The Journal of Supercomputing
https://doi.org/10.1007/s11227-021-04259-5

A highly optimized skeleton for unbalanced and deep divide-and-conquer algorithms on multi-core clusters

Millán A. Martínez[1] · Basilio B. Fraguela[1] · José C. Cabaleiro[2]

Accepted: 21 December 2021

Abstract
Efficiently implementing the divide-and-conquer pattern of parallelism in distributed memory systems is very relevant, given its ubiquity, and difficult, given its recursive nature and the need to exchange tasks and data among the processors. This task is noticeably further complicated in the presence of multi-core systems, where hybrid parallelism must be exploited to attain the best performance, and when unbalanced and deep workloads are considered, as additional measures must be taken to load balance and avoid deep recursion problems. In this manuscript a parallel skeleton that fulfills all these requirements while providing high levels of usability is presented. In fact, the evaluation shows that our proposal is on average 415.32% faster than MPI codes and 229.18% faster than MPI + OpenMP benchmarks, while offering an average improvement in the programmability metrics of 131.04% over MPI alternatives and 155.18% over MPI + OpenMP solutions.

Keywords Algorithmic skeletons · Divide-and-conquer · Template metaprogramming · Load balancing · Multi-core clusters · Hybrid parallelism

1 Introduction

The development of parallel applications requires a great effort, particularly when the best possible performance is sought and distributed memory systems, or worse, systems with portions of both shared and distributed memory, are

✉ Millán A. Martínez
millan.alvarez@udc.es

Basilio B. Fraguela
basilio.fraguela@udc.es

José C. Cabaleiro
jc.cabaleiro@usc.es

1 Universidade da Coruña, CITIC, Computer Architecture Group, 15071 A Coruña, Spain

2 Centro Singular de Investigación en Tecnoloxías Intelixentes (CiTIUS), Dpto. Electrónica e Computación, Universidade de Santiago de Compostela, 15782 Santiago de Compostela, Spain

Published online: 24 January 2022

involved. In our opinion, one of the best approaches to deal with this problem are parallel skeletons [17], which hide the complexity of parallelism while providing good performance as well as high-level semantics and easy-to-use APIs. Skeletons target different parallel patterns, one of the most relevant ones arguably being divide-and-conquer, hence denoted D&C. The reason for its importance stems from two facts. The first one is that it is the natural expression for many essential algorithms in a wide array of fields [16]. The second one is that a properly designed D&C skeleton can be used to express several of the other most basic and critic parallel patterns such as map or reduce, thus much further expanding its scope of applicability.

As a result of the importance of D&C, many libraries of parallel skeletal operations consider it, although often restricted to shared memory environments [8, 15, 22]. Still, a critical restriction found in the literature is the lack of implementations optimized for unbalanced problems. This is particularly true for distributed memory systems, as shared memory implementations usually provide limited load balancing strategies which, even if present, are noticeably less performant than implementations optimized for unbalanced workloads [24]. This is a very serious restriction, as many problems naturally present different degrees of imbalance, thus heavily restricting the performance achievable with solutions that do not consider it. This problem is aggravated by the fact that manually implementing good dynamic load balancing techniques is very challenging, particularly in distributed memory environments, not to mention systems where both shared and distributed memory parallelism are exploited.

A second issue that was also considered in [24] in shared memory is the fact that D&C frameworks usually rely on stack-based recursivity. While this is convenient and performant, in the case of deep levels of recursion this approach can easily break the application due to the limited availability of this kind of memory if not properly configured, which is sometimes non-trivial or even cannot be done due to system-level restrictions. The heap-based alternative proposed in [24] proved to avoid this problem while providing good performance.

Given the discussion above and the existence of many large problems that require the usage of distributed memory to be solved, where load imbalance is much more critical, we set to implement the first skeleton for the resolution of D&C problems in multi-core clusters developed with unbalanced and deep problems in mind. Our proposal is a natural evolution of [24], which was restricted to shared memory, and as a result of its focus, it follows a totally new strategy compared to all the previous parallel skeletons we know of. Our implementation relies on MPI for interprocess communications and on C++11 threads for efficient multithreading within each process. Dynamic work balancing is provided both between the different processes and the threads within each process, always using known efficient work-stealing techniques to minimize overload. The design also solves the problems associated with deep recursion trees in the D&C problem mentioned above. Furthermore, the proposal provides several configurable parameters and mechanisms that allow adapting its behavior to different types of problems and environments. While most of them can be safely ignored, since their default value is suitable for most situations, they can be used to fine-tune the execution and further increase performance. The new implementation is available at https://github.com/fraguela/dparallel_recursion together with the material published in [15, 16, 24].

The rest of this manuscript is organized as follows. Section 2 reviews the related work, while Sect. 3 presents the new proposal and explains its syntax and implementation. The performance and programmability evaluation is found in Sect. 4. This is followed by our conclusions and future work in Sect. 5.

2 Related work

While there are numerous libraries of skeletal operations, many of them are restricted to shared memory environments, e.g., [8, 22, 24]. In fact, our proposal uses shared memory techniques inherited from `parallel_stack_recursion` [24], a skeleton that relies on stacks where the problems that must be processed are stored and advanced work-stealing techniques to achieve load balancing among the participating threads. In our library, work-stealing is also applied to the distributed part of the implementation thanks to the good levels of performance and scalability that it allows to achieve. This is a widely known technique used by several libraries, such as [29] for shared memory and [10] for distributed memory. A variety of solutions that use different types of work-stealing adapted to their specific needs are collected in [31], where their advantages and counterparts are analyzed. Some of these strategies have been incorporated into our implementation adapting them for D&C problems.

There is also a fair share of skeletal libraries that support distributed memory systems, our interest being focused on those that provide D&C. For example, `eSkel` [7] provides parallel skeletons for C on top of MPI, including one for D&C. Due to the C language limitations the API is somewhat low-level, which leads, for example, to the exposure of MPI details. This also precludes this library from benefiting from the large advantages that object-oriented languages provide to the development of libraries such as encapsulation or polymorphism. `Lithium` [1] is a Java library that provides, among others, a parallel skeleton for D&C. Its implementation is based on macro data flow instead of templates and it extensively relies on runtime polymorphism. This differs from `Quaff` [13], where the programmer must encode a task graph by means of C++ type definitions which are processed at compile time to produce optimized message-passing code. Due to these static definitions, tasks cannot be dynamically generated at arbitrary levels of recursions. Therefore, the library allows skeleton nesting but it has the limitation that this nesting must be statically defined.

A C++ library built on top of MPI is `SkeTo` [20]. It centers around data-parallel skeletons on distributed data types and it provides no support for task-parallel skeletons such as D&C. `Muesli` [6] is a C++ library that combines the use of MPI and OpenMP. However, this hybrid solution is only applied to its data-parallel skeletons, so that its D&C skeleton is only optimized for distributed memory. Further, this library relies on runtime polymorphism, which represents large overheads compared with the use of template metaprogramming and static polymorphism techniques such as the our skeleton uses.

D&C skeletons specifically oriented to multi-core clusters are [19] and [16]. Both of them combine message-passing in order to support distributed memory with multithreading within each process. While [16] lacks load balancing mechanisms among processes and only provides basic load balancing among threads, [19] supports load balancing by means of work-stealing, although its balancing operations always involve a single task, which can be somewhat inefficient. In addition, this latter proposal is only evaluated with very balanced algorithms and unfortunately, contrary to ours, it is not publicly available.

There are other high-level proposals that support the parallelization of D&C problems beyond skeletons. For example, the `Merge` [23] framework couples a new language based on map-reduce and an associated compiler with a dynamic runtime that automatically distributes computations among different cores in a heterogeneous multi-core system. `Petabricks` [2] proposes a new implicitly parallel programming language and compiler. Programs written in this language can naturally describe multiple algorithms for solving a problem and how they can be fit together. This information is used by the compiler and runtime to create and autotune an optimized hybrid algorithm.

Another line of research related to our work is the development of techniques and tools to identify computational patterns in sequential programs and refactor these codes in order to enable their effective parallelization. Many of these works consider explicitly or are specifically devoted to the D&C pattern targeted by our skeleton [14, 21, 30], further justifying the importance of this pattern. Coupling the analysis and transformations proposed by these works with efficient implementations like ours could enable the much desirable automatic optimized parallelization of D&C algorithms by advanced compilers.

3 A divide-and-conquer skeleton for unbalanced problems on distributed memory systems

In this section we describe our new proposal, called `dparallel_stack_recursion`. We will begin with its syntax and semantics and continue with the implementation and optimizations.

3.1 Syntax and semantics

A D&C algorithm involves four kinds of tasks: deciding whether a problem is a base case or it can be further subdivided, solving a base case, subdividing a non-base case, and combining the results of the subproblems to get the joint solution to the original problem. Our library provides two C++ class templates, called `Info` and `Body`, whose member functions provide all the information needed to perform these tasks, and which users must therefore specify in order to adapt the skeleton to each specific problem at hand. Listing 1 shows a detailed description of these templates.

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

template<typename T, typename S, typename ProcNonBase = false>
struct Body : EmptyBody<T, S, ProcNonBase> {
  void pre(T& t); //preprocessing of t before partition
  void pre_rec(T& t); //preprocessing of t before partition
  S base(T& t); //solve base case
  S non_base(T& t); //solve non−base case
  S post(const S& local_result, S& global_result); //combine children solutions
  S gather_input_post(const T& t, int n, T& root); //reduction over node data elms.
};
```

Listing 1: Templates with signatures for the `Info` and `Body` classes

The `Info` class provides the structure of the problem and it must derive from a class `Arity<N>` provided by the skeleton, where `N` is either the number of children of every non-base case of the problem, when it is fixed, or the identifier `UNKNOWN` when this value is not a constant. In the last case, the user must implement the `num_children` function to return the number of subproblems of each non-base problem. Function `child(i, t)` must return the `ith` child of the non-base problem `t`, while `is_base(t)` must return a Boolean that indicates whether the problem `t` is a base case or not.

The `Body` class provides the computations and it must provide the functions shown in Listing 1. Here, `base(t)` supplies the solution for a base case `t`, while `non_base(t)` solves a non-base case. This last function is optional and it is not used in all D&C algorithms, but it is especially useful when the internal nodes of the recursion tree contribute to the result. Function `post(local_result, global_result)` receives a `local_result` element that is a partial result obtained by processing a problem in the recursion tree, and `global_result` is the global result with which this result must be reduced. Finally, `gather_input_post` is only occasionally used, as we will see later.

This class also supports two member functions that allow performing computations on a problem `t` before further processing. Namely, `pre` is run before checking whether the problem is a base case, while `pre_rec` is executed only for non-base cases after this checking. This way it is ideal to perform computations before the subproblems are generated, and even before the number of children of this problem `t` is obtained. These two functions are optional and they were found to be useful for some D&C problems. The library provides a class template `EmptyBody<T,S,ProcNonBase>` that can be optionally used as base case for the `Body` classes, where `T` is the type of the problems and `S` is the type of the solutions. The `ProcNonBase` is an optional Boolean parameter, with `false` value as default, that must be set to `true` if the `non_base` function is used. The main advantage of this template is that it provides empty implementations of all the `Body` functions, so that deriving a class from it avoids writing unneeded components.

An optional parameter of our skeleton is the partitioner, which can be a `simple_partitioner` or a `custom_partitioner`. The simple partitioner, which is the default, parallelizes every problem. The custom partitioner relies on a user-provided `do_parallel(t)` function of the `Info` class to decide whether a problem `t` should be processed in parallel or not. When a problem proceeds with a sequential execution, this problem and all its subproblems are processed in a sequential execution. This partitioner is especially useful when the problems can have a very fine grain, so that the overhead caused by their management can be significant. Custom partitioning should be used with caution because the sequential execution of whole subtrees of a problem can cause performance losses and even stack overflow errors, as this execution relies on a natural recursive stack-based implementation. This last issue can be avoided by increasing the system limits to allow for larger stack sizes, although this also has its drawbacks.

The only argument to the skeleton that is related to distributed memory is called *behaviour_flags*. It is a bitmask of flags that informs about the distribution of the inputs and outputs. These flags are basically those from [16], and they are:

- `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. The only copy of the result of the algorithm will be located in this process when the computation finishes.
- `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.
- `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.
- `DistributedInput`: 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.
- `ReplicateInput`: indicates that the input of the process with rank 0 should be replicated to all processes before the computations starts.
- `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. The use of this flag entails an additional computational and memory cost, since it is necessary to store and not delete the data of each child problem generated. The `gather_input_post` function of the body object will be executed at the end for each child problem

in order to perform a reduce operation on them and save the results in the initial *root* problem.

The last argument received by our skeleton is a configuration object that allows the user to control multiple aspects of the internal behavior of the library that can greatly influence the performance achieved. The configuration object contains a data member for each controllable parameter, and those not explicitly specified by the user take default values that provide very good performance in general. Since these parameters are related to the internal operation of the library, they are explained in Sect. 3.2.

Other API specifications were developed for data communication since MPI does not directly allow sending arbitrary data structures and classes. Our skeleton uses the `Boost` [5] library to serialize objects for transmission through MPI, being deserialized at the destination and converted back into the corresponding object. It may be thus necessary for the user to indicate how this serialization is done, which is quite easy to do with `Boost`. When primitive data types are to be transmitted, no additional API or indications are needed. When the objects involved are not primitive but they are bitwise serializable, it is only necessary to indicate that the class meets this condition with the macro `BOOST_IS_BITWISE_SERIALIZABLE(class)`. In case the class is not bitwise serializable, then the user must follow the API of the Boost serialization library.

```
struct result_t {
  unsigned int nfolders;
  unsigned int nfiles;
  unsigned long int tsize;
};

struct item_t {
  std::vector<item_t*> childrenObj;
  unsigned long int size;
  bool isFile;
};

struct FileSystemInfo: public Arity<UNKNOWN> {
  bool is_base(const item_t *t) const { return t->isFile || t->childrenObj.empty(); }
  int num_children(const item_t *t) const { return t->childrenObj.size(); }
  item_t *child(int i, const item_t *t) const { return t->childrenObj[i]; }
};

struct FileSystemBody: public EmptyBody<item_t *, result_t, true> {
  result_t base(item_t * t) { return result_t{!t->isFile, t->isFile, t->size}; }
  result_t non_base(item_t * t) { return base(t); }
  void post(result_t local, result_t& global) {
    global.nfolders += local.nfolders;
    global.nfiles += local.nfiles;
    global.tsize += local.tsize;
  }
};

result_t r = dparallel_stack_recursion<result_t>(root, FileSystemInfo(), FileSystemBody(),
      partitioner::simple(), DefaultBehavior, Config());
```

Listing 2: Using `dparallel_stack_recursion` to traverse a file system and gather information.

Listing 2 exemplifies the usage of our skeleton on a tree representing a file system. Each `item_t` node in the tree contains information about its size, whether it is a file or a folder, and, in the latter case, a vector of pointers to the items that the folder contains. Each folder can contain a different number of items, so the number of children of each node is variable. The problem consists in traversing this tree and computing a structure `result_t` with the total number of folders and files, and their total size. The `is_base` member function of the `FileSystemInfo` class identifies the base cases, which are the nodes in the tree that have no children or are files. The `num_children` function returns the number of children of each node, while the child nodes are returned by the `child` function. The `base` function of the `FileSystemBody` class processes the base nodes returning a `result_t` object with the info of the node. The `ProcNonBase` value of `FileSystemBody` class is set to `true`, which means that function `non_base` must also be executed for each non-base case. The results returned by all the nodes are processed via the `post` function of body object, where the reduction on the global `result_t` is done. The last line in Listing 2 illustrates the invocation of the skeleton with the root of the tree, the info and body objects, a simple partitioner to run all tasks in parallel without cutoffs, and a default behavior flag and configuration object. The user should also add the necessary `Boost` macros and functions related to the serialization of relevant classes, as indicated before.

Finally, it also deserves to be mentioned that while the `dparallel_stack_recursion` skeleton is the kernel of our library, it also includes other components that facilitate its use. The main ones are range classes that provide automatic partitioning and macros and function templates to implement parallel loops and reductions on top of our skeleton using a very simple syntax.

3.2 Implementation and optimizations

Since our skeleton strongly focuses on unbalanced workloads, it implements advanced work-stealing methods both in shared and distributed memory. The shared memory parallelization and load balancing rely on C++11 threads and synchronization facilities, inheriting the techniques used and explained in detail in [24]. This way, each thread uses a separate stack allocated in the heap memory to store the subproblems that it generates. When a thread finds that its stack is empty, it first tries to steal a set of tasks from another thread. The granularity of the steals, that is, the number of problems that are stolen at once, is defined by a parameter called *chunkSize*. This value allows stealing several problems at once, which reduces the average overhead per stolen item. However, using too large values limits the chances of a successful steal and it can keep some threads idle for longer.

The distributed memory implementation of the skeleton relies on MPI. Therefore, after an unsuccessful attempt to steal work from other local threads, an idle thread performs an MPI request to obtain tasks from other process and it waits for the response. After that, if no work could be stolen, the cycle is repeated, so that the thread tries again first to steal work from other local threads and then, if this fails, to steal work from other process, and so on.

Except for some data synchronizations at the beginning and the end of the execution, all communications between processes are made through point-to-point messages. This allows for greater scalability and good performance even when using a very high number of cluster nodes. Almost all these MPI communications are non-blocking, thus allowing threads to overlap communications with other activities. Our implementation allows a single thread to request work in a single MPI request for several threads, thus reducing the number of messages. Every request receives an associated response, which can be either the requested work itself or a `null` response if the process that received the request has no spare work available. In the case of a request for multiple threads, the response can be partial, that is, it may only return work for some threads in the case in which the target process does not have enough tasks for all the requester threads.

In our implementation, any thread can make work requests, but these requests will only be addressed to specific processes and not to threads. In addition, any thread of a process that receives a work request can handle it. The thread in charge of handling an MPI request will check the available work in all the threads of the process, it will steal the available work from these threads in a similar way to how the steals are carried out between the threads in shared memory, and it will send this work as a response. In case of not finding work, it will send a message indicating that the request could not be satisfied. As we previously discussed, among threads of the same process, the *chunkSize* parameter determines the number of tasks that are stolen in one steal step. Among processes, the new parameter *chunksToSteal* defines how many chunks should be stolen at once with one MPI request. Its default value is `1` and there is seldom need to modify it, since it is usually a good value that provides good performance.

Each process must periodically check whether there are unprocessed work requests from another process. All threads perform this check, and they have the ability to search and collect work from among all the threads in the same process in order to gather the requested work, pack it, and send it to the requester. This check is done periodically when the thread is idle because it has no work. But in addition, it is also necessary to carry out this check even if the threads have work in order to avoid very long waits in the requester processes. Unfortunately this activity can greatly affect performance. If the checks are performed too often, the associated overhead degrades the performance, but a long wait between checks implies slower responses to work requests, and therefore a global worst performance. The parameter *polling_rate* controls the rate for checking new MPI work requests. The optimal value for this parameter depends on many factors, such as the type of problem, its implementation, and the hardware available. In order to address this, a functionality has been implemented that allows using an automatic adaptive polling rate. This mechanism dynamically changes the value of the polling rate during execution, linearly increasing it when no work is found as response to the request, and multiplicatively decreasing it when work is found. This adaptive polling rate is enabled by default, and it performed very well in our tests, providing performance very close to the one achieved with the optimal value.

Another important feature of our implementation is the limitation of the number of outstanding MPI requests per process. Indeed, we have observed that not restricting it can severely hurt performance due to network congestion and the messages processing costs when there is high imbalance or the algorithm execution is in its final stages. This limit can be defined with the *mpi_workrequest_limits* parameter. As a related optimization in our implementation, before a thread launches an MPI request, it observes the state of the other threads in order to check whether anyone needs work and has not made an MPI request yet. If this is the case, the thread launches a single joint MPI request for it and these other threads. This allows seeking work for all the threads even if the outstanding number of requests allowed is smaller than the number of threads, making a smarter usage of the requests available and contributing to the proper load balancing among processes despite the reduction in messages and network congestion. Taking into account this second optimization, the default limit in our library for the number of requests from each process given by *mpi_workrequest_limits* is equal to half the number of threads per process, which resulted in stable and close to optimal performance in all our tests. As a result, users do not really need to modify this parameter unless they want to fine tune it.

Detecting when an algorithm finishes in a distributed memory environment is not as easy as it is in shared memory. Our library uses a ring-based termination detection algorithm similar to that proposed by Dijkstra et al. [9]. In this algorithm, each process passes the token to its right in order to reach a consensus. The token can be of three colors: *white*, *pink* or *red*, and it has information about the number of requests sent and received from each process. *White* is the default color while the algorithm has not finished. With the token information, a process can detect a termination condition and set the token color to *pink*. This color is a transition state to terminate pending communications, but new requests are forbidden. When all these communications are finished, the token is set to *red* and the computation ends.

4 Evaluation

The evaluation benchmarks and their sequential runtime are described in Table 1. This time was obtained by running a purely sequential optimized version of each benchmark in a single core of our evaluation system. These codes were compiled with the same software environment and optimization flags as the parallel experiments, which will be discussed later. Also, all the runs where performed in an exclusive fashion, where the nodes used were available only for the tests and no other software besides the OS was running.

The *uts* (Unbalanced Tree Search [26]) benchmark processes unbalanced trees of different shapes and sizes that follow different distributions according to the parameters passed to the program. The *T3XXL* tree is predefined in the *uts* distribution package and it processes a binomial tree, which is a very unbalanced and unpredictable problem. This is an optimal adversary for load balancing strategies. The second configuration used is *T2XL*, which is a geometric tree with a circular factor branch, which although somewhat more predictable, is unbalanced enough to be challenging. Its *uts* parameters are `-t 1 -a 2 -d 26 -b 7 -r 220`. The *N Queens* benchmark solves the N Queens puzzle problem, which computes on how many ways can n chess queens be placed on an $n \times n$ chessboard so that no two queens threaten each other. The *fib* benchmark implements the recursive algorithm to compute the *nth* Fibonacci number. This is an inefficient method, but it is often used in the literature of D&C and unbalanced algorithms. This algorithm with very simple calculations is interesting when evaluating parallel solutions because the overheads can be clearly observed. Finally, *topsorts* computes the topological sorts of a direct acyclic graph using the reverse search approach for enumeration [3]. A topological

Table 1 Benchmarks used

Name	Problem size	Seq. time (s)
uts-T3XXL	Binomial tree, 2793M n, h 99049	519.87
uts-T2XL	Geometric [cyclic branch factor] tree, 1495M n, h 104	457.09
N Queens	17×17 number	1316.60
fib	57th Fibonacci number	1395.64
topsorts	24466M n	4795.64

n stands for nodes and h for heights

sort or order is a linear ordering of the nodes where for each directed edge from node A to node B, A appears before B in the ordering. This is an unbalanced benchmark, where the level of imbalance is highly determined by the input, thus an input that produces a fairly considerable level of imbalance is used in the evaluation.

We will first analyze the performance of the skeleton, which will be followed by a study on programmability and a discussion. The evaluation will always consider a sequential version, a version developed using only MPI, and another one based on the `dparallel_recursion` [16] algorithm template, which also provides D&C computations optimized for environments with shared and distributed memory, but which lacks efficient load balancing mechanisms. This latter skeleton uses the low-level API of the Intel TBB [27] for shared memory operations and a custom MPI implementation for the distributed part. In addition, some benchmarks will also consider a manually developed MPI + OpenMP version. For the sake of fair comparison, the MPI-only and MPI + OpenMP implementations take a balanced approach, using code that is not very complex but which includes a reasonable initial load balancing mechanism that allows for significant performance improvement without worsening too much the programmability metrics. In OpenMP this involves a user selected cutoff mechanism to prevent excessive task creation that greatly benefits performance. The best value of this parameter for each execution was used in every benchmark. The exceptions to this methodology are *uts* and *topsorts*, for which no manual MPI + OpenMP version was developed because of their complexity and the fact that they enjoy complex MPI-based baselines developed by other authors that are specifically targeted to addressing their imbalance problems. Namely, the code used for the *uts* MPI implementation was the MPI version developed by its authors [11]. Similarly, for the MPI version of *topsorts* the *mts* [4] framework was used, which is especially developed and optimized for this type of problem. As for the MPI, MPI + OpenMP and `dparallel_recursion` versions of *N Queens* and *fib*, we used those from [16], which are available in its public repository. Finally, we wrote the `dparallel_recursion` versions of *uts* and *topsorts* following the same strategy used in the other benchmarks implemented with this algorithm template, which is also the same used by the new proposal.

Two different versions are tested for `dparallel_recursion` and `dparallel_stack_recursion`, one that does not limit the creation of tasks (*simple*), and another one manually tuned to limit the number of tasks created by a cutoff mechanism (*manual*). The `dparallel_recursion` tests include a third *automatic* version that relies on its *automatic* partitioner.

4.1 Performance evaluation

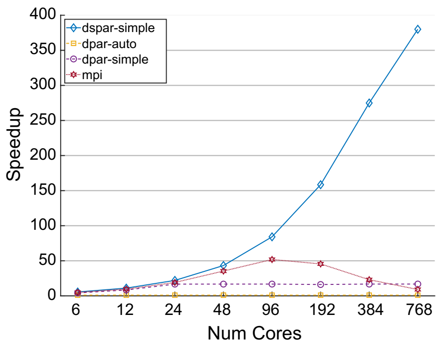
All the experiments, both sequential and parallel, were compiled and run with optimization level O3 in a cluster with the specifications described in Table 2. A total of ten runs are performed for each benchmark case and their average runtime is selected as the final execution time. The speedup of the parallel executions is computed by dividing the sequential execution time taken from Table 1 by the measured parallel

Table 2 System configuration

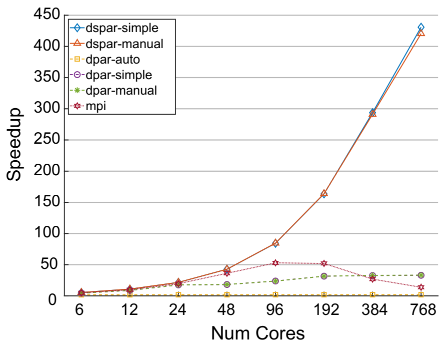
Feature	Value
CPUs per node	2 x Intel Xeon E5-2680v3
CPU family	Haswell
CPU frequency	2.5 GHz
Num cores/CPU	12
Total cores per node	$2 \times 12 = 24$
Memory per node	128 GB DDR4
Operating system	RHELS 7.5 (Maipo)
Kernel version	3.10.0-862.14.4.el7.x86_64
Num nodes	32
Compiler	g++ 6.4.0
OpenMP version	4.5
TBB version	2018.4.222
MPI version	OpenMPI 2.1.5
Interconnection network	Infiniband FDR@56 Gbps
Topology	Fat-tree

runtime. Also, in every execution it was verified that the result of the program was correct. The performance tests use 1, 2, 4, 8, 16, and 32 nodes. In the single node runs, 6, 12, and 24 cores are used. The experiments with 2, 4, 8, 16, and 32 nodes always involve the 24 cores of each node, resulting in a total of 48, 96, 192, 384, and 768 cores, respectively. We tested the usage of a variable number of processes per node and threads per process, always exploiting all the cores in each node. The best for `dparallel_stack_recursion` and `dparallel_recursion` is always to use one process with 24 threads per node. For MPI + OpenMP the optimal number of processes per node depends on the benchmark and the number of nodes used, varying between 1, 2, or 4 processes per node. Our evaluation in Fig. 1 always uses the best configuration for each situation.

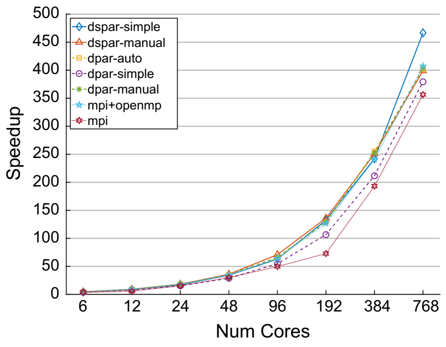
The MPI implementation of *uts* achieves a decent speedup until 96 cores, showing a very large performance drop after that point. This contrasts with the results in [12], where good scalability is achieved for a large number of cores. We believe that this is due to the fact that its results are somewhat old, hardware having noticeably evolved since then. No *manual* versions are shown for *uts-T3XXL* because they did not improve upon the *simple* one. The *automatic* version of `dparallel_recursion` hardly improves sequential execution time. The *simple* version of this skeleton achieves certain performance when using one node, but it does not improve performance beyond that point. It should be noted that using the default system limits, the executions failed due to stack overflow, a consequence of the skeleton not being prepared to deal with very deep problems as a result of being based on traditional recursion in the limited stack memory. In order to perform the executions, it was necessary to increase the maximum stack size allowed for the applications in the evaluation system, the maximum stack size of each application thread and the maximum stack size of the TBB worker threads. The new proposal addresses this issue with the use of a stack located



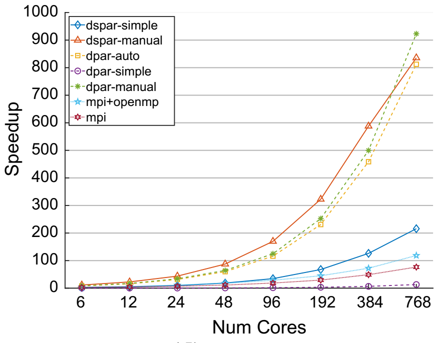
(a) *uts-T3XXL* benchmark



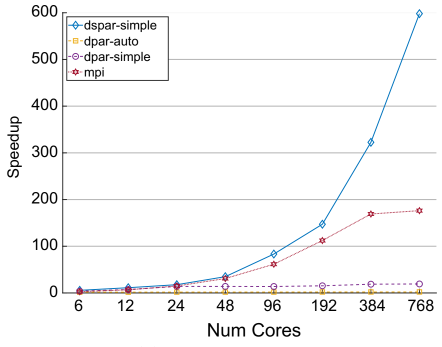
(b) *uts-T2XL* benchmark



(c) *N Queens* benchmark



(d) *fib* benchmark



(e) *topsorts* benchmark

Fig. 1 Performance metrics of the parallel implementations. *dpar* stands for `dparallel_recursion` and *dspar* for `dparallel_stack_recursion`

in the heap. In addition, it clearly outperforms the other alternatives in both *uts* benchmarks, which are among the most challenging ones due to their unbalance and characteristics. This happens especially beyond 96 cores, obtaining efficiency levels above 82% up to this limit, which drop to 70% and 50% for 384 and 768 cores, respectively. Altogether our skeleton allows reducing the runtime from

519.87 and 457.09 s in the sequential version to 1.37 and 1.07 s for *uts-T3XXL* and *uts-T2XL*, respectively, when using 32 nodes.

All the implementations achieve good performance for *N Queens*. The *simple* version of `dparallel_recursion` offers slightly lower performance, but using a *automatic* or *manual* partitioner solves this. The MPI version also shows somewhat lower performance than hybrid alternatives, which is expected due to the overhead of MPI calls in shared memory. The performance difference between the best implementations is minimal up to 384 cores, but when we use 768 cores `dparallel_stack_recursion` with a *simple* partitioner clearly achieves better speedup than the other alternatives.

The *fib* benchmark has very fine grained subproblems, which leads to low performance on all the *simple* implementations. The *automatic* partitioner of `dparallel_recursion` does a very good job with this problem, reaching performance levels close to those achieved with a custom tuned *manual* partitioner. Despite its limited dynamic load balancing capabilities, this library can achieve very good performance in this benchmark due to its ability to balance work in the initial distribution among processes according to a given cost function [16], which perfectly suits *fib*, as it is possible to accurately estimate the cost of the computation of its subproblems. The best performance is, however, almost always achieved by our `dparallel_stack_recursion` with a *manual* partitioner, begin only surpassed by a small margin by `dparallel_recursion` for 768 cores. The main reason for the better behavior of the new library up to that point despite the perfectly balanced distribution per process of `dparallel_recursion` is its much more advanced load balancing strategy in shared memory. It is also interesting that even if the *simple* partitioner is clearly a bad option for this benchmark, it allows our library to achieve better performance than MPI and MPI + OpenMP. This highlights the optimizations made in our implementation with the aim of reducing the overheads in the handling of tasks.

The `mts` MPI *topsorts* benchmark provides good results up to 192 cores, a decent but lower performance for 384 cores and a clearly bad scalability for 768 cores. `dparallel_recursion` fails to obtain good performance, and the use of the *automatic* partitioner only worsens the results, noticing the lack of a dynamic load balance system. Our solution, however, perfectly suits this unbalanced problem, always outperforming the other alternatives and scaling very well up to 768 cores, where the efficiency is still at 78%.

In addition to the great scalability achieved by our proposal when using a large number of nodes, the good performance obtained when using a single node is also notable, being very similar to that achieved by `parallel_stack_recursion` [24]. This was expected since `dparallel_stack_recursion` inherits the shared memory techniques used by that skeleton.

4.2 Programmability comparison

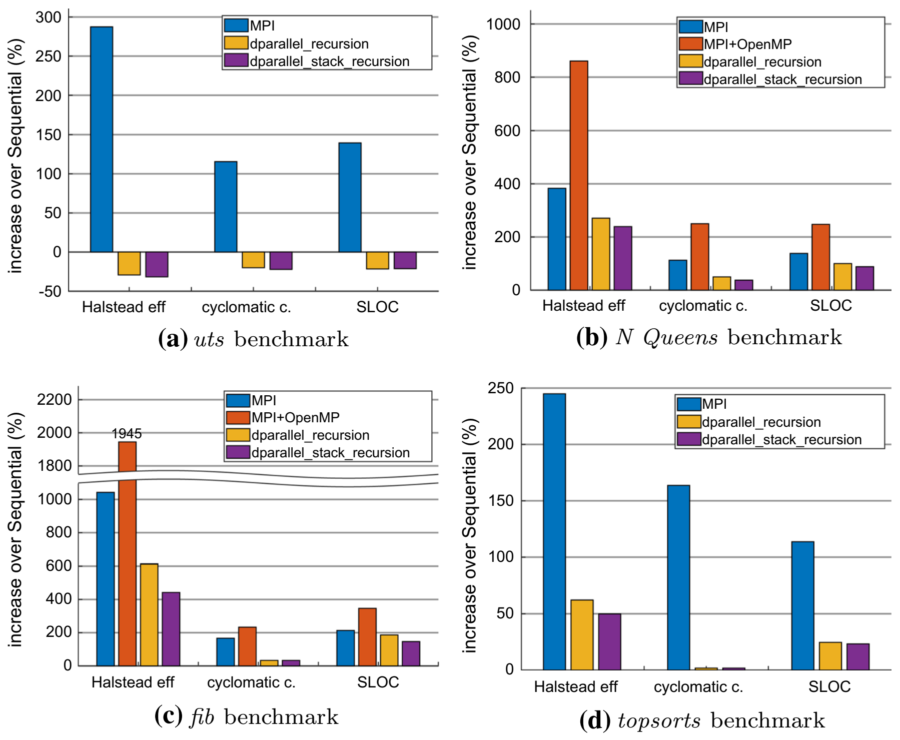
While the ideal approach for programmability analyses involves asking programmers with similar expertise to develop codes and comparing the results [28], this is seldom possible. Therefore, our study relies on objective metrics extracted from the source codes, which is a widely used alternative. Our evaluation relies on three different metrics. The first one is the number of lines of code, excluding comments and blank lines (SLOC). This value can be very sensitive to the programming style and it does not consider the length and complexity of the lines. For this reason the second metric is the Halstead programming effort [18], which estimates the cost of development considering the number of unique operands, unique operators, total operands and total operators found in the code. The last metric is the cyclomatic complexity [25], defined as *P + 1*, where *P* is the number of predicates or decision points in a program.

Table 3 shows the absolute values of the aforementioned metrics obtained for the different implementations of the *fib*, *NQueens*, *uts* and *topsorts* programs. A graphical depiction is found in Fig. 2, which shows the growth of the metrics obtained for the different parallel implementation of the benchmarks relative to the cost of the associated sequential version as a percentage. It must be noted that the MPI and MPI + OpenMP versions of *N Queens* and *fib* perform an initial simple load balancing that tries to provide a similar number of subproblems to each process. The reason for the great increase in the programmability metrics of all the parallel versions, but mostly the manual ones, in *fib* with respect to the sequential counterpart is the great simplicity of this latter code.

The *uts* benchmark is the only one in which the versions based on skeletons attain better programmability metrics than the sequential source code. The reason is that the sequential code has to manually build and manage a stack of nodes located in the heap in order to support the large recursion depths that can be experienced in this application. Since this feature is automatically provided by the skeletons, a noticeable code complexity reduction is experienced in the three metrics.

Table 3 Programming effort absolute values of different sequential and parallel implementations of several benchmark programs. *dpar* is `dparallel_recursion` and *dspar* is `dparallel_stack_recursion`. Halstead effort is expressed in thousands

	Sequential	MPI	MPI + OpenMP	dpar	dspar
	SLOC				
uts	306	733	–	240	241
N Queens	42	100	146	84	79
fib	15	47	67	43	37
topsorts	73	156	–	91	90
	Halstead effort				
uts	1110.40	4301.69	–	786.06	758.28
N Queens	65.24	315.13	626.63	241.90	221.24
fib	6.42	73.35	131.29	45.73	34.79
topsorts	205.02	707.24	–	332.57	307.32
	Cyclomatic complexity				
uts	45	97	–	36	35
N Queens	8	17	28	12	11
fib	3	8	10	4	4
topsorts	11	29	–	11	11



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

Fig. 2 Relative programming effort of *uts*, *N Queens*, *fib* and *topsorts* benchmarks compared with their respective sequential implementation

The MPI *topsorts* benchmark relies on the `mts` general framework, which allows it to achieve decent performance levels thanks to its dynamic load balancing mechanisms. Another advantage is that this reduces the programming effort compared to a totally manual implementation not relying on a pre-developed specialized library. Nevertheless, as we can see in Fig. 2d, the programmability metrics are noticeable worse than those obtained using the skeletons tested.

Finally, all the metrics are quite similar for the two skeletons tested, which was expected given the similarity in their API. In addition, they clearly provide the best metrics compared to the other alternatives tested, which is especially relevant given the good performance levels obtained. This is possible thanks to the fact that users only have to worry about adapting the D&C problem to the interface provided by the skeletons, which hide all the complex implementation that provides both the efficient divide-and-conquer implementation and, particularly in the case of the skeleton presented in this paper, the complex load balancing mechanisms supported in shared and distributed memory.

4.3 Discussion

One of the main problems with the `dparallel_recursion` framework is that it relies on recursive function calling and nested TBB tasks. Recursive function calls are also the natural performant and easy to program approach for D&C problems once the granularity for a sequential resolution has been reached in manual MPI or MPI + OpenMPI implementations. This is very stack intensive, which is very efficient, but it is problematic for deep workloads, as the stack memory is limited and thus the application can break. While proper configuration of the stack limit can avoid this problem, it requires additional effort from users, requiring both multiple steps, as there are multiple gears involved in its control, as well as a trial and error or estimation process to learn the required size in order to be successful. In contrast, in our proposal, each thread uses a separate stack allocated in the heap memory to store the subproblems that can dynamically grow as required, so that it can adapt to any needed depth requiring no particular configuration.

A downside to our skeleton compared to `dparallel_recursion` is that it lacks an *automatic* partitioner. This is not especially relevant because our design tried to minimize the overheads of the *simple* partitioner as much as possible so that it is very efficient in most cases. Still, in some very fine grained problems like *fib* this partitioner may not be enough to obtain the best performance, the *custom* available partitioner being the solution. An *automatic* partitioner would avoid the additional programming involved by the *custom* partitioner in these situations.

Another minor disadvantage of our skeleton compared to `dparallel_recursion` is the lack of a static cost-based system for workload distribution, our load balancing being totally based on dynamic mechanisms. We think that there are few unbalanced problems where it is easy and feasible to compute beforehand their cost in order to statically optimally distribute them. Furthermore, in our experience, even in those algorithms our skeleton can be quite competitive and even outperform them, as we could see in most cases in the evaluation of *fib* `dparallel_recursion` implementation, which exploits that possibility.

Something that could be more relevant in comparison with `dparallel_recursion` is the restriction to D&C problems where the reduction stage can be performed independently for each subresult. Although this restricts the scope of applicability of the skeleton, it enables very critical optimizations and sufficed to cover all the algorithms considered.

Finally, the largest degree of flexibility and potential performance is of course provided by manual implementations based on low level tools such as MPI or C++11 threads, or even somewhat higher level approaches such as OpenMP. Needless to say, programmers can develop with these tools codes that can replicate or may be even surpass the performance of our skeleton, the downside being the enormous amount of effort that they will have to put in order to program and debug them, including the non-trivial dynamic load balancing mechanisms. This is particularly true in the case that they want to exploit both shared and distributed memory parallelism in order to obtain the best possible performance.

5 Conclusions

The development of applications optimized for multi-core clusters and supercomputers requires combining parallel programming models for shared and distributed memory to achieve good efficiency. One promising technique to facilitate this task is parallel skeletons, which abstract users from this complexity while achieving high levels of efficiency, largely thanks to current very well-optimized compilers.

In this paper we present the C++ `dparallel_stack_recursion` skeleton, designed to efficiently process D&C algorithms, with a special focus on problems with large levels of recursion and/or high degree of imbalance. D&C is a widely used programming pattern that is highly parallelizable and can be applied to a large number of algorithms. The skeleton offers a simple API while making transparent to users the great complexity of distributing, processing, and balancing the tasks to perform using efficient shared and distributed memory mechanisms.

Special emphasis has been placed on keeping the skeleton as efficient as possible in order to be competitive with hand-optimized implementations. It has also been sought to grant flexibility in the configuration of the internal behavior of the library, so that users can try different options for fine-tuning. However, the default values of the library parameters provide in general good levels of performance, so that usually there is no need to test different settings for each problem unless totally optimal performance is sought. Also, an adaptive algorithm has been implemented for one of the most important parameters, which allows dynamically adjusting its value according to the state of the system.

Our evaluation shows an excellent performance of our skeleton in all the benchmarks tested. It achieves outstanding results in the most irregular and unbalanced benchmarks, providing in addition high levels of efficiency for large numbers of nodes, where even the most competitive alternatives often show a significant drop in scalability. Namely, our skeleton is on average 415.32% faster than the reasonably optimized manual MPI versions tested and 229.18% faster than the MPI + OpenMP codes manually developed. Our proposal is also 63.42% faster than the `dparallel_recursion` skeleton [16] in the *fib* and *NQueens* benchmarks, and 4350.56% faster in the *uts* and *topsorts* benchmarks, where `dparallel_recursion` is much less competitive due to its limited load balancing capabilities, which are crucial in these very unbalanced algorithms. In addition, `dparallel_recursion` required special testing and handling of the stack configuration in order to complete one of the *uts* tests due to the high level of recursion needed, while our new skeleton straightforwardly achieved excellent performance. Our library also shows very good programmability rates, which contrasts with the great effort required to implement the low-level MPI or, even worse, MPI + OpenMP solutions. On average, our library offers a 131.04% improvement in programmability metrics with respect to the MPI benchmarks, and a 155.18% improvement over the MPI + OpenMP versions. These differences are particularly relevant considering the high level of performance obtained by our proposal.

As future work we propose the extension of the library to consider other skeletal operations beyond D&C. Another interesting possibility is the development of mechanisms that allow the use of not only regular CPUs but also hardware accelerators inside the skeleton to aid the computations.

Acknowledgements This research was supported by the Ministry of Science and Innovation of Spain (PID2019-104184RB-I00 and PID2019-104834GB-I00, AEI/FEDER/EU, 10.13039/501100011033) and the predoctoral Grant of Millán Álvarez Ref. BES-2017-081320), and by the Xunta de Galicia co-founded by the European Regional Development Fund (ERDF) under the Consolidation Programme of Competitive Reference Groups (ED431C 2018/19 and ED431C 2021/30). We acknowledge also the support from the Centro Singular de Investigación de Galicia "CITIC" and the Centro Singular de Investigación en Tecnoloxías Intelixentes "CiTIUS", funded by Xunta de Galicia and the European Union (European Regional Development Fund- Galicia 2014-2020 Program), by Grants ED431G 2019/01 and ED431G 2019/04. We also acknowledge the Centro de Supercomputación de Galicia (CESGA).

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

References

1. Aldinucci M, Danelutto M, Teti P (2003) An advanced environment supporting structured parallel programming in Java. Future Gener Comput Syst 19(5):611–626
2. Ansel J, Chan C, Wong YL, Olszewski M, Zhao Q, Edelman A, Amarasinghe S (2009) PetaBricks: a language and compiler for algorithmic choice. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI -09. ACM, pp 38–49
3. Avis D, Fukuda K (1996) Reverse search for enumeration. Discrete Appl Math 65(1):21–46
4. Avis D, Jordan C (2021) mts: a light framework for parallelizing tree search codes. Optim Methods Softw 36(2–3):279–300
5. Boost.org (2021) Boost C++ libraries. http://boost.org. Accessed 28 Apr 2021
6. Ciechanowicz P, Kuchen H (2010) Enhancing Muesli's data parallel skeletons for multi-core computer architectures. In: 12th IEEE International Conference on High Performance Computing and Communications, (HPCC 2010), pp 108–113
7. Cole M (2004) Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput 30(3):389–406
8. Danelutto M, De Matteis T, Mencagli G, Torquati M (2016) A divide-and-conquer parallel pattern implementation for multicores. In: Proceedings of the 3rd International Workshop on Software Engineering for Parallel Systems, SEPS 2016. ACM, New York, NY, USA, pp 10–19
9. Dijkstra EW, Feijen WHJ, van Gasteren AJM (1983) Derivation of a termination detection algorithm for distributed computations. Inf Process Lett 16:217–219
10. Dinan J, Larkins DB, Sadayappan P, Krishnamoorthy S, Nieplocha J (2009) Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC-09. ACM, New York, NY, USA
11. Dinan J, Olivier S, Sabin G, Prins J, Sadayappan P, Tseng CW (2007) Dynamic load balancing of unbalanced computations using message passing. In: 2007 IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), pp 1–8
12. Dinan J, Olivier S, Sabin G, Prins J, Sadayappan P, Tseng C-W (2008) A message passing benchmark for unbalanced applications. Simul Model Pract Theory 16(9):1177–1189

13. Falcou J, Sérot J, Chateau T, Lapresté J-T (2006) Quaff: efficient C++ design for parallel skeletons. Parallel Comput 32(7–8):604–615
14. Farzan A, Nicolet V (2017) Synthesis of divide and conquer parallelism for loops. SIGPLAN Not 52(6):540–555
15. González CH, Fraguela BB (2010) A generic algorithm template for divide-and-conquer in multicore systems. In: Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC 2010). IEEE, Los Alamitos, CA, pp 79–88
16. González CH, Fraguela BB (2017) A general and efficient divide-and-conquer algorithm framework for multi-core clusters. Clust Comput 20(3):2605–2626
17. Gorlatch S, Cole M (2011) Parallel skeletons. In: Padua D (ed) Encyclopedia of Parallel Computing. Springer, New York, pp 1417–1422
18. Halstead MH (1977) Elements of software science. Elsevier, New York
19. Hosseini Rad M, Patooghy A, Fazeli M (2018) An efficient programming skeleton for clusters of multi-core processors. Int J Parallel Program 46:1094–1109
20. Karasawa Y, Iwasaki H (2009) A parallel skeleton library for multi-core clusters. In: Proceedings of the 2009 Internatinal Conference on Parallel Processing (ICPP'09). IEEE, Los Alamitos, CA, pp 84–91
21. Kozsik T, Tóth M, d Bozó I (2018) Free the conqueror! refactoring divide-and-conquer functions. Future Gener Comput Syst 79(P2):687–699
22. Leyton M, Piquer JM (2010) Skandium: multi-core programming with algorithmic skeletons. In Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP 2010). IEEE, Los Alamitos, CA, pp 289–296
23. Linderman Michael D, Collins Jamison D, Wang Hong , Meng Teresa H (2008) Merge: a programming model for heterogeneous multi-core systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XIII. Association for Computing Machinery, New York, NY, USA, pp 287–296
24. Martínez MA, Fraguela BB, Cabaleiro JC (2021) A parallel skeleton for divide-and-conquer unbalanced and deep problems. Int J Parallel Program 49(6):820–845
25. McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 2:308–320
26. Olivier S, Huan J, Liu J, Prins J, Dinan J, Sadayappan P, Tseng CW (2006) UTS: an unbalanced tree search benchmark. In: Languages and Compilers for Parallel Computing (LCPC 2006), pp 235–250
27. Reinders J (2007) Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly, Sebastopol, CA
28. Teijeiro C, Taboada GL, Touriño J, Fraguela BB, Doallo R, Mallón DA, Gómez A, Mouriño JC, Wibecan B (2009) Evaluation of UPC programmability using classroom studies. In: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models, PGAS '09. ACM, New York, NY, USA, pp 10:1–10:7
29. van DijkT, van de Pol JC (2014) Lace: non-blocking split deque for work-stealing. In: Euro-Par 2014: Parallel Processing Workshops. Springer, pp 206–217
30. von Koch TJKE, Manilov S, Vasiladiotis C, Cole M, Franke B (2018) Towards a compiler analysis for parallel algorithmic skeletons. In: Proceedings of the 27th International Conference on Compiler Construction, CC 2018, pp 174–184
31. Yang J, He Q (2018) Scheduling parallel computations by work stealing: a survey. Int J Parallel Program 46(2):173–197