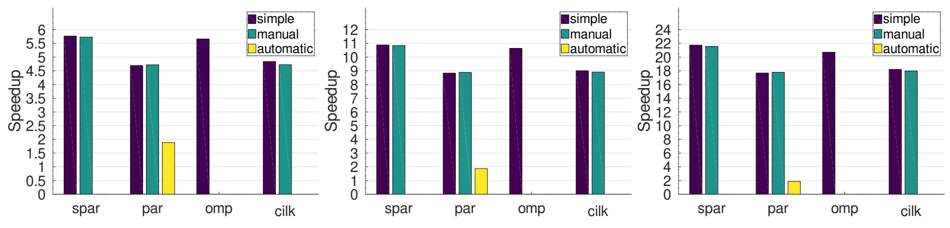
4.1 Performance evaluation

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. We measured the performance when using 6, 12 and 24 cores, always using a single thread per core. 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. This percentage should not be taken as a rule and may vary for other benchmarks, we simply consider that for our examples it is enough to provide a close to optimal chunk size with reliability. 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*, those that are implemented with *OpenMP* will be labeled as *omp* and those that are implemented with *Cilk* will be labeled as *cilk*.

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 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 tree sn very deep and extremely unbalanced. Figure 2 shows the speedup with respect to the standard sequential implementation achieved by our skeleton and the OpenMP UTS implementation of the benchmark for 6, 12 and



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

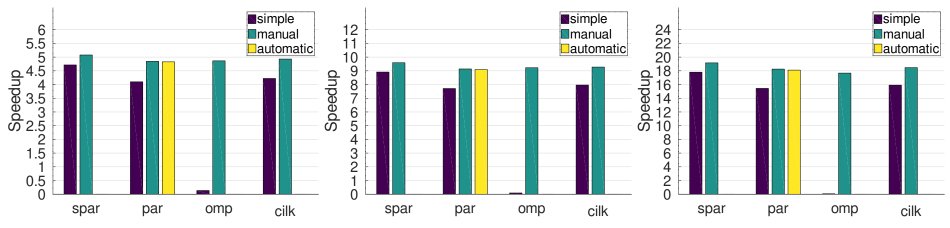
Fig. 3: Performance results of *uts-T2XL* benchmarks.

24 threads. 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. Similarly, all the executions with Cilk failed due the fixed spawn deep limit of the framework, which is currently hardcoded. 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 4.2, systematically offers between 6.5% and 7.1% better performance than the 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. Experiments with smaller binomial trees using *Cilk* were unsuccessful and kept failing.

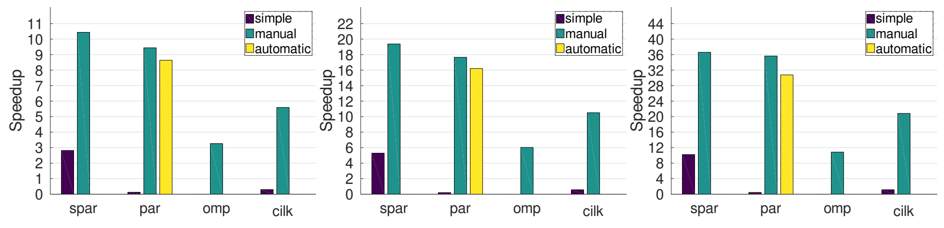
T2XL is a geometric tree with a cyclic branch factor, which makes somewhat difficult to balance its processing. As can be seen in Figure 3, the use of the *simple* partitioner in `parallel_stack_recursion` is enough to obtain the best performance, there being no advantage in the use of a *custom* partitioner. The speedups obtained by Cilk and `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 its standard manually optimized version, it performs about 2.7% 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, better than the *simple* version of all the other benchmarks, although slightly worse than those of *manually* tuned OpenMP and Cilk. As expected, all tuned *custom* benchmarks improve their performance, with our new proposal systematically offering better performance than the other implementations. This way, it is consistently $\sim$ 4.9% faster than `parallel_recursion`, $\sim$ 3.4% faster than Cilk and $\sim$ 4.2% faster than OpenMP, except for 24 threads, where its advantage grows to 8.4%.



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

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

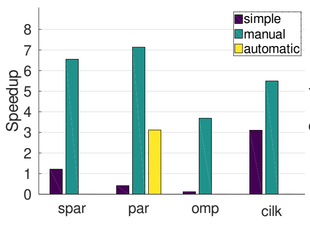


(a) 6 threads (b) 12 threads (c) 24 threads

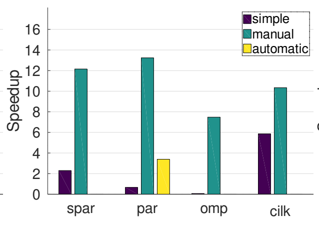
Fig. 5: Performance results of *fib* benchmarks.

Figure 5 shows the performance of all the parallel implementations of *fib*. 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 all *simple* implementations, 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 the others in this situation, being in fact 22 times faster than `parallel_recursion` when 24 threads are used, and reaching a performance similar to that of *manual* tuned OpenMP. The performance of all implementations grows considerably for the *manual* versions, where the calculations of Fibonacci numbers under some threshold are done sequentially. In this scenario our new proposal also consistently outperforms the other implementations across all the parallel executions, although for the `parallel_recursion` 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 10.6% for 6 threads to 2.7% for 24.

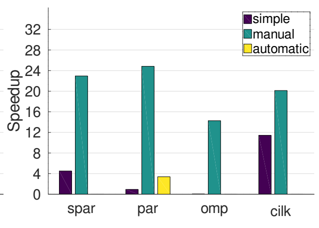
As for the absolute performance, both skeletons achieve superlinear speedups, which are in addition much higher than that of the OpenMP and Cilk version, even for its *manual* versions. Both behaviors are related to the fact, already observed and discussed in [15], that the object code that the compiler generates from our skeleton is much more efficient than the one it



(a) 6 threads

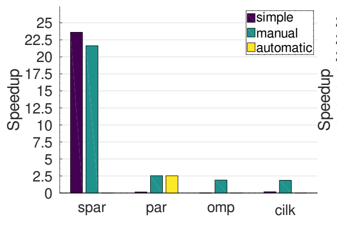


(b) 12 threads

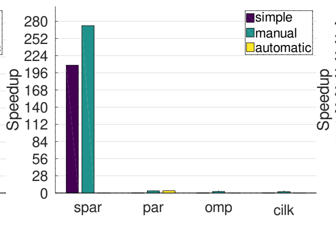


(c) 24 threads

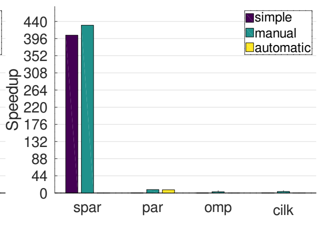
Fig. 6: Performance results of *floorplan* benchmarks.



(a) 6 threads



(b) 12 threads



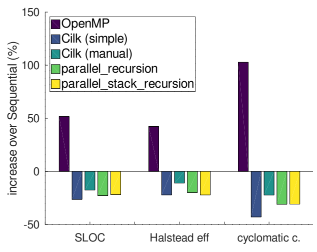
(c) 24 threads

Fig. 7: Performance results of *knapsack* benchmarks.

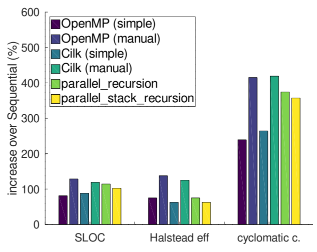
generates from the typical recursive implementation used by the sequential, OpenMP and Cilk versions.

Figure 6 shows the results of *floorplan*. For the *simple* implementation of benchmarks, Cilk provides the best performance. The adaptations performed in the `parallel_recursion` and `parallek_stack_recursion` versions to implement this benchmark as a D&C problem prevent us from achieving good performance with the *simple* partitioner. Using the *automatic* partitioner for `parallel_recursion` only helps to some extent this problem. However, it is possible to achieve higher performance than the rest of the implementations when we use a *custom* partitioner in the *manual* versions, being our new skeleton a little behind `parallel_recursion` in this case.

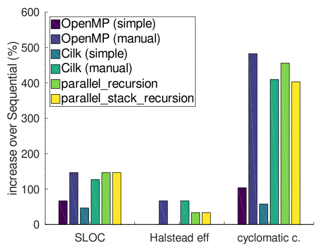
Finally, Figure 7 shows the results of *knapsack*. The prune system implemented in this benchmark causes the final execution time to be highly dependent on the order of execution. All the benchmarks were written to ensure that they all execute the loop iterations in the same order. Despite this effort, in a multithread environment, each implementation has its own behavior and distributes the tasks among the threads differently, giving place to great differences in performance. The task distribution of `parallel_stack_recursion` is the best for this benchmark by far.



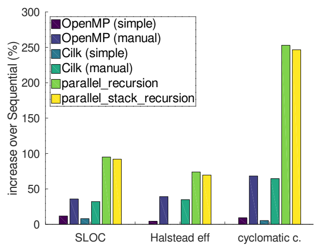
(a) *uts* benchmark



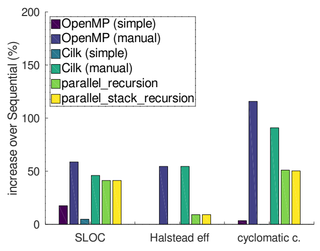
(b) *N Queens* benchmark



(c) *fib* benchmark



(d) *floorplan* benchmark



(e) *knapsack* benchmark

Fig. 8: Growth of the programmability metrics of the parallel implementations with respect to the sequential one.

4.2 Programmability comparison

The best approach to measure 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 [29]. 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 [17], 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 [24], defined as $V = P + 1$, where P is the number of predicates or decision points in a program.

Figure 8, 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*.

The *simple* implementations using OpenMP and Cilk are the best in terms of programmability, but as we have seen in the previous section, they offer suboptimal performance and even infeasible performance for all benchmarks. It is interesting that, despite relying on compiler directives, whose API is usually terser than that of libraries, the OpenMP and Cilk *manual* versions yield consistently worse programmability metrics than the skeletons. This way, depending on the metric used, the OpenMP *manual* version requires between 83% and 193% for more effort than `parallel_stack_recursion` for *uts*, between 13% and 46% for *N Queens*, between 16% and 25% for *fib* and between 12% and 44% for *knapsack*. An important reason for this in the case of the OpenMP and Cilk *manual* benchmarks is the need to write two versions of the algorithm in order to obtain the best performance, something which was already observed in [15]. 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.

The case of *floorplan* is an exception where the skeletons require more programming effort than the other implementations. This is caused by the fact that some adaptations had to be made to be able to treat this problem as a D&C problem. Also, the sequential version had to be added to the code of the skeletons to activate optimizations in the manual version that avoids memory copies that are not necessary for a sequential execution. This extra effort that the user must make is rewarded by obtaining the best performance of all the benchmarks using a custom partitioner, even for a problem that was difficult to adapt as a D&C problem.

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 that in the skeleton implementations are implicitly provided by the library runtime. It is also interesting that sometimes, despite the better performance observed in Section 4.1, 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`.

4.3 Practical original application

The benchmarks *floorplan* and *knapsack* used in the previous sections solve actual problems in a realistic fashion and could be thus part of an application. However, in this section our skeleton will be further evaluated in the context of a totally original application not taken from any benchmark suite. This application, called *topsorts*, computes the topological sorts of a direct acyclic graph using the reverse search approach for enumeration [5]. A topological sort

Table 2: Performance metrics of the *topsorts* application

	Performance Metrics		
	Threads	Execution Time	Speedup
sequential	1	4795.639 s	-
`parallel_recursion`	6	1108.858 s	4.325
	12	617.832 s	7.762
	24	336.389 s	14.256
`parallel_stack_recursion`	6	868.439 s	5.522
	12	471.229 s	10.177
	24	235.966 s	20.323

or order is a linear ordering of the nodes where for each directed edge from node A to node B, node A appears before node B in the ordering. Topological orderings are not unique and have a variety of practical applications, for example, in resolving dependencies when building a program, or in general when scheduling any set of tasks with dependencies. Demonstrating the parallelization of this reverse search process implemented as a D&C algorithm by means of our skeleton is very interesting, as this strategy has many applications in a wide range of fields, from Biomechanics to Chemistry or Robotics [31].

Since the APIs of `parallel_recursion` and `parallel_stack_recursion` are very similar, we parallelized the application with both skeletons. As sequential baseline we used an optimized version of the original code, while the input was a graph with 42 nodes and 61 edges.

Table 2 shows the performance of the sequential and the parallel versions when using 6, 12 and 24 threads. The *simple* partitioner was used because we could not find a heuristic from which a *custom* partitioner could benefit in this very imbalanced problem. The *automatic* partitioner was also tested for `parallel_recursion`, but the performance was very poor, resulting in a speedup of ≈1.83 regardless of the number of threads used. As we can see, the performance and scalability obtained by `parallel_stack_recursion` is noticeably better than the one provided by `parallel_recursion`, while the effort made in the implementation is basically identical given the analogous API. In this regard, our parallel versions only required 11 more lines of code than the sequential implementation, which we find very reasonable given the performance obtained and the irregularity of the problem.

In conclusion, the feasibility of the implementation of reverse search by means of our library demonstrates that it can be successfully used in a large set of applications for solving practical problems. With relatively little effort from the programmer, the skeleton provides automatic parallelization with dynamic load balancing that allows obtaining good performance in multithreaded executions.

5 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 *Skandium* [21], which follows a producer-consumer model in which the tasks are pushed and popped from a shared ready queue by the participating threads. The *DAC* parallel template [9] supports three different underlying runtimes, OpenMP [26], Intel TBB [27] 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 `parallel_recursion` skeleton [14], explained in detail in Section 2. This skeleton supports an optional 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 parallelize 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 have seen, even with this higher degree of control, `parallel_recursion` is not well suited to problems that exhibit a large degree of imbalance.

Another task-based approach that cares about the task granularity is [30], which implements an automatic granularity control. At compile-time it generates multiple versions of each task, increasing granularity by task unrolling and subsequent removal of superfluous synchronization primitives. A runtime heuristic automatically chooses the code version to execute at each task spawning point.

In the distributed memory realm we find *eSkel* [8], 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* [6], 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* [13], 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 [18], 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 have seen, can be very inefficient. Another work in this area is *dparallel_recursion* [15], an evolution of [14] in which the shared memory portion relies on [14] 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 tools to identify computational patterns and refactor the codes containing them [19,20] 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.

There are other high level proposals that support the parallelization of D&C problems beyond skeletons. For example, Petabricks [4] proposes a new implicitly parallel programming language and compiler. Programs written in this language can naturally describe multiple algorithms for solving a problem and how they can be fit together. This information is used by the compiler and runtime to create and autotune an optimized hybrid algorithm. In this category is also the *Merge* [22] framework, which couples a new language based on map-reduce and associated compiler with a dynamic runtime that automatically distributes computations among different cores in a heterogeneous multi-core system.

Internally, our new proposal bases its implementation on the use of a stack where the problems that must be distributed and processed are stored, as can be found in previous works [28]. The proposed skeleton uses work-stealing, a technique to distribute the work of the stack that has turned out to be very flexible in most situations and presents good scalability. There are different libraries that use work-stealing, such as [11] for distributed memory or [10] for shared memory. Both libraries present several differences, but they have in common with our approach that they divide the stack queue in two sections, a local section and a shared section. In [32] a variety of solutions that use different types of work-stealing techniques are collected and analyzed, each one being adapted to their specific needs and having its advantages and drawbacks. Our implementation relies on some of these strategies and adapts them to specifically suit to D&C problem solving.

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 imbalance. 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 11.2% faster than `parallel_recursion` in the benchmarks and applications that the latter one supports excluding *knapsack*, where our library is greatly benefited from the prune system. The new approach is also 18.0% and 13.1% faster than optimized OpenMP and Cilk implementations respectively if we disregard the *knapsack* and *fib* benchmarks, where for the latter the compiler gives an unfair advantage to our skeletons. The maximum speedups however can go up to 42.6% when compared to `parallel_recursion`, 77.6% when compared to the OpenMP and 20.9% when comparex to the Cilk, again discarding *fib* and *knapsack*. 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 optimized OpenMP and Cilk. 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. The standard non-optimized OpenMP and Cilk versions have best programmability efforts metric than the skeleton implementations, but a worse performance in all benchmarks. Finally, an unbalanced practical application has also been parallelized using the new skeleton, obtaining a very good performance with relatively little effort.

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 Science and Innovation of Spain (TIN2016-75845-P, PID2019-104184RB-I00 and PID2019-104834GB-I00, AEI/FEDER/EU, 10.13039/501100011033) and the predoctoral Grant of Millán Álvarez Ref. BES-2017-081320), and by the Xunta de Galicia cofounded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2017/04 and ED431C 2018/19). We acknowledge also the support from the Centro Singular de Investigación de Galicia "CITIC" and the Centro Singular de Investigación en Tecnoloxías Intelixentes "CiTIUS", funded by Xunta de Galicia and the European Union (European Regional Development Fund- Galicia 2014-2020 Program), by grants ED431G 2019/01 and ED431G 2019/04. We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of their computers.

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. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: PetaBricks: A language and compiler for algorithmic choice. In: Proc. 30th ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '09, p. 38–49. ACM (2009)
5. Avis, D., Fukuda, K.: Reverse search for enumeration. Discrete Applied Mathematics **65**(1), 21–46 (1996)
6. 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)
7. Cole, M.: Algorithmic skeletons: structured management of parallel computation. MIT Press (1989)
8. Cole, M.: Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Computing **30**(3), 389–406 (2004)
9. 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)
10. van Dijk, T., van de Pol, J.C.: Lace: Non-blocking split deque for work-stealing. In: Euro-Par 2014: Parallel Processing Workshops, pp. 206–217. Springer International Publishing (2014)
11. Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J.: Scalable work stealing. In: Proc. Conf. on High Performance Computing Networking, Storage and Analysis, SC '09. ACM, New York, NY, USA (2009)
12. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: 2009 International Conference on Parallel Processing, pp. 124–131 (2009)
13. Falcou, J., Sérot, J., Chateau, T., Lapresté, J.T.: Quaff: efficient C++ design for parallel skeletons. Parallel Computing **32**(7-8), 604–615 (2006)
14. 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)
15. 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)
16. Gorlatch, S., Cole, M.: Parallel skeletons. In: Encyclopedia of Parallel Computing, pp. 1417–1422. Springer (2011)
17. Halstead, M.H.: Elements of Software Science. Elsevier (1977)
18. 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)

19. 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)
20. Kozsik, T., Tóth, M., Bozó, I.d.: Free the conqueror! refactoring divide-and-conquer functions. Future Gener. Comput. Syst. **79**(P2), 687–699 (2018)
21. 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)
22. Linderman, M.D., Collins, J.D., Wang, H., Meng, T.H.: Merge: A programming model for heterogeneous multi-core systems. In: Proc. 13th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII, p. 287–296. Association for Computing Machinery, New York, NY, USA (2008)
23. Mattson, T., Sanders, B., Massingill, B.: Patterns for parallel programming. Addison-Wesley Professional (2004)
24. McCabe, T.J.: A Complexity Measure. IEEE Transactions on Software Engineering **2**, 308–320 (1976)
25. 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)
26. OpenMP Architecture Review Board: OpenMP application program interface version 5.0 (2018)
27. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly (2007)
28. Rudolph, L., Slivkin-Allalouf, M., Upfal, E.: A simple load balancing scheme for task allocation in parallel machines. In: Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures, (SPAA'91), pp. 237–245. ACM (1991)
29. Teijeiro, C., Taboada, G.L., Touriño, J., Fraguela, B.B., Doallo, R., Mallón, D.A., Gómez, A., Mouriño, J.C., Wibecan, B.: Evaluation of UPC programmability using classroom studies. In: Proc. Third Conf. on Partitioned Global Address Space Programing Models, PGAS '09, pp. 10:1–10:7. ACM (2009)
30. Thoman, P., Jordan, H., Fahringer, T.: Adaptive granularity control in task parallel programs using multiversioning. In: Euro-Par 2013 Parallel Processing, pp. 164–177. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
31. White, J.L.: Reverse search for enumeration — applications. http://cgm.cs.mcgill.ca/~avis/doc/rs/applications/index.html (2008). Accessed: 2021-03-06
32. Yang, J., He, Q.: Scheduling parallel computations by work stealing: A survey. Int. J. Parallel Program. **46**(2), 173–197 (2018)