A General and Efficient Divide-and-conquer Algorithm Framework for Multi-core Clusters

Carlos H. González · Basilio B. Fraguela

the date of receipt and acceptance should be inserted later

Abstract Divide-and-conquer is one of the most important patterns of parallelism, being applicable to a large variety of problems. In addition, the most powerful parallel systems available nowadays are computer clusters composed of distributed-memory nodes that contain an increasing number of cores that share a common memory. The optimal exploitation of these systems often requires resorting to a hybrid model that mimics the underlying hardware by combining a distributed and a shared memory parallel programming model. This results in longer development times and increased maintenance costs. In this paper we present a very general skeleton library that allows to parallelize any divide-and-conquer problem in hybrid distributed-shared memory systems with little effort while providing much flexibility and good performance. Our proposal combines a message-passing paradigm at the process level and a threaded model inside each process, hiding the related complexity from the user. The evaluation shows that this skeleton provides performance comparable, and often better than that of manually optimized codes while requiring considerably less effort when parallelizing applications on multi-core clusters.

Keywords parallelism · libraries · C++ · template metaprogramming · distributed memory · shared memory · programmability · skeletons · multi-core clusters

Universidade da Coruña
A Coruña, Spain.
Phone: +34-881-011-219. Fax: +34-981-167-160
E-mail: {cgonzalezv, basilio.fraguela}@udc.es

1 Introduction

Parallelism, whose exploitation is never trivial, is nowadays ubiquitous in every kind of software. In addition, many applications require the usage of clusters either because their memory requirements exceed the capacity of a single node or because they need a large number of processors to complete their computations in a reasonable time or both. Since the appearance of multi-core processors every cluster is a hybrid distributed-shared memory system, as each node contains its own separate memory, which is shared by one or more local multi-core processors. The applications that run in these clusters require a programming paradigm suitable for distributed memory in order to cope with the distributed memory nodes, while they can take advantage of the parallelism inside each node by means of either distributed or shared memory programming models, the latter ones usually providing the best performance thanks to the reduced communication and synchronization costs within the shared memory of a node. The usage of two programming models in order to achieve the best performance, where one or both are often relatively low-level, results in low programmability and therefore increased programmer effort and costs. This has motivated extensive research on the improvement of the programmability of these systems, which has led to proposals such as the Partitioned Global Address Space (PGAS) paradigm [46], which offers a global view of the data in an application together with information on the locality of each portion of the data to each processors. Unfortunately these approaches have not been widely adopted for different reasons, important ones being their suboptimal performance [31] and code reusability, since many proposals are new languages. As a result, most current high performance codes for hybrid distributed-shared memory systems are still written using message passing (typically on MPI), often combined with shared memory solutions such as OpenMP, and the best strategy to program these systems is still an open problem.

In this paper we explore the efficient programming of hybrid distributed-shared memory systems following the algorithmic skeleton approach [8], which identifies typical patterns of parallelism [32] and automates their management by means of predefined skeletons that hide the complexity of the parallel implementation from the programmer. Namely, we propose an algorithmic skeleton for the well-known divide-and-conquer pattern of parallelism, which parallelizes the divide-and-conquer problem resolution strategy [1]. We chose to parallelize this strategy for two reasons. The first one is that it is widely applicable, appearing in fact in many crucial algorithms in different fields [23,15,4,49,34,48,39,26]. The second one is that, if properly designed, this skeleton also allows to express simpler common computation patterns such as the ones provided by the higher-order functions map and reduce [19], thus covering even more problems.

Our proposal is a substantial extension of [16], which was restricted to shared memory systems. Our new algorithm template not only efficiently combines two programming models in order to try to achieve the best performance in hybrid distributed-shared memory systems, but it also provides an enormous flexibility for the execution of the divide-and-conquer algorithms in these systems, as we will see. All this is achieved with an easy-to-use high-level interface that requires small effort from the user. The main contributions of this work are:

- We present the first divide-and-conquer skeleton optimized for hybrid distributed-shared memory systems we know of.
- The skeleton has a large configurability that allows it to adapt to the required input and output conditions as well as to control its internal behavior in several ways.
- Our proposal allows to define, build and operate on arbitrary distributed data structures, even with partial replications, that are amenable to the application of this skeleton.
- Our library provides novel and handy mechanisms to optimize data transfers of complex data structures.
- We present a demanding evaluation of our algorithm template that compares it both in terms of performance and programmability with hand-optimized versions based on two of the most popular tools used in applications parallelized for computer clusters, MPI and OpenMP.
- The library is also favorably compared to two state-of-the-art tools that are particularly well suited to parallelize the divide-and-conquer pattern, namely Cilk Plus [24] and the most recent skeleton for divide-and-conquer that we found [10].
- The software package is made publicly available at https://github.com/fraguela/dparallel_recursion under an open-source license.

The remainder of this manuscript is organized as follows. Section 2 reviews the related work. Then, Sect. 3 discusses the key aspects of the divide-and-conquer pattern of parallelism and presents an algorithm template that implements this pattern in shared memory systems. Section 4 analyses the challenges of the implementation of this skeleton in hybrid memory systems and presents our new skeleton. This is followed by an evaluation in Sect. 5 and our conclusions and future work in Sect. 6 .

2 Related work

Divide-and-conquer [1], hence denoted D&C, is a very widely applicable strategy, therefore it has been implemented in many libraries of skeletal operations. While some of them are restricted to shared-memory environments [29,16,10], including the first skeleton designed to parallelize irregular problems [18], many others support distributed-memory systems, enabling the use of clusters. This multiplies by the number of nodes existing in the cluster both the amount of parallelism and the amount of memory available to the problem to be solved, having in exchange to deal with the complexities inherent to distributed memory. Unfortunately, almost all of the libraries in this second group only provide distributed-memory parallelism, which can severely restrict the performance and the scalability in current multi-core clusters, as we will see in our evaluation in Sect. 5.1. However, that is not the only difference with our proposal. For example, the fact that eSkel [9] relies on C precludes it from benefiting from the large advantages that object-oriented languages provide to the development of libraries such as encapsulation or polymorphism. As a result its API results somewhat low-level, exposing many MPI-specific implementation details. Lithium [2] is a skeleton library for Java that exploits a macro data flow implementation schema instead of the more usual implementation templates, and largely enjoys the advantages of objects, including run-time polymorphism. Our library however almost exclusively uses approaches resolved at compile time, and thus cheaper, such as static polymorphism and C++ template metaprogramming. Another library that heavily relies on these latter techniques is Quaff [13], as its task graph must be encoded by the programmer by means of type definitions from which the compiler produces optimized message-passing code. This static generation of the tasks implies that, unlike our proposal, Quaff cannot dynamically generate new parallel tasks depending on runtime conditions.

SkeTo [25] and Muesli [7] are the two libraries of skeletal operations we know of that have made an effort to better adapt to multi-core clusters. They also have in common with our proposal that they are written in C++ and they support distributed memory parallelism on top of MPI. However, SkeTo centers around data-parallel skeletons on distributed data-types it provides and offers no support for task parallel skeletons. Thus, it does not provide any D&C skeleton. As for Muesli, its adaption to hybrid memory systems, based on OpenMP, was only performed on its data-parallel skeletons. As a result, its D&C skeleton is built for pure distributed-memory systems. In addition, while our library heavily uses on template metaprogramming and static polymorphism, Muesli reliance on runtime polymorphism leads to large overheads for simple applications in [27].

Skeletons are not the only high-level approach suitable to parallelize D&C algorithms. For example, Cilk [5], and more recently, Cilk Plus [24], largely simplify their implementation by means of keywords to spawn and synchronize parallel tasks. Cilk Plus provides some additional facilities such as simple loop parallelization or specific support for reductions, but also only within shared-memory environments. The Java-based Satin system [43], which also relies on spawn-sync primitives and was recently extended with support for heterogenous many-cores [22], allows to parallelize D&C problems in distributed memory environments adding many additional features such as replicated shared objects, speculative parallelism, fault tolerance, malleability and cluster-aware stealing for load balancing. The lack of knowledge on the structure of the problem does not allow Satin to implement global-level optimizations enabled by our proposal such as broadcasts or gather/scatter operations and puts the user in charge of the explicit parallelization and synchronization of the required tasks. Tools that simplify the exploitation of task-level parallelism by analyzing the dependencies between tasks and managing their execution in order to provide a data-flow model avoid this latter shortcoming in the parallelization of D&C problems. This is the case of DepSpawn [17] and ClusterSs [41], although compared to our skeleton, while the first one is restricted to shared-memory systems, the second one does not support nested spawning of tasks.

Finally, many big data processes can be seen as D&C algorithms, and there are several specialized frameworks to support them [11,45,47]. These tools operate at a different level to that of our proposal, not only because they have been particularly designed to manipulate large amounts of data, but also because they provide features that can be critical for this kind of processes such as high performance distributed file systems, resource management, or resilience. In situations in which these features were not required, or our skeleton could be complemented with modules that provided them, our proposal could be an interesting alternative to these frameworks.

Other contributions and differences of our work with respect to the approaches discussed above are the possibility of building and supporting arbitrary truly distributed data structures that can be reused across different algorithms and the facilitation of several optimizations that can have an important impact on performance.

3 A divide-and-conquer skeleton for shared memory systems

The divide-and-conquer strategy applies to problems whose solution can be obtained from solutions of smaller separate subproblems into which the problem can be divided. Since the subproblems usually have the same nature as the original one, this strategy gives place to a recursive subdivision that stops when a base case is detected. Also, the independence of the subproblems naturally enables parallelism, the D&C pattern of parallelism [32] being in fact one of the most commonly applicable and used. For this reason this pattern is supported by several libraries of algorithmic skeletons, as we have seen in Sect. 2. In the remainder of this section we describe in detail the approach taken by [16], a C++ D&C algorithm template for shared-memory systems called `parallel_recursion`, as it is the base for our work.

A simple analysis of the D&C parallel pattern shows that it consists of four basic blocks: the determination of whether a problem is a base case that must be solved at once or a decomposable one, the resolution of a base case, the subdivision of a non-base case, and finally the combination of the results of the subproblems of a decomposable problem. A more careful analysis reveals that these components can be classified in two groups. One of them, which is comprised of the identification of the base case and the subdivision in subproblems of non-base cases, is more strongly related to the structure of the problem, which is usually directly related to the data structure(s) used to represent it. This way, if we

```
template<typename T, int N>
struct Info : Arity<N> {

  bool is_base(const T& t) const; //base case detection

  //number of subproblems of t
  int num_children(const T& t) const;

  //get i−th subproblem of t
  T child(int i, const T& t) const;
};

template<typename T, typename S>
struct Body : EmptyBody<T, S> {

  void pre(T& t); //preprocessing of t

  S base(T& t); //solve base case

  S post(T& t, S *r); //combine children solutions
};
```

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

apply different algorithms that can be accommodated to the D&C strategy (e.g. finding the minimum value, adding all the values, etc.) to different data structures (e.g. a binary tree, a vector, etc.) we will find that these components will be naturally different for the different data structures, but they will be often reusable for different computations on the same data structure. As a result [16] proposes to use an object called *info* object whose aim is to provide information on the structure of the problem, including these two D&C components. The second group comprises the other components of the algorithm, i.e., the resolution of the base case and the combination of partial solutions, which are more strongly related to the concrete problem at hand to be solved, and they are encapsulated in a second object called the *body* object.

The C++ templates `Info` and `Body` in Fig. 1 describe the requirements and signatures for the info and body objects discussed above, respectively. Since an info object only deals with the problem input, its template depends on the datatype `T` of the input, but not on the type of the algorithm result, which we call here `S`, possibly with both types being the same. As expected, the info object has a method that returns a boolean specifying whether a problem is a base case or not. The non-base case decomposition is split in two methods: `num_children`, which specifies the number of subproblems identified, and `child`, which given an integer `i` between 0 and `num_children`−1 and the current problem `t`, returns an object that holds the `i`-th subproblem of `t`. This design was chosen so that when a non-base case is split, each subtask can be in charge of building its subproblem from the parent. The info object must derive from a provided class `Arity<N>` that is parameterized by the number `N` of subproblems in which a non-base case can be subdivided, which we call the arity of the problem. When `N` is a fixed value known in advance, `Arity<N>` provides the `num_children` method. When the arity is variable or unknown, the argument for `Arity` must be the predefined variable `UNKNOWN` and the user is responsible for implementing a proper method for `num_children`.

Regarding the `Body` object, it has the expected method `base` to solve a base case, and `post`, which combines the solutions of the subproblems of a non-base case (provided by means of a pointer in order to facilitate the support for variable numbers of children) into a single one. This latter method also receives the parent problem in case it has information required to compute the global solution that is not found in the children solutions, a situation we have found to be very common. Finally, the body also has a method `pre` that is invoked on the problem object before any processing, or even checking whether it is a base case, is performed on it. This method is motivated by the observation that in some algorithms it is useful to perform some processing on the input before considering it for the first time in the D&C algorithm. The `parallel_recursion` library provides a utility class template `EmptyBody<T, S>` from which body object classes can be derived, which provides empty definitions of all the body object methods, so that the users does not need to define those that are not required.

Besides the input problem, and the info and the body objects, this skeleton supports a fourth optional argument, called the partitioner, that controls the parallelism depending on its data type. Three classes of partitioners are supported. The `simple_partitioner` just runs a new parallel task for each child identified in any level of subdivision of the recursive processing of the D&C algorithm. The `auto_partitioner` is a smarter partitioner that tries to launch just enough parallel tasks to keep all the cores busy and allow them to balance their load by means of a work-stealing mechanism that is automatically provided by the underlying Intel TBB library [37], which `parallel_recursion` uses to define and run its parallel tasks. Finally, there is a `custom_partitioner` class that must implement a method `do_parallel(const T& t)` that returns a boolean specifying whether the children of the problem `t` must be processed in parallel, if true, or sequentially, otherwise.

Figure 2 illustrates the parallelization, using this skeleton, of the `treeadd` benchmark from the Olden

```
 1 struct TreeAddInfo: public Arity<2> {
 2   bool is_base(const tree_t *t) const
 3   { return t−>level == 1; }
 4
 5   tree_t *child(int i, const tree_t *t) const
 6   { return t−>child[i]; }
 7 };
 8
 9 struct TreeAddBody: public EmptyBody<tree_t *,int>{
10   int base(tree_t * t) { return t−>val; }
11
12   int post(tree_t * t, int *r)
13   { return r[0] + r[1] + t−>val; }
14 };
15 ...
16 int r = parallel_recursion<int> (root, TreeAddInfo(),
        TreeAddBody(), auto_partitioner());
```

Fig. 2 Reduction on a binary tree using `parallel_recursion`

benchmark suite [38], which adds the values in all the nodes of a binary tree. The arity of the D&C algorithm is 2, since every decomposable node will have two children, and this is reflected in the definition of the info object in line 1. The problem inputs are pointers to tree nodes (`tree_t *`). Each node has a value `val`, its `level` in the tree and an array of two pointers to children called `child`. The base case of the recursion are the nodes at level 1 (lines 2-3), which just return the value they store (line 10). Getting the `i`-th child of a decomposable node just involves returning the `i`-th element of its `child` array (lines 5-6), while reducing the values computed by the two children subtasks of a node with the node value itself involves adding these three values (lines 12-13). The usage of an algorithm template with arguments that are objects whose template classes are available to the compiler allows the inlining of the required methods in the code generation and a large degree of optimization. The result is that users do not need a separate definition of the algorithm steps for the sequential and the parallel cases. Rather, the skeleton is able to internally build separate high-performance parallel and sequential components from this specification, achieving a performance similar, and often even better than that of more burdensome approaches such as the native TBB algorithm templates, or standard compiler directives such as OpenMP [16].

4 Supporting divide-and-conquer in hybrid memory systems

The presence of a distributed memory, and even further, a hybrid distributed-shared memory system, noticeably complicates the implementation of a D&C parallel algorithm. The most important consideration is probably the distribution of the input problem on the distributed-memory nodes. In this regard we have three possible situations, all of which should be efficiently supported by a skeleton for maximum generality. The first one is that the input is replicated in all the nodes that participate in the computation. In this situation no initial data distribution is needed and all the nodes will work in parallel on their local copies, taking care that each one of them solves a different portion of the problem.

Another common situation is that the input is located in a single source node. Depending on the relative cost of the broadcast of the input to all the nodes and the problem subdivision component of the D&C algorithm we may choose between two possibilities. If the broadcast is cheaper, the algorithm should replicate the input in all the nodes by means of a broadcast and the proceed as in the situation when the input is replicated. Most often however the best policy will be to decompose the input problem until at least one subproblem per each participating node is obtained, and send to each node its subproblem(s).

The last possibility is that the data structure that represents the initial problem is already distributed among the participating nodes, possibly with some partial replication (for example, the top part in the case of a tree). In this case, the distribution stage can be skipped and each node can just work on its local portion.

In all the situations the parallelism within each node should be exploited to the fullest. Besides, this should be usually done using a shared-memory (threaded) strategy in order to facilitate load balancing and avoid message passing between its parallel tasks, rather exploiting the fast communication and synchronization facilities enabled by shared memory. The usage of multiple processes per node should be also supported, as in some applications this may perform better than a purely threaded approach. For this reason during the rest of our explanation we will refer to processes rather than to nodes when talking about the executing entities that have distributed memory. Relatedly, if the application were run using a single process, the skeleton should automatically only rely on a threaded strategy for its parallelization. Also, the skeleton should be able to support any arbitrary data types, and hide the details of interprocess communication as much as possible. Finally, regarding the result of the D&C algorithm, users should be able to choose between obtaining it only in the source process, letting it distributed on the processes that participated in the computation, or getting it replicated in all the processes.

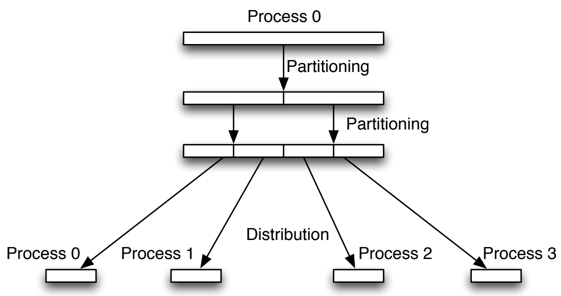
Our `dparallel_recursion` skeleton was designed having all these requirements in mind. Let us now discuss its syntax and functionality, followed by some implementation details.

4.1 Syntax and functionality

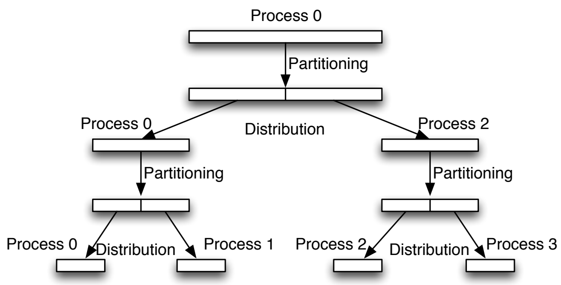
Since the abstract nature of a D&C parallel algorithm is the same no matter the kind of system where it is executed, we wanted our skeleton to experience minimum changes in order to adapt it to hybrid memory systems. In fact, its syntax only differs in three points from the one described in Section 3. The first difference is that the info object class must derive from the class template `DInfo<T, N>`, where `T` is the type of the input problem and `N` is its arity, i.e., the number of subproblems of a non-base case, or `UNKNOWN` when it is variable or not known in advance. The constructor of this class admits an optional integer that indicates the minimum number of tasks in which the user wants to partition the work in each process when the `auto_partitioner` is used. This improvement was motivated by our observation of the most common requirements for the execution of this kind of algorithms. The most important property of the `DInfo` objects is, however, that they store the information on the distribution of the input of the algorithm. This information is stored in the object after it has been used in the first `dparallel_recursion` invocation on a given input, which will either distribute that input or learn that it is already distributed, based on user-defined flags that are described below. The availability of this information in the `DInfo` object is useful because once a given data structure is distributed using our algorithm template, other D&C algorithms can be directly applied to the same input using our skeleton and allowing it to optimally exploit the actual data distribution, just by providing the same `DInfo` object. The result is maximum performance with minimum programmer effort.

The second change is that the skeleton allows a fifth optional argument that is a bitset of flags used to configure its behavior. The large variety of behaviors supported by the skeleton is now explained through the description of some of the available flags:

– `DefaultBehavior` implements the behavior applied when no bitset is provided. In this configuration the skeleton assumes that the input is only in the process with id or rank 0 (called source process), from which it must be partitioned, and where the only copy of the result of the algorithm will be located when the computation finishes.
– `ReplicatedInput` informs that the input problem is replicated in all the processes. The skeleton partitions the problem locally in each process making sure that each process works on a different subproblem once a given level of subdivision of the initial common parent problem is reached.
– `DistributedInput` allows to apply the skeleton to distributed data structures that have not been created using our skeleton, and thus, for which the `DInfo` object contains no distribution information. Namely, this flag reports that the input is already distributed among the processes, and the portion resident in each process is the input provided to the skeleton by that process. Notice that the existence of a pre-distributed input in which each process has an independent portion implies that there is a not a top-level single element from which to obtain every level of decomposition of the problem. Therefore this situation requires that the `post` operation that combines the results of the subproblems of a given input either does not use this input for the reduction or accepts one that is default-constructed by the skeleton to complete the reduction in the upper levels.
– `ReplicateInput` indicates that the input is only in the source process and it requests that instead of partitioning the input and sending a chunk to each one of the other processes, the input is replicated in all the processes and the algorithm then proceeds as in the `ReplicatedInput` case.
– `ReplicateOutput` affects the placement of the result. Instead of obtaining the final result only in the source process, a copy of it is obtained in all the processes.
– `DistributedOutput` informs the skeleton that there is no need to gather or replicate the output. Each process will simply keep its portion of the result.
– `GatherInput` controls the behavior of the skeleton with respect to the input problem after the D&C algorithm execution. By default the skeleton only collects the result of the reduction of the algorithm, that is, the value returned by the `post` method of the body object. This flag requests that the skeleton also gathers the input problem in the source process (or all the processes, if `ReplicateOutput` is also active). The most relevant situation when this is interesting is when the D&C algorithm modifies the initial input problem. This can happen in any or all the methods of the body object, as one can see that the model for these methods in Fig. 1 uses a non-`const` reference to the user problem, allowing to change it. Users are nevertheless free to use, and in fact should use, a `const` reference when the input is not going to be modified.
– `PrioritizeDM` asks to prioritize the distribution and reduction on distributed memory (DM) rather than



(a) Prioritizing partitioning



(b) Prioritizing distribution

Fig. 3 Partitioning strategies supported by `dparallel_recursion`, assuming 4 processes.

on shared memory. By default the source process partitions the problem until there are subproblems for all the processes, then gives these problems to the other processes to be solved, solves its own portion, and finally gathers all the sub-results to compute the final one. Figure 3(a) represents the partitioning stage of this strategy. When `PrioritizeDM` is requested, the source process follows the strategy depicted in Fig. 3(b), which sends subproblems as soon as possible to other processes, and all the processes that have sub-problems continue partitioning them in parallel and sending subproblems to other processes until all of them have work to do. The reduction stage follows exactly the reverse order.

Other flags express potential optimizations of different kinds. For example, some of them indicate that either the input or the result should be communicated by means of collective gather/scatter operations rather than point to point messages. Others help with the balancing of the distribution of work. Namely, the `Balance` flag balances the number of subproblems per process, while `UseCost` balances the computational cost of the problems assigned to each process. This latter functionality requires a user-provided function to estimate such cost.

The third change has nothing to do with the distributed nature of the new skeleton, but with our observations on D&C algorithms. Namely, we found that sometimes it is useful to perform some computations on a problem before partitioning it in subproblems, but not when it is a base case. Since the `pre` method of the body objects is always run on a problem, regardless of whether it is a base or not, it can perform these tasks, but at the cost of checking before whether the problem is a base case, which is something the skeleton has also to do anyway. As a result, better performance and programmability can be achieved by allowing a new method in bodies that is run only before problem subdivisions. We call this optional used-provided method `pre_rec`. An empty implementation is provided by `EmptyBody`, so that it need not be defined if it is not useful.

Figure 4 shows how these new features work together in the `treeadd` benchmark used as example in Sect. 3. Unlike Fig. 2, this code includes not only the reduction but also the construction of the tree using our skeleton, as it also constitutes a D&C algorithm. This allows to illustrate two uses of the skeleton and, furthermore, the creation and reuse in different invocations of distributed data structures using our library. In addition, the solution is very efficient, as not only is the tree built in a parallel and distributed fashion, but it also enables the second D&C algorithm to begin to work locally on its portion of the distributed structure in each process without initial communications.

The only value used by this benchmark to allocate the tree is its number of levels, and in fact it is the only value required by the constructor of the tree nodes. Also, each node in the tree, shown in lines 1-9, stores its level in the tree (variable `nlevel`), the leaves being at level 1, the value `val` to be added, and pointers to its children. The code in Fig. 4 assumes that the number of levels of the desired tree is available in all the nodes in the variable `NumLevelsTree`. This allows to build a replicated root for the tree in all the nodes in line 36. It deserves to be mentioned that while the constructor of a tree node `tree_t` in the sequential version triggers the recursive allocation of all its subtree, in our skeleton-based version each node allocation (lines 5-8) only creates one node. The reason is that it is the responsibility of `dparallel_recursion` to perform and parallelize the D&C allocation process, thus filling in the appropriate pointers to children. This task is performed by the skeleton invocation in line 38, which specifies that its input is replicated across the processes and the result will be obtained in a distributed fashion. As we can see in the figure, the `TreeInfo` class that describes the partitioning of the problem is identical to the `TreeAddInfo` class used in Fig. 2 with the exception that it derives from `DInfo<tree_t *, 2>` instead of `Arity<2>`. In the recursive creation of the tree, non-base nodes will fill in their `child` components with pointers to nodes of the

```
1  struct tree_t {
2    int val, nlevel;
3    tree_t *child[2];
4
5    tree(int lvl) : nlevel(lvl) {
6      val = ...;
7      child[0] = child[1] = nullptr;
8    }
9  };
10
11 struct TreeInfo: public DInfo<tree_t *, 2> {
12   bool is_base(const tree_t *t) const
13   { return t−>nlevel == 1; }
14
15   tree_t *child(int i, const tree_t *t) const {
16     return t−>child[i];
17   }
18 };
19
20 struct ParAllocBody : EmptyBody<tree_t *, void> {
21   void pre_rec(tree_t *t) {
22     t−>child[0] = new tree_t(t−>nlevel − 1);
23     t−>child[1] = new tree_t(t−>nlevel − 1);
24   }
25 };
26
27 struct TreeAddBody : EmptyBody<tree_t *, int> {
28   int base(tree_t *t) { return t−>val; }
29
30   int post(tree_t *t, int *r)
31   { return t−>val + r[0] + r[1]; }
32 };
33
34 TreeInfo tree_info;
35
36 tree_t *root = new tree_t(NumLevelsTree);
37
38 dparallel_recursion<void>(root, tree_info,
          ParAllocBody(), auto_partitioner(),
         ReplicatedInput | DistributedOutput);
39 ...
40 int r = dparallel_recursion<int>(root, tree_info,
         TreeAddBody(), auto_partitioner());
```

Fig. 4 Main elements of a `treeadd` implementation based on `dparallel_recursion`.

immediately lower level. This is achieved in the `pre_rec` method of the class `ParAllocBody` used by the skeleton. Notice how in this problem the body object does not generate a separate output, but rather modifies the input of the algorithm. As a result, and since the resulting tree will be available through the `root` variable in each process, the return type of this `dparallel_recursion` invocation is `void`, which is specified as the only template argument to the invocation in line 38. Therefore in this case the skeleton operates as a procedure that builds the tree exploiting the property mentioned above that it can modify its input. This is in contrast with the invocation in line 40, where the `int` template argument to the invocation informs that the skeleton will return

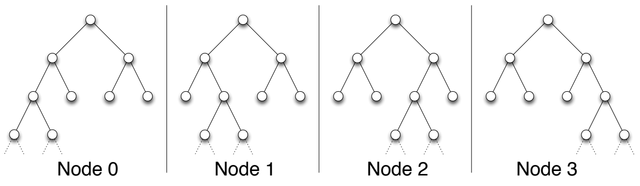


Fig. 5 Shape of the local tree built by the code in Figure 4 in each process, assuming 4 processes.

an integer, which is stored in the destination variable `r`. Of course, the return type must be compatible with the operations and types specified in the body object of the associated invocation.

When the root of a problem is present in a single process, unless the user requests to prioritize the distribution across processes over the partitioning (`PrioritizeDM`), this process recursively subdivides the problem until there is at least one subproblem for each process, then distributes the work to the other processes, and later works on the subproblems it has assigned to itself. When the root is replicated, however, all the processes subdivide in parallel the original problem until there is at least one subproblem for each process, then choose the subproblems they keep for themselves, and continue working only on them. As a result of this policy, in our example the local tree built in each process replicates the top levels of the tree, but just below the level where there are as many or more vertices than computing processes, each process only has one or some of the branches, as Fig. 5 shows. Our skeleton is totally general, so any number of computing processes is supported. As a result, in some situations there can be some imbalance, that is, some processes can keep more low level portions of the distributed data structures than others. In any case, users do not need to be aware of these details. They just need to know that the information on the concrete partitioning used is stored in the info object, called `tree_info` in Fig. 4, and that using it in subsequent invocations of `dparallel_recursion` will allow the skeleton to operate correctly and optimally on the data structure. This way, the usage of this object in the invocation in line 40 to perform the reduction on the values of the tree ensures that each process will correctly identify the portions of the structure it owns.

Notice that the invocation in line 40 does not use the `DistributedInput` flag for two reasons. The first one is that `tree_info` already contains all the information on the distribution of the input, making the flag useless. The second and most important one is that, as we explained above, this flag is actually not suited for this situation, its purpose being to allow the application of the algorithm template to data structures that have not been distributed using `dparallel_recursion`. An example input for which this flag would be appropriate is a distributed array where each process has a separate portion of the global array following some strategy predetermined by the programmer.

It deserves to be mentioned that the design of this algorithm template allows to use it to provide the functionality of other very common skeletons. For example the map operation that applies in parallel some function to the elements of a list giving place to another list with the results can be naturally implemented by partitioning the input list until there are enough chunks to exploit all the parallelism available, processing these chunks as base cases, and merging the resulting lists in the `post` operation of the body object. In the case of reduce, which reduces to a single value several elements using an associative operator, `post` would perform the reduction of the partial reductions from several chunks. In fact this implementation for reduce follows a strategy similar to the one used by the `parallel_reduce` template function of the Intel TBB [37], although ours is more general, the main advantages of `dparallel_recursion` being the support of distributed memory and arbitrary problem arities.

Finally, while the `dparallel_recursion` skeleton is the kernel of our library, it also includes some items to facilitate its use. The main ones are range classes that provide automatic partitioning, shallow arrays that allow to partition arrays without replicating their data, and macros and function templates to implement parallel loops on top of our skeleton using a very simple syntax. The framework also provides `parallel_recursion`, our skeleton for shared-memory parallelization, as well as similar utilities built on top of it. Also, while they are not part of the public API, it is very easy to access internal functions that provide communications between processes on top of MPI using a simple syntax similar to that of Boost.MPI [20], and more importantly, applying the optimizations described in Sect. 4.2, which was in fact the reason for their development.

4.2 Implementation and optimizations

As shown in Fig. 6, our framework, which is the area enclosed in the thicker black line, relies on three external libraries. Both the `dparallel_recursion` skeleton proposed in this paper, and the `parallel_recursion` algorithm template introduced in [16] rely on Intel TBB [37] for the shared-memory parallelism, using its low level API to build and synchronize tasks. This is a C++ library for parallel programming on multi-core processors based on tasks. TBB provides mechanisms to define, order (i.e., specify dependences) and synchronize

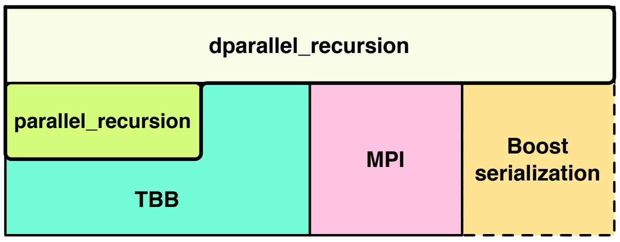


Fig. 6 Library dependences of `dparallel_recursion`

tasks, letting the runtime of the library in charge of the low level details such as managing thread pools, enforcing the dependences declared by the user, or stealing tasks between threads for the sake of load balancing. This library was preferred over other alternatives such as OpenMP because it provides better control and according to studies like [36] its task creation, scheduling and load balancing mechanisms seem to be more sophisticated and optimized than those of OpenMP. As an added benefit, a compiler without support for OpenMP can be used to compile our skeleton[1], something that is not trivial and requires special measures for libraries that rely on OpenMP [7]. The shortcoming of the Intel TBB library with respect to OpenMP is that even if the user relies on the algorithm templates it provides, which largely simplify its usage compared with its low level API, its programming costs are much higher than those of compiler directives [16]. In our case, all this complexity is hidden inside our library. The distributed-memory parallelism is supported by means of the MPI standard. A final dependence of our skeleton is Boost [6], the well-known collection of C++ peer-reviewed libraries, which is mainly used to serialize data to be transmitted in the MPI messages. This library is delimited by a dashed line in Fig. 6 because while its headers are always required, it only needs to be linked to an application based on our skeleton when its most advanced features are used.

The implementation follows five stages. First, the problem is decomposed until there is at least one subproblem for each MPI process or the load balancing criteria set by the user are met. This stage is skipped if the input is distributed, since this implies each process has already a subproblem. This top level decomposition is stored in the `DInfo` object in case the problem is used in further skeleton invocations. Then, the subproblems are distributed among the processes, except if the input was replicated or distributed, in whose case each process directly takes care of its subproblems. The

[1] One might think that in 2016 every major compiler distribution should support OpenMP, but as a representative example, during the development of this work we found that the compilers of the standard development environment for the current version of Mac OS X do not support OpenMP.

third stage processes in each process the problem(s) assigned to it. This stage is very much the algorithm template presented in [16] except for the new `pre_rec` method and the inclusion of new optimizations enabled by C++11. The fourth stage gathers the partial results in the source process, and the fifth one performs the final reduction until a single result is obtained. Of course, these two latter stages are skipped if the user requested to keep the output distributed. Also, if `PrioritizeDM` was requested, the first two stages happen in an interleaved way until all processes get work, and the same happens with the two last stages until a single result is obtained.

The implementation contains numerous optimizations that make it very competitive with hand-optimized codes. First, it extensively relies on C++ template metaprogramming so that polymorphism is efficiently resolved at compile time rather than at runtime. Second, we tried to exploit as much as possible the new optimizations enabled by the C++11 standard, mainly those associated to rvalue references and move constructors and assignments. Third, every relevant step inside the library has been parallelized. The only exception are MPI messages, which can be sent or received from different threads in a process, but never simultaneously from several threads. The main reason for this design was ease of installation and portability, since important MPI distributions are not compiled by default to support this possibility, while others do not even support this feature in popular environments [12]. Also, the implementation exploits MPI collective communication primitives whenever it identifies it is possible and safe to do so.

The parallelization pattern followed in the initial decomposition proceeds by levels, generating a new level of subproblems out of the most recently generated one by decomposing all its elements in parallel. This pattern was chosen due to the need to (a) generate a minimum number of subproblems before proceeding to their distribution among the processes and (b) try to make these subproblems as similar as possible in terms of size to balance work. Since by default the skeleton has no information on the cost of each subproblem, it follows the heuristic of distributing subproblems obtained at the same level of decomposition. The algorithm template stops the initial decomposition either at the first level with enough problems to feed all the processes, or when the conditions set by the user for the load balance by means of the `Balance` or `UseCost` flags commented in Sect. 4.1 are met, or when further problem subdivision is possible.

As explained before, the MPI calls cannot be made simultaneously from different threads. Despite this fact, the communication stages can also exploit parallelism because the skeleton tries to parallelize the (de)serialization process of the data involved in the communications when such process is needed. The parallelization is both among different (de)serialization tasks as well as with the active communication task in each moment.

Typically the most expensive part of the execution is the stage in which each process solves its subproblem(s), which has been parallelized following a recursive pattern. Namely, whenever a task partitions a problem, it checks whether there are enough parallel tasks in the system depending on the partitioner provided by the user (see Sect. 3). If this is the case, the children are processed using a purely sequential implementation of the D&C algorithm. For example, this version makes no further checks related to parallelism. Otherwise, the task generates an independent parallel task for the processing of each subproblem, launches them to execution, and awaits their completion. This wait is not active; rather the task simply remains in the stack of the thread that run it until all its children tasks finish, which allows the thread to return to it. At that point, the task performs the reduction of the results using the `post` method and finishes. Finally, the last stage naturally follows the same parallelization pattern as the first one, but in the reverse order, that is, bottom-up.

An issue that can play an important role in performance and where our library provides very simple and effective mechanisms to improve the performance is the data serialization. Our framework implements three serialization policies that the user can choose from. First, arithmetic types or types marked as bitwise serializable by means of the `BOOST_IS_BITWISE_SERIALIZABLE` macro can be represented just by the consecutive sequence of bytes that constitutes them. Thus they require no actual serialization and they are directly sent from, or received in, their original storage in memory by our algorithm template. Otherwise, the user has to provide functions to serialize/deserialize the object in/from an archive provided by the skeleton using the API supported by the Boost serialization library. Relying on this library is very convenient given its degree of optimization and the facilities it provides for the (de)serialization with minimum effort of pointers, arrays, STL collections, etc. The user can choose between two possibilities for the transmission of non-bitwise serializable data types. If she marks the type with the macro `TRANSMIT_BY_CHUNKS`, each interaction of the user (de)serialization function(s) with the archive provided by the skeleton, i.e., the (de)serialization of each individual component of the object to transmit, will give place to a separate message that will transmit only this element. In its turn, whenever any of these

Table 1 Benchmarks used.

Name	Arity	Assoc	Imbalance	Input	Output
`fib`	2	Yes	yes	scalar	scalar
`quicksort`	2	-	yes	array	array
`nqueens`	var	Yes	yes	board	scalar
`strassen`	7	No	yes	arrays	array
`treeadd`	2	Yes	no	tree	scalar
`tsp`	2	No	no	tree	cycle
`barnes hut`	var	Yes	Yes	array	array
`ep`	var	Yes	No	range	histogram

chunks is bitwise serializable, it will be directly sent/received from/in its existing location avoiding any copy or translation cost. If the type is not labeled with this macro, the skeleton will serialize all the components of the object in a single buffer and transmit it in a single message. Our implementation has been made in such a way that serialization functions are written in exactly the same way for both kinds of serialization, making the process totally oblivious to users but for the application of the macro to the data type.

Choosing the best serialization policy can be critical for performance. A good example is a variable-length vector whose components are an integer `sz` with its size and a pointer `ptr` to the elements it stores. This type is not bitwise serializable, as the bits of its two data members are not enough to represent all the data associated to it. As a result, the user has to provide functions that serialize/deserialize in/from an archive the size `sz` and the array of `sz` elements pointed by `ptr`. By default `dparallel_recursion` will copy these two components to/from a single temporary buffer and send/receive them in a single message. Nevertheless, if the type is marked with `TRANSMIT_BY_CHUNKS`, `sz` will be transmitted in one message and the array of elements in another one. If the elements stored in the vector are bitwise serializable, there will be no need for any temporary allocation or copy of data, neither in the sender nor in the receiver.

Other simple user-level optimization enabled by our library are the flags related to collective communications mentioned in Sect. 4.1 and the usage of `DInfo` objects belonging to the subclass `BufferedDInfo`. These objects optimize memory usage by keeping the buffers used during the communication between processes to avoid their repetitive allocation and deallocation. Also, they allow the user to provide those buffers, so that if an existing data structure can used as temporary storage, even the allocation, and sometimes more importantly, the extra memory footprint, is avoided. In a similar fashion, the skeleton has also mechanisms to let the programmer specify the location of the object that will hold the final result in order to avoid the creation of temporaries as well as unneeded copies or movements.

5 Evaluation

In this section our skeleton is evaluated both in terms of performance and programmability using the eigth benchmarks described in Table 1. The table provides, for each benchmark, its arity (number of subproblems in which each problem can be divided), whether the combination of the results of the subproblems is associative or not or not needed, whether there is imbalance between children problems of the same parent, and the kind of input and output of the algorithm. Let us now briefly discuss each one of these programs.

The `fib` benchmark recursively computes a Fibonacci number $fib(i) = fib(i-2) + fib(i-1)$. Although this is an inefficient method to compute this value, this benchmark is widely used in academia (e.g. [30,42]) as an example of D&C algorithm with imbalanced tasks. Furthermore, let us notice that since skeletons execute the same computational blocks as serial or manually parallelized versions of the same code, adding the elements needed to connect and run them in parallel, it is in simple benchmarks such as this one or `treeadd`, described in the preceding sections, where skeletons are expected to more clearly show their overheads. Our test assumes that the input is replicated in all the processes and the result is obtained only in one. Notice that since the input is a scalar, if it were not initially replicated, it would be trivial to replicate it with little cost using a MPI broadcast operation, or just using the `ReplicateInput` flag in the case of our skeleton.

Our second example, `quicksort`, sorts a vector of integers initially located in a single process using the quicksort algorithm and leaves the result distributed among the participating processes. The imbalance of the tasks of this algorithm is highly variable, as depending on the pivot (randomly) chosen for the partitioning of an array, the resulting children tasks can be heavily imbalanced. When a subproblem reaches a size below 10^4 elements, our implementations resort to the `std::quicksort` function provided by the standard library to complete the sorting process.

The main interest of the third algorithm, `nqueens`, which computes the number of solutions to the N Queens problem, lays on the variable number of children of each subproblem. Just as in `fib`, the input is assumed to be replicated and the result is obtained only in one process. Again, since the board object is bitwise serializable, its replication in MPI or with `ReplicateInput` would be trivial and inexpensive.

The fourth benchmark is Strassen's algorithm for matrix multiplication, which has complexity $O(N^{2.8074})$ compared to the $O(N^3)$ of the traditional algorithm. Our implementations begin with the input matrices in a single process and gather in it the final result. The tasks of this algorithm present very little imbalance, but the fact that its arity is 7 makes recursive overdecomposition necessary to optimally exploit the number of cores, as it is typically even, and often a power of 2. When the decomposition reaches matrices of size 256×256 our programs resorts to a standard matrix product algorithm provided by uBLAS [44].

The next two applications have been taken from the Olden benchmarks suite [38]. The first one is `treeadd`, which has been used as example in the preceding sections. The second one, `tsp`, solves the traveling salesman problem on a tree in which each node represents a city. As in `treeadd`, `tsp` contains two distributed D&C algorithms that are interrelated because the result of the first one (tree construction) is the input of the second one (traveling salesman problem resolution). For this reason the best implementation strategy for `tsp` is also to build the tree in a distributed fashion, so that the second D&C algorithm can proceed in parallel in the different processes without the need of messages to distribute the input. In our tests the result of `treeadd` is obtained in all the processes, while the one of `tsp` is obtained only in the source process.

The seventh benchmark is the Barnes-Hut n-body algorithm [3], which classifies the bodies in an octree of cells in order to reduce the computations. Namely, the octree agglomerates the bodies in hierarchical cells so that a single computation representing the whole cell suffices to compute the approximate impact of the bodies within the cell on bodies that are beyond a given distance threshold. Our implementation started from the Barnes-Hut code of the Lonestar suite [28], which only parallelized the computation of the forces. Our benchmark is more ambitious, as we also parallelized the update of the bodies due to those forces and the computation of the center and the diameter of the space where the simulation takes place.

The last benchmark is the ep application of the NAS Parallel Benchmarks [35], which generates independent gaussian random values using the Marsaglia polar method and then performs a reduction on them. This benchmark was chosen for two reasons. First, it illustrates the use of our skeleton on problems that can be easily expressed as a parallel loop with a reduction. Second, it is a well-known benchmark with standard optimized implementations with which to compare and where the optimal implementation is straightforward.

Five versions of each benchmark, in addition to the one based on our proposal, were developed for this evaluation. First, we built, or took from the existing suite, an optimized sequential baseline. Then, in order to compare with optimized codes that only rely on distributed memory communications, we developed MPI versions. Since the most widespread approach to exploit hybrid memory systems in HPC applications is the combination of MPI with OpenMP, the well-known standard for shared-memory parallelism, we also wrote versions that combine these two paradigms. The main purpose of the other two versions is to compare our proposal with other high level approaches that provide programmability advantages for D&C algorithms. As we will see in Sect. 5.1, the MPI-only implementations considerably lag behind the `dparallel_recursion` and the hybrid MPI/OpenMP versions for many benchmarks. Thus, comparing with any approach without support for multithreading and shared-memory parallelism would have been unfair. Also, as discussed in Sect. 2, we found no skeletons optimized for multi-core clusters that support the D&C pattern. This way, in the end we developed versions based on MPI combined with Cilk Plus [24] and with the newest D&C parallel skeleton we found [10]. This skeleton, which will be called `dac` in the following, follows a multi-threaded approach to parallelize D&C problems in shared-memory environments, and like ours. is a parallel template that uses C++11 features. Since several backends were developed for this skeleton, all of which provide similar performance in [10], our experiments use the backed based on Intel TBB [37] in order to maximize the similarity of the approaches compared. In the rest of this paper we use the term hybrid versions/codes/implementations to refer to those that combine MPI with a threaded approach, including our skeleton.

We actually developed many more than six versions of many benchmarks, because several parallelization strategies were tested for the codes in which the best one was not obvious, seeking the implementation with the best performance. For example we found that the best MPI-only implementation of `quicksort` followed the decomposition strategy in Fig. 3(b), while the best `dparallel_recursion` and hybrid implementations followed Fig. 3(a), which besides facilitates the use of `MPI_Scatterv` to optimize the data distribution. The final manually developed hybrid versions apply exactly the same optimizations and patterns of parallelization, which are equivalent to those of our skeleton, or sometimes better thanks to hand-made optimizations. Also, the parallelization was not restricted to the kernel of the D&C algorithms. Rather, it was applied to all the meaningful parts of the applications. For example, the deserialization of the cycles of cities built in the `tsp` problem can be accelerated with a parallel loop. Following with this code, even with this improvement, it is very important for performance to parallelize the deserialization process with the receipt of the cycles of cities that come from other processes in the reduction stage of the computation. The MPI + OpenMP implementations parallelized these portions of the code using OpenMP directives; the `dparallel_recursion` and MPI + `dac` versions resorted to the TBB facilities for this, and the codes based on MPI and Cilk Plus relied on Cilk tasks, including those generated by `_Cilk_for` loops. The Cilk Plus and OpenMP versions follow very similar schemes because `omp for` pragmas parallelize loops in a similar way to that of `_Cilk_for`, and the OpenMP tasking mechanism introduced in version 3.0 of the standard was used to parallelize recursive processes. This enables a style, which although based on directives, is similar to the one provided by Cilk Plus keywords. The schemes are only slightly different in that the parallel loops that contain nested parallelism based on tasks were parallelized in OpenMP by means of tasks from a single common ancestor in order to try to facilitate the load balancing of all the tasks involved in the parallel computation.

All our parallel versions are written to support any number of processes. In addition, the threaded versions, that is, all of them except the sequential and the MPI-only version, support any number of threads per process and allow to choose the number of tasks per thread. In several algorithms it is impossible to generate an exact number of tasks per thread unless those tasks correspond to different levels of decomposition of the initial problem, which could imply heavy imbalances between tasks. For this reason, the threaded versions are designed to generate all their sequential tasks at the same level of decomposition of the original problem. As a result, they stop the parallel partitioning and generation of tasks when they reach the first level of decomposition that allows to generate at least the number of tasks per thread requested by the user. This means that the actual number may be larger than the requested one.

Table 2 Experimental environment.

Feature	Value
Processors/node	2 x Intel Xeon E5-2680 v3
Family	Haswell
Frequency	2.5 GHz
#cores/CPU	12
Memory/node	64GB DDR4
#nodes	32
Network	Infiniband FDR
Compiler	g++ 6.1
OpenMP version	4.0
MPI	OpenMPI 1.10.2
TBB	4.4

5.1 Performance evaluation

Our experiments were performed in the Linux cluster described in Table 2, which consists of 32 nodes with 24 cores each, totaling 768 cores. The optimization level `O3` was used in all the compilations. Table 3 shows the relevant configuration parameters that describe the problem size of each benchmark tested, the runtime of the sequential execution and the number of processes per node that gave place to the shortest runtime of each application when using the 32 nodes available. The `treeadd` and `tsp` benchmarks contain two very different D&C kernels that are parallelized in our experiments: one that builds the tree (allocate) and another one that performs the computations (compute). While the compute kernels are interesting for clear reasons, we think that the allocate kernels also deserve attention because they illustrate how `dparallel_recursion` can build distributed data structures (in this case, with partial replication) that can be used in other D&C kernels, as our example based on `treeadd` in Section 4.1 showed. Also, the two allocation kernels differ among themselves, as the one in `treeadd` is very intensive on memory operations, with almost no computations, while the one in `tsp` contains several double-precision floating point operations, including non-trivial calculations such as logarithms.

In what follows, unless otherwise stated, in the executions that use several nodes, which are all the ones that involve more than 24 cores, the number of threads used by each process was fixed to $24/N$ where N is the number of processes per node shown in Table 3. In the executions using $c \leq 24$ cores, which always use a single node, the number of processes was set to $p = \lceil c/(24/N) \rceil$, with c/p threads each. It also deserves to be mentioned that the degree of variability observed in the runtimes was the normal one. This way, the standard deviations were below 1% of the average for the large runtimes and they were under 10% for the shortest runtimes, which are below 10 milliseconds for the allo-

Table 3 Problem sizes and common configuration for performance evaluation.

Name	Problem size	Seq. time	Procs/node
`fib`	54st Fibonacci number	390.79	1
`quicksort`	500 million 32-bits integers	50.38	1
`nqueens`	16×16 board	175.23	1
`strassen`	8192×8192 double-precision matrix	137.05	4
`treeadd` allocate	binary tree with 30 levels	25.99	2
`treeadd` compute	100 reps. of the tree reduction	415.76	2
`tsp` allocate	binary tree with 27 levels	29.60	2
`tsp` compute	traveling salesman problem	281.41	2
`barnes hut`	10 iter. 5×10^5 bodies in 3D space	232.13	2
`ep`	class D (2^{36} values)	2758.86	2

cate kernels of `treeadd` and `tsp` when using the whole cluster.

Several combinations of flags for those benchmarks for which the best combination was not obvious were tried in order to decide the best implementation of each algorithm. Figure 7 shows the results of these experiments, plotting the average speedup achieved by 16 executions with at least 4 subtasks per thread with respect to the optimized serial implementation for different numbers of nodes. Notice that all the combinations for `fib` use `ReplicatedInput` because, in order to evaluate different situations, the input scalar is assumed to be available in all the nodes for this experiment. Similarly, `DistributedOutput` appears in all the combinations for `quicksort` because in order to test different possibilities, our codes assume that the user wants the resulting vector distributed across the processes that participate in the computation. We must also note that the flags in Fig. 7(d) apply to the only stage of the algorithm that requires communications in our implementation, namely the update of the bodies with the previously computed forces. In our implementation all the bodies exist in all the processes so that each process can build the whole octree and the force computation stage can access any arbitrary body found in the octree. For this reason, the flags for this stage include `ReplicatedInput`. The flag `ReplicateOutput` must also be used, because the bodies must be replicated again in all the processes for the next iteration of the simulation.

In order to interpret the results we must remember that `Balance` just balances the number of subproblems per process, while `UseCost` balances the cost of the subproblems assigned to each process. This latter flag requires the user to provide a function to estimate this cost. Since `fib` has an arity 2, its number of subproblems is always a power of 2, and `Balance` does not have any influence on performance in Fig. 7(a). The `quicksort` kernel has also arity 2, and for this reason we skipped trying this flag in this benchmark. Nevertheless, both `fib` and `quicksort` are very imbalanced in the cost of their subproblems, and therefore `UseCost` can improve their performance. It must be mentioned that our template allows to configure several parameters related to the load balancing process, including the maximum time spent in it or the maximum imbalance allowed, measured as the ratio of additional subproblems/cost of the process with more load with respect to the process with less load. All our experiments use the default configuration, which allows a maximum imbalance of 20%. While cost-based load balancing is very positive for `fib`, its effects are not consistent in `quicksort`. The reason is that in this algorithm the initial partitioning stage is very expensive, and the imbalance can be very large even after many levels of subdivision. As a result, requesting to balance the load among the processes can force the threads of the source process to perform many levels of decomposition of the problem that would have been otherwise parallelized among the threads of all the processes. A second problem is that without `UseCost` the skeleton partitions the problem until there is at least a subproblem per each process, and then sends a single subvector to each process, but with `UseCost` there are several subproblems per process, each process in general receiving a different number of subproblems. This leads to many more messages, which results in additional performance degradation. This is the reason why the `Scatter` flag, which informs `dparallel_recursion` that the input is a vector that can be distributed by means of `MPI_Scatterv`, almost does not help when load balancing is not requested, but clearly improves the execution of the algorithm for most numbers of nodes when `UseCost` is applied. This latter combination (`DistributedOutput`|`Scatter`|`UseCost`) is the one with the best average performance, and thus the chosen one. Although `nqueens` is an imbalanced algorithm, we have not experimented with the application of `UseCost` to it because we do not know of a heuristic that allows

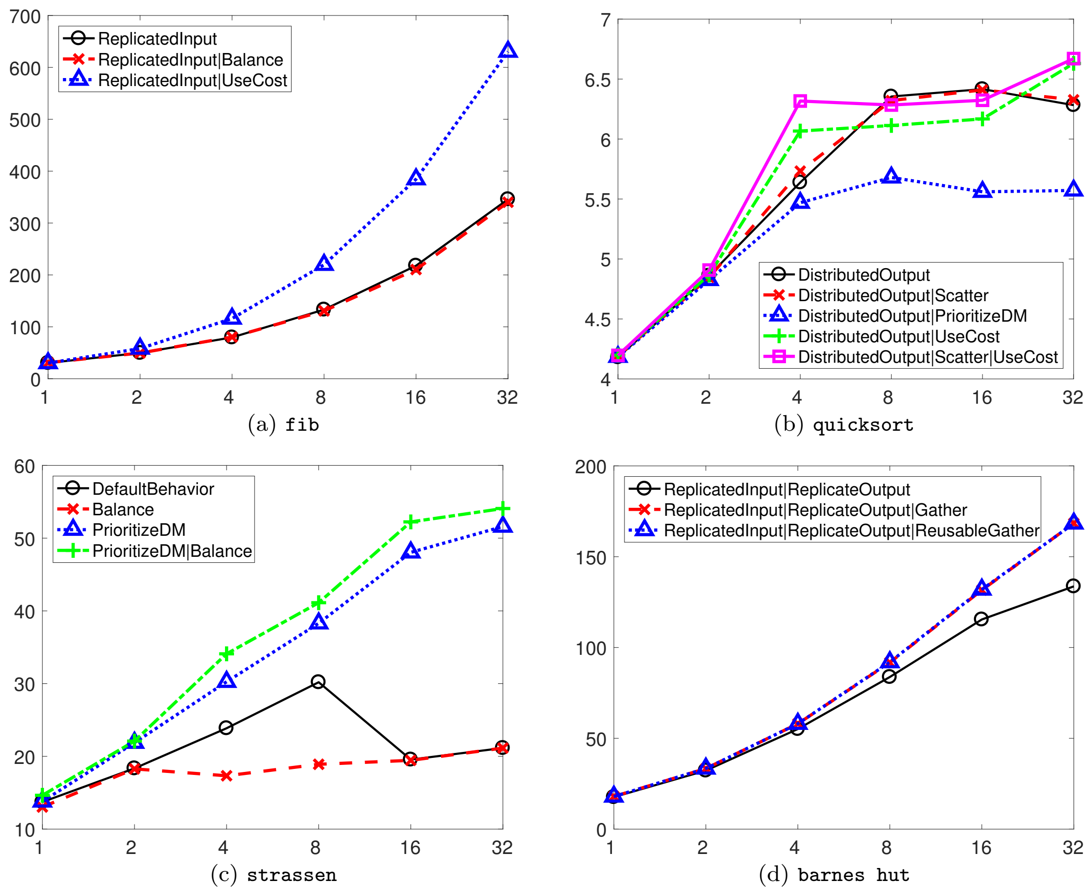


Fig. 7 Impact of different optimization flags on the speedup of `dparallel_recursion` over the sequential version as a function of the number of nodes used in the execution.

to estimate the cost of a subproblem for this benchmark.

As for `strassen`, this is an algorithm where the `PrioritizeDM` partitioning algorithm is clearly needed to obtain the best performance. Also, since the number of subproblems is always a power of 7, and thus not divisible by the number of processes used, which is a power of 2, the `Balance` flag is useful for this algorithm.

Regarding `barnes hut`, as explained before, by default communications take place by means of point-to-point messages. Since all the bodies of this application are located in a consecutive vector, collective communications based on the `MPI_Gatherv` family can speedup the execution if the programmer uses the `Gather` flag. In addition, since the number and relative position of the bodies assigned to each process within the vector remain constant during the execution, the communications that prepare the collective communication ensuring that every process knows how much to receive and send from/to any other process can be performed just once instead of in every iteration of the simulation. The user can provide this information to the skeleton by means of the `ReusableGather` flag. We can see in Fig. 7(d) that collective communications are increasingly important for `barnes hut` as the number of processes grows, while the `ReusableGather` optimization plays a minimal role. Notice that since `ReusableGather` implies a gather collective, its use makes unnecessary the specification of the `Gather` flag.

Table 4 shows the partitioner and the flags used for our `dparallel_recursion` experiments. The partitioners have been chosen so that they simplify the implementation of a program that allows the user to control the number of tasks per thread. Algorithms that basically parallelize loops are well served by a single level of subdivision in which each loop is divided in as many tasks as desired, which are considered base cases. This situation, found in `barnes hut`, is easily expressed with a `simple` partitioner. Algorithms that necessarily require a recursive subdivision in order to generate different numbers of subproblems are better served by an `automatic` partitioner, because it allows users to specify the number of tasks they want and then it computes and manages the number of subdivisions required to achieve the desired granularity. Finally, a problem

Table 4 Configuration of `dparallel_recursion` for performance evaluation.

Name	Partitioner	Flags
`fib`	automatic	ReplicatedInput|UseCost
`quicksort`	automatic	DistributedOutput|Scatter|UseCost
`nqueens`	custom	ReplicatedInput|Balance
`strassen`	automatic	PrioritizeDM|Balance
`treeadd` alloc	automatic	ReplicatedInput|DistributedOutput
`treeadd` comp	automatic	ReplicateOutput
`tsp` alloc	automatic	ReplicatedInput|DistributedOutput
`tsp` comp	automatic	DefaultBehavior
`barnes hut`	simple	ReplicatedInput|ReplicateOutput|ReusableGather
`ep`	-	- (dpr_pfor_reduce was used)

with variable arity in which several levels of subdivision may be needed to generate the desired number of tasks demands a more complex approach. For this reason `nqueens` is the only algorithm that relies on a `custom` partitioner. Regarding the flags, as explained before, some of them correspond to the assumptions made on the initial input conditions, such as whether the inputs are already available in all the processes. Other flags indicate the desired output conditions; for example whether the output must be distributed or replicated. The flags related to collective communications, partitioning strategy (`PrioritizeDM`) and load balancing have been chosen for performance reasons, as explained during the discussion of Fig. 7. The table shows that `ep`, rather than making a standard invocation to the `dparallel_recursion` function template, resorts to `dpr_pfor_reduce`. This is a macro provided by our framework that relies on our algorithmic skeleton to parallelize the very common pattern consisting in a parallel loop with a reduction, which is in fact the nature of the `ep` benchmark. The macro efficiently distributes the iterations and the reductions both across processes and threads.

Since the most comparable high-level solutions chosen, the `dac` skeleton [10] and Cilk Plus [24], do not natively support multiple processes, the performance comparison with them, shown in Fig. 8, uses a single process and a single node in our cluster. The figure shows the percentile speedup achieved by each implementation with respect to the sequential code when using the 24 cores available. In order to find this speedup, an exhaustive search allowing 2^i, with $0 \leq i \leq 30$, tasks per thread was made. Each execution with each different number of subtasks was repeated 4 times, and the minimum time of the series for each benchmark was taken. All the benchmarks were initially implemented following the scheme illustrated in our examples in Fig. 4, meaning that the D&C algorithm was only written once using the tool of choice, with the recursion finishing in the actual base case. While this policy perfectly fitted our skeleton, the other high level solutions suffered from reduced performance in algorithms with many levels of recursion and light computations such as `fib` or `treeadd`. In the case of the `dac` skeleton, strong slowdowns with respect to the serial version were observed in several cases, as the negative bars in Fig. 8 show. As a result, we optimized the `dac` and Cilk Plus versions of the algorithms that require a recursive decomposition to achieve parallelism, which are all of them except `barnes hut` and `ep`, by writing the D&C algorithm in two stages. Namely, in these versions the execution of each algorithm begins at the top level with an implementation parallelized with `dac` or Cilk Plus that stops its recursion not in the actual base case of the algorithm, but in one in which we want to switch from the parallel recursive decomposition to a sequential one. At that level, the resolution of the problem is entrusted to a serial implementation of the algorithm. This is in fact the usual strategy followed to make manual optimized parallel implementations of this kind of algorithms [42].

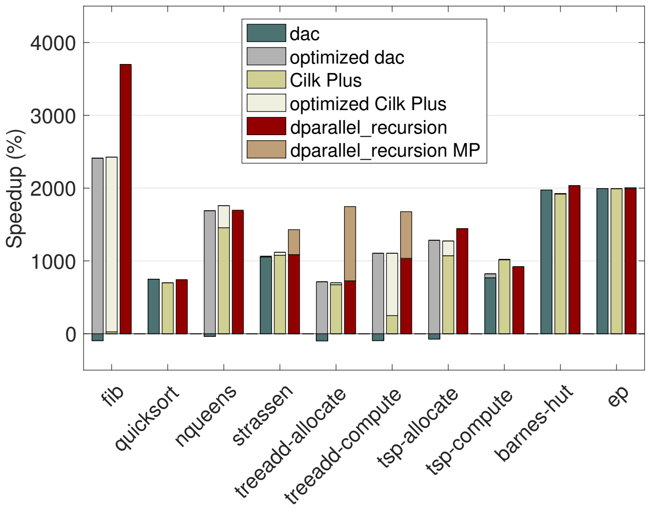


Fig. 8 Performance comparison with other high level approaches in a single node (24 cores).

It deserves to be mentioned that while some algorithms favor the development of the two stages as separate entities, mainly for performance reasons, in others, particularly in the most complex ones, it is possible to reuse most of the code, just choosing a different execution path depending on the level of decomposition of the problem. The improved versions, labeled as `optimized` in Fig. 8, allowed `dac` and Cilk Plus to reach a performance similar to that of `dparallel_recursion` in all the benchmarks except `fib`. In fact, we can notice that the speedup of `dparallel_recursion` for this algorithm is super-linear, reaching a value of 39 on 24 cores. The main reason is 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 other versions. This way, the sequential computation of the 54th Fibonacci number using our algorithm template is 80.3% faster than the sequential implementation.

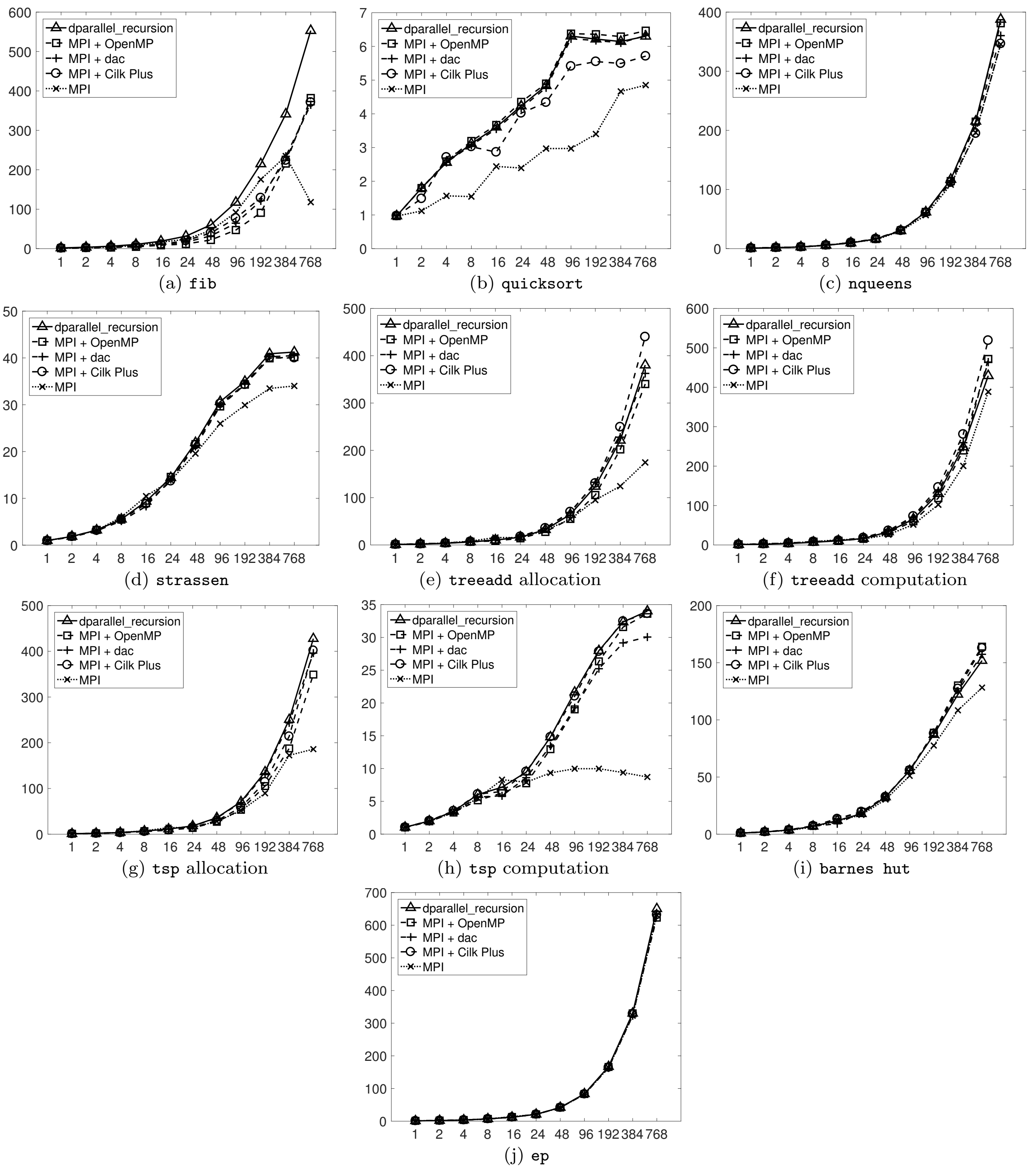
On average, `dparallel_recursion` was 6.7% and 4.7% faster than the optimized `dac` and Cilk Plus versions, respectively, or 2.6% and 0.5% if `fib` is not considered because of the favorable treatment that the compiler provides to the version generated by our skeleton. These values, as well as all the other averages of ratios and speedups in this paper have been computed as geometric means [14]. Since our skeleton allows to exploit multi-process parallelism with very little effort, a final piece of data shown in Fig. 8 is the speedup that it can achieve on the same system when using the number of processes indicated in Table 3, which is labeled as `dparallel_recursion MP` (for multi-process) in the figure. The ability to exploit multi-process parallelism allows our proposal to be on average 24.1% and 21.7% faster than `dac` and Cilk Plus, respectively, or 21.4% and 18.9% without `fib`, respectively.

Figure 9 shows the speedup of the parallel versions of each D&C algorithm with respect to the sequential time for a varying number of cores. The versions that combine MPI with OpenMP, `dac` and Cilk Plus apply the optimized recursive implementation motivated in the previous experiment. The configuration of the runs was the one explained at the beginning of this Section, based on the number of processes per node reflected in Table 3. The speedups for the MPI versions correspond to the average of 12 executions. The multithreaded versions were also run 12 times, but in their case the tests were performed generating at least one, two, or four tasks per thread, and repeating the execution with each number of subtasks 4 times. The figure plots for each version and number of cores the average speedup achieved by the degree of partitioning that offered the best average performance for that configuration.

The reason for the large advantage of our algorithm template with respect to the other approaches in `fib` has already been discussed. The behavior of the MPI implementation for 768 cores is due to the cost of the balancing algorithm, which is more expensive as the number of processes among which to subdivide the work grows. While the other implementations split the computation among 32 processes, and then let each process freely assign its subtasks to its threads, this one has to deal with 768 processes. Also, since at this level of parallelism the runtime of the problem is very short, the relative impact on it of the partitioning algorithm we implemented, which is the same in all the versions, is very strong despite being below a couple of seconds in this worst-case situation. Implementing a more efficient partitioning algorithm, from which our skeleton would also benefit, is part of our future work. The MPI versions also underperform with respect to the hybrid implementations for several other benchmarks for different reasons. For example, the best partitioning strategy for MPI `quicksort`, which prioritizes distribution over partitioning (see Fig. 3) not only makes very complex and expensive the load balancing but also makes it impossible to benefit from collective communication primitives. The larger requirements for communications and related data (de)serialization processes are common to all the MPI implementations, but while in some kernels the impact is negligible, in others it totally precludes the application from scaling. This is the case of the computational kernel of `tsp`.

The hybrid versions have a pretty similar performance for most benchmarks, the largest difference happening in `fib` because of the better code that the compiler generates for our skeleton. Despite being the only alternative that applies a high-level programming model both for the inter-process and inter-thread parallelization of the applications and that avoids having to write two versions of the D&C algorithms in all the situations, our framework is on average 10.6%, 5.8% and 5% faster than optimized hand-made codes that combine MPI with OpenMP, the `dac` skeleton and Cilk Plus, respectively across the set of parallel executions. If we discard `fib` because of the advantage the compiler provides to our proposal, the average improvement is still 2.9%, 0.2% and 0.5%, respectively. This way, the qualitative conclusion is that the skeleton is competitive with hand-optimized codes.

The impact of problem over-decomposition on performance is explored in Fig. 10, which shows for each benchmark and implementation the geometric mean of the speedup shown in Fig. 9 with respect to the one



(a) `fib` (b) `quicksort` (c) `nqueens` (d) `strassen` (e) `treeadd` allocation (f) `treeadd` computation (g) `tsp` allocation (h) `tsp` computation (i) `barnes hut` (j) `ep`

Fig. 9 Speedup of the parallel versions with respect to the sequential executions as a function of the number of cores/threads used.

achieved generating the minimum number of tasks required to have at least one task per thread. Notice that in many benchmarks it is impossible to generate exactly one task per thread, as for example if the arity is 2, the number of subproblems will be a power of 2, while our nodes have 24 cores. As expected, the importance of problem over-decomposition is stronger in the problems that exhibit large imbalances between tasks, the biggest imbalance corresponding to `fib` and `quicksort`. It also favors benchmarks whose tasks are balanced but for which it is not possible to generate exactly one task per core –for the reason just explained–,

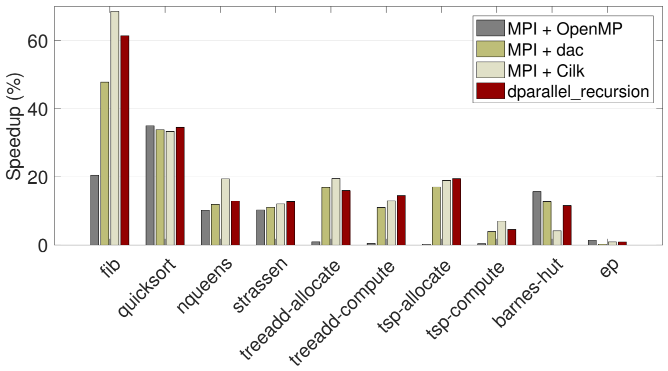


Fig. 10 Geometric mean of the speedup (as a percentage) achieved thanks to over-decomposition in the parallel executions.

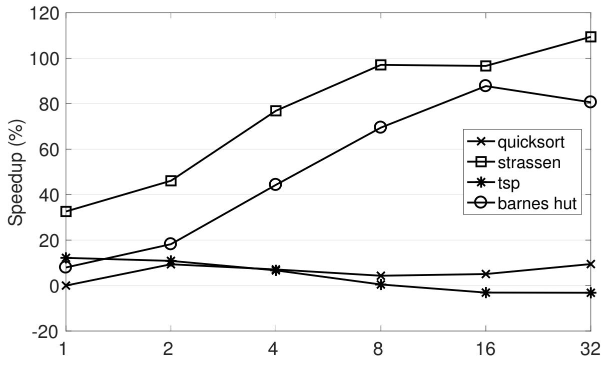


Fig. 11 Speedup increase of the skeleton thanks to the transmission by chunks for different numbers of nodes.

such as `treeadd` or `tsp`. In these applications the generation of a larger number of more fine-grained tasks thanks to over-decomposition helps reduce the imbalance of work among the cores. We also see that in general OpenMP is the approach that benefits less from this technique. This suggests that its task management mechanisms are less optimized than those of other threading approaches, something which has been pointed out by previous works [36]. These problems for OpenMP do not appear when the code does not need `taskwait` clauses, which is the case of `quicksort`.

Another optimization enabled by our framework that can be easily applied is the transmission by chunks of data items whose storage is not consecutive. Figure 11 shows the percentage of speedup growth achieved by the skeleton by using this optimization for the benchmarks where it can be applied. The values reported for `strassen` and `tsp` are the actual ones, as these benchmarks use this optimization. In the case of `quicksort` and `barnes hut`, the best version of these benchmarks does not use this optimization. The reason is that in these codes `dparallel_recursion` is invoked with flags that request to perform scatter or gather operations (see Table 4), which require, and thus assume, that the data to transmit is stored in a vector and can be therefore transmitted with a single collective communication primitive. For this reason, in their case the figure plots for informative purposes which would have been the impact on these benchmarks if they had not enjoyed the optimization based on collective communications, which would have implied sending the data by means of point to point messages with the associated serialization process for the data to communicate. We can see that this optimization is critical for `strassen`, and it would have been so for `barnes hut` if our framework could not exploit the collective gather optimization. The impact on `quicksort` would have been smaller because, as shown in Fig. 9(b), this benchmark has low scaling due to the fact that most of its cost is concentrated in its initial stages, in which the reduced number of subproblems allows to exploit little parallelism. Finally, although on average the optimization is positive for `tsp`, it introduces a small performance degradation when more than 8 nodes of the cluster are used. The reason is that there is a tradeoff between this optimization and the parallelism in the reception and deserialization of messages. Namely, the transmission by chunks implies that these chunks are received and deserialized in a given sequence by the receiver, as often the unpacking of a message must precede the processing of the next one. A good example is the transmission of a vector, in which the size must be obtained before allocating memory to receive and store the contents. Because `tsp` is the algorithm with the more expensive deserialization process, and it is also among the ones with the largest messages, the reduction of parallelism available when this optimization is applied outweighs its advantages by a narrow margin when the number of messages (and thus, the parallelism lost) is large.

5.2 Programmability comparison

The ideal strategy to compare the programmability of different approaches would be to ask a team of programmers to use them and compare the development times, the quality of the results and their opinions [40]. Unfortunately this is seldom possible. For this reason another widely used approach is to rely on objective metrics automatically extracted from the codes. The best known metric of this kind is probably the number of the source lines of code excluding comments and empty lines (SLOCs). Unfortunately this is a quite rough measure, as lines of code can widely vary in terms of length and complexity, which makes SLOCs a somewhat unreliable estimator. A more accurate metric is the Halstead programming effort [21], which estimates the development cost of a code by means of a reasoned for-

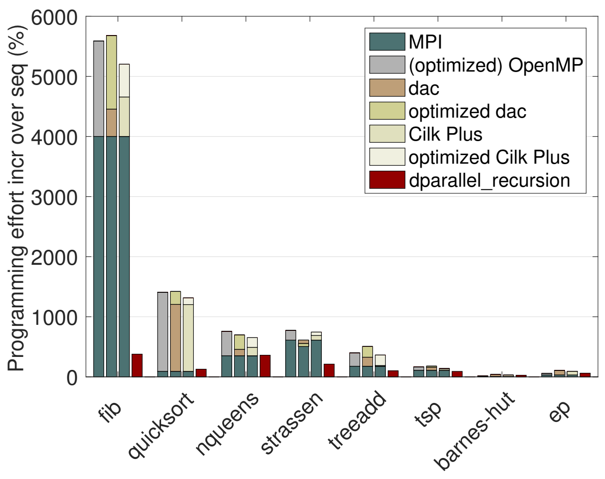


Fig. 12 Halstead programming effort comparison.

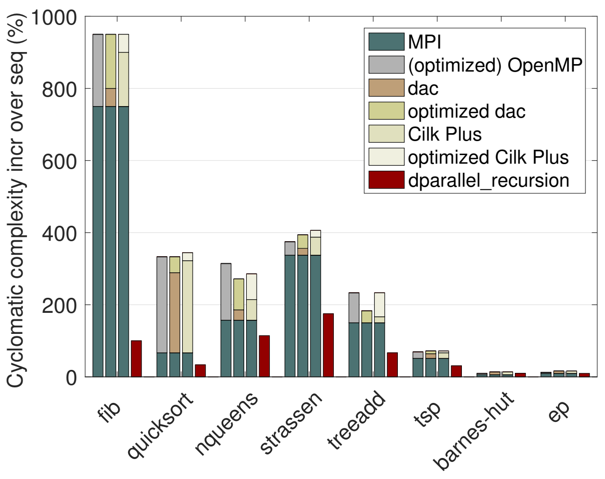


Fig. 13 Cyclomatic complexity comparison

mula based on the number of unique operands, unique operators, total operands and total operators found in the code. For this, the formula regards as operands the constants and identifiers, while the symbols or combinations of symbols that affect the value or ordering of operands constitute the operators. Another interesting metric is the cyclomatic complexity [33], which is defined as $V = P + 1$, where P is the number of decision points or predicates in a program. There is one predicate for each condition in the program that leads to a different execution branch, there being one for each `if`, `while`, `for`, or `case` statement as well as for each ternary conditional (`?:` operation). The larger V, the more complex the program is. Our programmability analysis will be based on these two latter metrics.

Figures 12 and 13 show the increase in programming effort and cyclomatic complexity of the different parallel versions of our benchmarks as a percentage of the corresponding ones of the sequential version of the algorithm, respectively. The figures represent in the same column the metrics for the MPI-only version and the increase that appears when parallelization based on OpenMP, the `dac` skeleton and Cilk Plus is performed too. Also, in the case of these two latter codes, the figure shows separately the increase when developing the basic and the optimized versions discussed in Section 5.1 and evaluated in Fig. 8. The MPI parallelization exhibits high costs when the optimal version involves balancing mechanisms and/or a relatively complex implementation such as the one required by the interleaving of partitioning and distribution of data favored by `strassen`. It is also natural that the simpler the kernel, the higher the relative cost and viceversa. This, together with the balancing algorithm is the reason why the complexity metrics increased in `fib` much more than in any other benchmark. In the opposite part of the scale, algorithms that can be parallelized with few MPI calls and a distribution of loop iterations among the threads of each process, such as `barnes hut` or `ep`, experience small complexity increases due to parallelization. When interpreting the results of `quicksort` we must take into account that since the MPI-only implementation could hardly benefit from a balancing algorithm in the distribution, this was integrated in the threaded versions and is thus attributed to the OpenMP, `dac` and Cilk Plus parallelization. Regarding the programming cost of the threading approaches, it is much smaller than the one of MPI, and since compiler directives, skeletons and Cilk Plus keywords are mechanisms with a reasonable high level of abstraction, the difference between the three alternatives is small when the code is fully optimized.

The metrics show a similar situation in all the benchmarks. Both the Halstead programming effort and the cyclomatic number for our skeleton are always similar or clearly better than those of the MPI-only version. When multithreading is incorporated to optimally exploit the resources within each node, the advantage of `dparallel_recursion` becomes even larger. Altogether, all the hybrid codes that are not based on our skeleton present a very similar complexity metrics. This way, no matter OpenMP, `dac` or Cilk Plus is considered, these codes have roughly about 150% more Halstead programming effort and a 90% higher cyclomatic number (geometric means) that those based on the skeleton proposed in this paper.

While the programmability metrics just discussed are very positive for our proposal, we really think that these figures do not make justice to the programmability advantages of `dparallel_recursion`. The reason is that they do not reflect the effort that a programmer may have to spend during the exploration of the implementation space of an algorithm, seeking the best one in a given hardware and software environment. While making changes to manually try different strategies for load balancing, serialization, communication, etc. can take large amounts of time depending on the problem at hand, with our library it is possible to quickly experiment with different alternatives just by changing the behavior bitset, and sometimes providing small support functions. In our experience based in the development of the codes for this paper, this is an enormous qualitative advantage.

6 Conclusions

Every cluster nowadays is composed of distributed memory nodes whose memory is shared by the cores of one or more processors. The optimal exploitation of these systems requires combining parallel programming models that are suited to these two situations, resulting in increased program complexity and cost, both for development and maintenance. A promising approach to deal with this situation is to encapsulate this complexity in skeletal operations that automate important parallel patterns, as long as they provide flexility to accommodate a reasonable range of situations and their performance is comparable with that of hand-tuned codes. However, there has not been much research on the development of skeleton libraries optimized for these environments.

In this paper we present `dparallel_recursion`, a C++ algorithm template with some supporting classes that implements the ubiquitous divide-and-conquer pattern of parallelism in current multi-core clusters. The skeleton was designed to provide a modular API based on simple semantics. It also supports large flexibility in the location of the data involved in the processing, allowing the parallelization of complex algorithms with reduced effort. This way, it not only supports the usage of existing data structures that can be distributed, replicated, or placed in a single node, but it can also distribute existing ones, or create in a distributed fashion new data structures. The distribution details are encapsulated in objects that allow to reuse the data structures in the skeleton invocations. Its design also makes it easy to use this algorithm template to implement simpler skeletons such as map or reduce, thus increasing its scope of application.

Much effort was put into making our skeleton as efficient as possible so that it could be competitive with hand-optimized implementations. The vast majority of the optimizations, such as its extensive internal parallelization or its exploitation of template metaprogramming to resolve polymorphism at compile time, are automatically provided by the library. Users can sometimes further optimize their codes with small hints. Examples are indicating whether the objects to be transmitted need no serialization or whether they benefit from sending separately each one of their components rather than packing them all in a single message.

Experiments using up to 768 cores show that the performance of our proposal is comparable to —and often better than– that of manually fine-tuned codes parallelized combining MPI with other approaches to exploit parallelism in shared memory. Even if we disregard one benchmark where the compiler gives a strong advantage to our skeleton, the codes based on it were on average between 0.2% and 2.9% faster than optimized manual implementations, depending on the tool chosen for thread parallelism. Regarding programmability, an evaluation based on objective metrics extracted from the codes indicates that the effort in the parallelization of an application for multi-core clusters using `dparallel_recursion` is on average between 47% (cyclomatic number) and 60% (Halstead programming effort) of the one involved by the other alternatives tested. Based on these results, we conclude that our algorithm template is an excellent alternative for the implementation of D&C algorithms in multi-core clusters from both the performance and the programmability points of view.

While other improvements are possible, we envision two main possible lines of future work for this library. One is extending it with implementations of other relevant skeletons following the same philosophy. Another possibility is to design mechanisms that allow the algorithm template to exploit hardware accelerators.

The library is publicly available under an open source license at https://github.com/fraguela/dparallel_recursion.

Acknowledgements

This research was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds (80%) of the EU (Ref. TIN2013-42148-P and TIN2016-75845-P), and by the Galician Government under the Consolidation Program of Competitive Reference Groups (Ref. GRC2013/055). We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the use of their computers, as well as the 16 anonymous reviewers because of their valuable feedback and suggestions.

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.

2. M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Gener. Comput. Syst.*, 19(5):611–626, 2003.
3. J. Barnes and P. Hut. A hierarchical O (N log N) force-calculation algorithm. *Nature*, 324(4):446–559, Dec 1986.
4. Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The Science of Deriving Dense Linear Algebra Algorithms. *ACM Trans. Math. Softw.*, 31(1):1–26, March 2005.
5. R. D. Blumofe, C. F. Joerg, B.C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.
6. Boost.org. Boost C++ libraries. http://boost.org, 2016. Last access December 10, 2016.
7. P. Ciechanowicz and H. Kuchen. Enhancing Muesli's data parallel skeletons for multi-core computer architectures. In *12th IEEE Intl. Conf. on High Performance Computing and Communications, (HPCC 2010)*, pages 108–113, Los Alamitos, CA, 2010. IEEE.
8. M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, 1989.
9. M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.
10. M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati. A divide-and-conquer parallel pattern implementation for multicores. In *Proc. 3rd Intl. Workshop on Software Engineering for Parallel Systems*, SEPS 2016, pages 10–19, New York, NY, USA, 2016. ACM.
11. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
12. A. Denis. pioman: A pthread-based multithreaded communication engine. In *Proc. 23rd Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP 2015)*, pages 155–162, Los Alamitos, CA, March 2015. IEEE.
13. J. Falcou, J. Sérot, T. Chateau, and J-T. Lapresté. Quaff: efficient C++ design for parallel skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.
14. P. J. Fleming and J. J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.
15. M. Frigo, C.E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Procs. 40th Annual Symp. on Foundations of Computer Science*, FOCS '99, pages 285–297, Washington, DC, USA, 1999. IEEE Computer Society.
16. C. H. González and B. B. Fraguela. 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)*, pages 79–88, Los Alamitos, CA, 2010. IEEE.
17. C. H. González and B. B. Fraguela. A framework for argument-based task synchronization with automatic detection of dependencies. *Parallel Computing*, 39(9):475–489, 2013.
18. C. H. González and B. B. Fraguela. An algorithm template for domain-based parallel irregular algorithms. *International Journal of Parallel Programming*, 42(6):948–967, 2014.
19. S. Gorlatch and M. Cole. Parallel skeletons. In *Encyclopedia of Parallel Computing*, pages 1417–1422. Springer, New York, 2011.
20. D. Gregor and M. Troyer. Boost.MPI. http://boost.cowic.de/rc/pdf/mpi.pdf, 2007.
21. M. H. Halstead. *Elements of Software Science*. Elsevier, New York, NY, USA, 1977.
22. P. Hijma, C. J. H. Jacobs, R. V. v. Nieuwpoort, and H. E. Bal. Cashmere: Heterogeneous many-core computing. In *29th IEEE Intl. Parallel and Distributed Processing Symposium (IPDPS 2015)*, pages 135–145, May 2015.
23. E. Horowitz and A. Zorat. Divide-and-conquer for parallel processing. *IEEE Transactions on Computers*, C-32(6):582–585, June 1983.
24. Intel®. Cilk^{TM} Plus. https://www.cilkplus.org, 2016. Last access December 10, 2016.
25. Y. Karasawa and H. Iwasaki. A parallel skeleton library for multi-core clusters. In *Proc. 2009 Intl. Conf. on Parallel Processing (ICPP'09)*, pages 84–91, Los Alamitos, CA, Sept 2009. IEEE.
26. T. Kawakatsu, A. Kinoshita, A. Takasu, and J. Adachi. *Divide-and-Conquer Parallelism for Learning Mixture Models*, pages 23–47. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.
27. H. Kuchen. A skeleton library. In *Euro-Par 2002 Parallel Processing*, volume 2400 of *Lecture Notes in Computer Science*, pages 620–629. Springer Berlin Heidelberg, Berlin, Germany, 2002.
28. M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *2009 IEEE Intl. Symp. on Performance Analysis of Systems and Software*, pages 65–76, April 2009.
29. M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Proc. 18th Euromicro Conf. on Parallel, Distributed and Network-based Processing (PDP 2010)*, pages 289–296, Los Alamitos, CA, Feb 2010. IEEE.
30. J.V.F. Lima, F. Broquedis, T. Gautier, and B. Raffin. Preliminary experiments with XKaapi on Intel Xeon Phi coprocessor. In *Proc. 25th Intl. Symp. on Computer Architecture and High Performance Computing (SBAC-PAD 2013)*, pages 105–112, Los Alamitos, CA, Oct 2013. IEEE.
31. D. A. Mallón, G. L. Taboada, C. Teijeiro, J. Touriño, B. B. Fraguela, A. Gómez-Tato, R. Doallo, and J. Carlos Mouriño. Performance evaluation of MPI, UPC and OpenMP on multicore architectures. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting*, pages 174–184, Berlin, Germany, 2009. Springer-Verlag.
32. T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, Boston, MA, 2004.
33. T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
34. Y. Nakatsukasa and N. J. Higham. Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the SVD. *SIAM J. Scientific Computing*, 35(3), 2013.
35. National Aeronautics and Space Administration. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/, 2010. Last access December 10, 2016.
36. S. L. Olivier and J. F. Prins. Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs. *Intl. J. Parallel Program.*, 38(5-6):341–360, 2010.
37. J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, Sebastopol, CA, July 2007.

38. A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. on Programming Languages and Systems*, 17(2):233–263, March 1995.
39. G. Tang, W. Yang, K. Li, Y. Ye, and K. Xiao, G.and Li. An iteration-based hybrid parallel algorithm for tridiagonal systems of equations on multi-core architectures. *Concurrency and Computation: Practice and Experience*, 27(17):5076–5095, 2015.
40. C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan. Evaluation of UPC programmability using classroom studies. In *Proc. Third Conf. on Partitioned Global Address Space Programing Models*, PGAS '09, pages 10:1–10:7, New York, NY, USA, 2009. ACM.
41. E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta. A high-productivity task-based programming model for clusters. *Concurrency and Computation: Practice and Experience*, 24(18):2421–2448, 2012.
42. A. Tousimojarad and W. Vanderbauwhede. Number of tasks, not threads, is key. In *Proc. 23rd Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP 2015)*, pages 128–136, Los Alamitos, CA, March 2015. IEEE.
43. R. V. Van Nieuwpoort, G. Wrzesińska, C. J. H. Jacobs, and H. E. Bal. Satin: A high-level and efficient grid programming model. *ACM Trans. Program. Lang. Syst.*, 32(3):9:1–9:39, March 2010.
44. J. Walter and M. Koch. Boost basic linear algebra library (uBLAS). http://www.boost.org/libs/numeric/ublas/, 2002. Last access December 7, 2016.
45. T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.
46. K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *Proc. 2007 Intl. Workshop on Parallel Symbolic Computation*, PASCO '07, pages 24–32, New York, NY, USA, 2007. ACM.
47. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. 2nd USENIX Conf. on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
48. W. Zang, P. Zhang, C. Zhou, and L. Guo. Locating multiple sources in social networks under the SIR model: A divide-and-conquer approach. *Journal of Computational Science*, 10:278 – 287, 2015.
49. Y. Zhang, J. C. Duchi, and M.J. Wainwright. Divide and conquer kernel ridge regression. In *26th Annual Conf. on Learning Theory (COLT 2013)*, pages 592–617, june 2013.