Enhancing and Evaluating the Configuration Capability of a Skeleton for Irregular Computations

Carlos H. González
Basilio B. Fraguela
Depto. de Electrónica e Sistemas. Universidade da Coruña. Facultade de Informática, Campus de Elviña, S/N. 15071. A Coruña, Spain
{cgonzalezv, basilio.fraguela}@udc.es

***Abstract*—Although skeletons largely facilitate the parallelization of algorithms, they often provide little support for the work decomposition. Also, while they have been widely applied to regular computations, this has not been case for irregular algorithms that can exploit amorphous data-parallelism, whose parallelization in fact requires much more effort from programmers and thus benefits more from a structured approach. In this paper we improve and evaluate the configurability of a recently proposed skeleton that allows to parallelize this latter kind of algorithms. Namely, the skeleton allows to easily change critical details such as the data structures, the work partitioning algorithm or the task granularity to use. The simple procedures to choose among these possibilities and their influence on performance are described and evaluated. We conclude that the skeleton allows to conveniently explore different possibilities for the parallelization of irregular applications, which can result in substantial performance improvements.**

I. Introduction

Most of the research to facilitate the development of parallel applications has focused on regular codes, as they are much easier to tackle than irregular applications, particularly the ones that operate on pointer-based data structures presenting amorphous data-parallelism [1]. For example, in these latter applications it is much harder or even impossible to perform a priori an optimal partitioning of the work or to avoid conflicts between their parallel tasks. As a result, when the libraries of predefined parallel implementations of popular algorithms do not cover their needs, users have to parallelize these more complex codes by hand or by means of transactional memory tools, whose performance and even programmability are often not satisfactory [2].

There have been some proposals to enable the parallelization of irregular algorithms through the exploitation of useful abstractions. For example, [3] proposes a framework that relies on user annotations that describe the properties of the operations, while [4] is based on a language to describe the evolution of the working set associated to each parallel task. This paper focuses on [5], a parallel algorithmic skeleton [6][7] that allows to parallelize a large class of irregular applications with little programmer effort. This skeleton, called `parallel_domain_proc`, is based on the concept of domain, so that the elements of the irregular structure to process can be mapped on a provided domain using their properties. The domain is used by the skeleton to partition the work, by subdividing the domain, as well as to detect potential conflicts between tasks, which is achieved by testing the ownership of a new element to access by the domain associated to a parallel task.

The skeleton approach is very practical, as skeletons can be implemented by means of libraries, which facilitates their application and code reuse. Also, they provide a high level of abstraction and generality. In fact, they are higher order functions that encapsulate the communication and computation patterns found in many parallel algorithms, taking as arguments the user-defined functions that perform the computations specific to the particular algorithm to implement. Unfortunately skeletons often provide either no tools or very restrictive possibilities to control essential aspects of their execution. This can result in suboptimal performance because parallel applications are highly sensitive to implementation decisions such as the work partitioning algorithm, the degree of work decomposition, or the data structures used. For example, [8] describes some disadvantages of the highly automated template implementation of skeletons, such as taking wrong decisions that the users cannot fix.

The reduced parametrization capability of a skeleton is a small problem and can be in fact very justified in regular algorithms, as it is easier or even straightforward to derive heuristics to choose good parallelization strategies for them. This is the case of skeletons for purely data parallel operations, which can offer only block distributions [9][10][11] or even totally hide the work decomposition from the user [12][13]. Nevertheless, this is not a good approach for irregular applications, and in particular for the amorphous data-parallel ones, where the patterns of computation can widely vary between algorithms, and the best work decomposition for a given algorithm can follow different strategies for different inputs, there being besides a large variety of partitioning strategies [14]. Therefore in these applications it is critical that users can experiment with several possibilities, and using a high-level approach such as a skeleton should not preclude but facilitate this. In this regard, although there are skeletons [15][16] that allow total control of the task decomposition, they require users to programmatically specify all the details of this decomposition except for basic trivial cases like ranges, and they do not support amorphous data-parallelism.

This paper expands the work presented in [5] in many aspects. First, by describing the configuration capabilities present in the library, which allow the programmer to tune the behavior of the skeleton for the problem at hand. The main configuration options in this regard affect the task spanning policy and the data containers used. Then we present and test several graph-

oriented domain decomposition technologies, evaluating their perfomance with several benchmarks and taking into account different variations of the configurations options presented. This paper also describes some improvements in the interface of the library, compared to the previous version.

The rest of this paper is structured as follows. The next section briefly describes the class of irregular problems we target and the `parallel_domain_proc` algorithm template, while its configuration possibilities are explained in Sect. III. This is followed by their evaluation in Sect. IV and a brief discussion on related work in Sect. V. Finally, Sect. VI contains conclusions and future work.

II. A DOMAIN-BASED SKELETON FOR IRREGULAR ALGORITHMS

Algorithms that can exploit amorphous data-parallelism [1] operate on a set of the elements of an irregular data structure, which can be considered in general as a graph. This set conforms a dynamic work list, from which several threads can take workitems to process in parallel. While some algorithms present constraints with respect to the order in which the workitems must be processed, others allow any ordering. These algorithms, called unordered, naturally present more parallelism and scale better [17].

Several issues complicate the effective parallelization of irregular algorithms. This way, the processing of a workitem often gives place to new workitems that must be added to the work list. Another problem is conflicts with other parallel tasks, which can appear whenever the processing of an element requires accessing, and sometimes modifying, a portion of the graph around it, which is called the neighborhood of the element. The management of the conflicts is simpler in what are called *cautious operations* —that is, the operations that read all the neighborhood before performing any modification—, as conflicting tasks need only abort the processing and try it later. Non-cautious operations need also to roll back the modifications performed, resulting in more expensive conflicts. Another situation that must be handled is the possibility that a work list element may have been modified or even destroyed during the processing of another workitem, if it happened to be within its neighborhood.

The `parallel_domain_proc` algorithm template [5] seeks to help programmers parallelize with minimal effort this kind of algorithms in shared memory systems based on the abstraction of domain. Namely, the skeleton requires a domain on which the elements of the graph can be mapped. The skeleton uses the domain to partition the work, by assigning the processing of the elements that map to different subdomains to different parallel tasks. The domain is also used to detect potential conflicts between parallel tasks, i.e., whenever an element outside the subdomain assigned to the current task is requested during the processing of an element, such processing is aborted in order to avoid a potential conflict with the parallel task whose subdomain owns the requested element. Elements whose processing has failed for this reason are automatically gathered by the skeleton and placed in new work lists whose processing is later reattempted by new parallel tasks that are associated to larger subdomains.

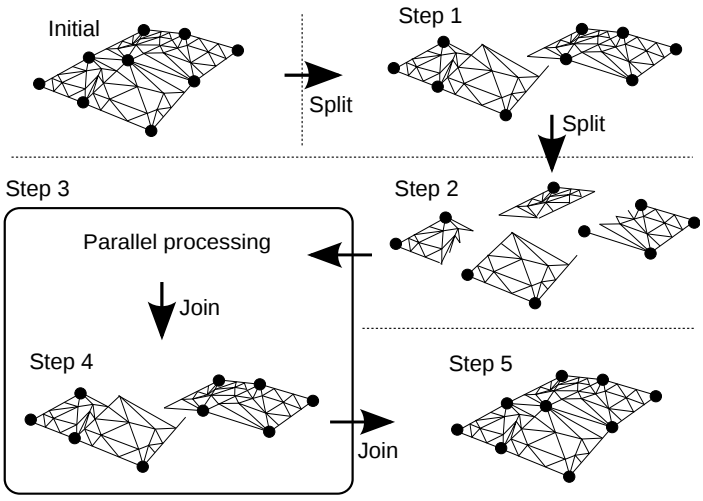

Fig. 1. Structure of the domain-based parallelization performed by `parallel_domain_proc`

As Fig. 1 shows by means of a graph in which the thicker dots represent possible limits of subdomains within the graph, `parallel_domain_proc` goes through three stages to apply this strategy. In the first one, the original domain provided by the user is recursively split in n steps until 2^n subdomains are generated. This is illustrated in steps 1 and 2 of the figure, which generate 4 bottom level subdomains. In the second stage, a parallel task to process the workitems of each low level subdomain is launched to execution. During this stage all the elements provided in the initial work list are successfully processed, except those whose neighborhood extends outside the bottom level subdomain they belong to. During the third stage, represented in the two last steps in Fig. 1, each time the tasks associated to the subdomains that descend from the same parent domain finish, the lists of the items whose processing failed in these subdomains are merged into a new work list. This work list is the input for a new parallel tasks that is associated to the parent domain. This way those elements whose neighborhood did not fit in any of the children domains but which fits within the parent domain will be successfully processed by the new task. The process is repeated for each level of subdivision, so that larger subdomains are considered in each step. Finally, as the topmost task is associated to the whole domain, it is guaranteed to finish the processing of the remaining workitems.

The implementation proposed in [5] is a C++ library that provides not only the skeleton but also all the components needed to write parallel applications in shared memory systems following this approach, from generic graph, node and edge classes, to domain and work list classes. These classes are C++ templates so that users can easily adapt them to their needs, however they are free to use any classes they want, as long as they provide the API and semantics expected by the algorithm template. The library has evolved enhancing its usability, flexibility and performance with respect to the initial proposal in [5]. The most important improvement is that work lists are now integrated inside the domain objects because this design increases locality and allows more implementation possibilities. For example the domain can build the initial work list directly as a set of local work lists from the workitems

```
1  bool is_divisible()
2  Domain& get_left()
3  Domain& get_right()
4  bool contains(Element* e)
5  void push(Element* e)
6  void push(Iterator begin, Iterator end)
```

Fig. 2. Required interface for the `Domain` class

to process when it is subdivided, avoiding the creation of an additional global work list that contains them all when this is not natural. The algorithm template takes the form

```
void parallel_domain_proc<bool redirect=false>
   (Graph, Domain, Operation)
```

where the arguments are objects that represent the graph in which the computation takes place, the domain used to classify and hold the workitems, and the operation to perform, respectively. The `redirect` template argument is a performance hint that will be discussed in the next section. Let us now briefly review the function arguments.

The graph is the only element that must be able to support concurrent modifications by parallel tasks, as all the tasks generated by the skeleton work on the same graph. It is also the only input of the skeleton on whose interface it does not have any requirement. The reason is that the runtime of the skeleton never needs to directly interact with the graph structure, as the only element that accesses it is the user provided operation.

The domain is used to partition the work, by subdividing it, and to identify the ownership of elements in order to avoid potential conflicts, by testing whether they belong to the subdomain associated to the current task or not. As a result of its first function, it also contains the worklists. The skeleton invokes the methods shown in Fig. 2 to perform these tasks. As for subdivisions, `is_divisible` indicates when a subdomain can still be subdivided. In this case, the methods `get_left` and `get_right` return the two halves of a split domain. Ownership of an element `e` by a domain is tested by the `contains` method. The last method, `push`, is used to insert individual workitems, or workitems stored between two iterators, in the local worklist of its corresponding subdomain.

Finally, the operation can be any functor, function pointer or C++11 lambda function with the form `void op(Workitem& e, Context& c)`, the arguments being the current workitem and a context object supplied by the skeleton that provides access to the graph, the task work list, and the domain associated to the task. Namely the context provides a method that allows to enqueue the new workitems generated during the operation in the work list. The domain is available so that before a graph element is accessed for the first time, its ownership by the current domain can be tested. When the operation finds an element outside the domain, it must call an abort function provided by the skeleton that delays the processing of the workitem to later stages where larger domains are used. The context object is a programmability improvement with respect to the API in [5] that reduces the number of arguments of the operation and facilitates the exploration of different types for the objects involved in the skeleton execution.

III. Skeleton behavior configuration

The proposed strategy to parallelize irregular applications allows users to explore a large space of possible concrete implementations without renouncing to the high level structured approach and good programmability enabled by the skeleton. A first configurable element is the boolean template parameter `redirect` mentioned in Sect. II, which controls the behavior of tasks that work at the bottom level of decomposition when they find an element outside their subdomain in their work list. Since the initial work lists in bottom level tasks only contain elements within their assigned subdomain, this only happens in algorithms whose processing can generate new workitems. When `redirect` is `false`, which is its default, the usual policy of delaying the processing of the workitem to later tasks that will be run with larger subdomains is applied. When it is `true`, if the task associated to the workitem subdomain at the bottom level of subdivision or a parent of it is already running, the usual policy is also followed. Otherwise, the workitem is placed in the work list of the owner bottom level task, and the task is submitted to execution. In order to enable this redirection process, the skeleton does not run tasks whose work lists are empty. While this flag controls a very specific situation, it is critical to enable the parallelization of the large class of algorithms that begin with a single node in the graph from which the processing propagates throughout the whole structure.

While `redirect` was introduced in [5] and is required to parallelize certain algorithms, in this paper we enable and evaluate other three implementation variations that are design decisions that can lead to different performance for different algorithms and inputs. The first one is the usage of different domain decomposition algorithms, which can lead to different load balance and number of conflicts among the tasks. The `parallel_domain_proc` skeleton allows to explore this possibility by writing domains with user-defined splitting strategies or simply by using or deriving from domain classes provided by the library that implement predefined partitioning schemes. The ones we have developed are:

- **Clustered Domain (cd)** tries to assign nodes that are close in terms of edges between them to the same subdomain. This is achieved by means of a clustering process in which each cluster starts from a random node and expands from it following a breadth first search. The clustering process is performed only once, generating all the bottom level domains required, and it stops when all the nodes have been assigned to subdomains. The intermediate levels of domain decomposition are generated by aggregating neighbor clusters following a bottom-up process. The fact that the decomposition does not follow the top-down approach explained in Sect. II is hidden inside the domain and it is totally oblivious to the user and the skeleton, which uses the same interface and logical process as for any other domain decomposition scheme.
- **Clustered Tree Domain (ctd)** Instead of building all the subdomains simultaneously, this alternative starts with one source node. It does a breath-first search and adds nodes to a subdomain, until half of the nodes are

in it. This splits the whole graph in two subdomains with almost the same number of elements. Then, it chooses one source node from each subdomain, and it repeats the subdivision process. This continues recursively until the number of desired subdomains is reached. This generally creates subdomains with a similar number of nodes, providing better work balance.

- **DomainND (d2d)** Very often the domains are N-dimensional spaces in which each dimension is associated to one of the data items of the graph elements and its extent is given by the range defined by the minimum and the maximum value found in the graph elements for that item. This domain is easily divisible by cyclically splitting each one of its dimensions (i.e., the i-th subdivision splits the domain(s) across dimension $i \bmod N$) until the required number of subdomains are generated. This was the only scheme tested in [5].

Another possibility is changing the number of levels of decomposition of the domains. Generating a bottom level domain, and thus a bottom level parallel task, per core available is a reasonable option. However, since the runtime of our skeleton is built on top of Intel TBB [15], it can provide improved load balancing if the domain is over-decomposed, so that the task-stealing scheduler can profit from the excess parallelism created. We have simplified the exploration of this possibility, which required manual user intervention in [5], by adding a new optional parameter to the constructor of the domains, `overdecomposition`. This parameter requests their decomposition in $2^{\texttt{i}}$ subdomains per core, the default being $\texttt{overdecomposition} = 0$, that is, a bottom level subdomain per core.

Finally, we can experiment with different data structures. For example, work lists, which are dynamic structures from which elements are being continuously removed, but which in some algorithms also dynamically receive new workitems, can play a crucial role in performance. Given these characteristics, it looks like regular (STL) lists are a good alternative for them, as they perfectly model the behavior required, and they were in fact the work lists used in [5]. Other implementations can be however considered. This way we have also built a (STL) vector-based work list that pushes new workitems at the end of the vector and which never removes the already processed workitems. Rather, it simply increases an internal pointer that indicates the first unprocessed workitem in the vector. This way this implementation trades space inefficiency for better locality and reduced allocation and deallocation cost, as vectors grow by chunks.

The programming effort required to explore these configuration variations is minimal. The `redirect` flag simply requires providing a boolean, while the level of domain decomposition is specified with a single method invocation to the domain object. Finally, the domains, work list containers, context objects, etc. are template classes, so they can accommodate any classes either provided by the library or built by the user for any of the objects involved. Of course this includes the skeleton, which is a function template that automatically adapts to the types of its arguments. This way exploring the possibilities available only requires changing the type of the associated object.

```
1  atomic<int> contracted;
2  contracted = 0;
3  ClusteredDomain<std::vector> domain(&graph);
4
5  // Initializing worklists
6  domain.push(graph->begin_nodes, graph->end_Nodes());
7
8  // Calling parallel skeleton
9  parallel_domain_proc(graph, domain, [&](BNode src, Context& ctx) {
10   BNode lightest = findLightest(graph, src);
11
12   if(lightest) {
13     ctx.check_node_and_neighbours(src, lightest);
14     contracted += graph->findEdge(src, lightest)->data();
15     ctx.push(edgeContract(graph, current, lightest));
16   }
17 });
18
19 return contracted;
```

Fig. 3. Boruvka algorithm implemented with our library

IV. Evaluation

Our evaluation is based on five algorithms with very different nature that we briefly describe here:

- **Boruvka** algorithm computes the minimal spanning tree through successive applications of edge-contraction on the input graph in an unordered fashion. In each step a random node is chosen, and it is edge-contracted with its lightest neighbor. The edge-contraction forms a new node with the union of the connectivity of the incident nodes of the two chosen nodes. If there are duplicate edges after the contraction, only the one with smallest weight is carried through in the union.
- **Independent Set** (IS) computes a maximal independent set of a graph, which is a set of nodes such that (1) no two nodes share the same edge and (2) it cannot be extended with another node. This greedy algorithm labels each node with a flag that may be in one of three states: Unmatched, Matched and NeighborMatched. All the flags begin in the Unmatched state. An unmatched node is selected from the graph. If none of its neighbors are matched, then the flag for the node is set to matched and all of its neighbors flags are set to NeighborMatched. This process continues until there are no more unmatched nodes, in which case, the nodes with matched flags are a maximal independent set.
- **Delaunay Mesh Refinement** (DMR) implements the algorithm described in [18]. A 2D Delaunay mesh is a triangulation of a set of points such that no point is inside the circumcircle of any triangle. This algorithm enforces the additional constraint of not having any angle with less than 30 degrees by operating on the triangles of the input mesh that do not fulfill this condition, which are called bad triangles. It achieves it by iteratively re-triangulating a cavity or neighborhood around each bad triangle.
- **Spanning Tree** (ST) computes the spanning tree of an unweighted graph. It starts with a random root node,

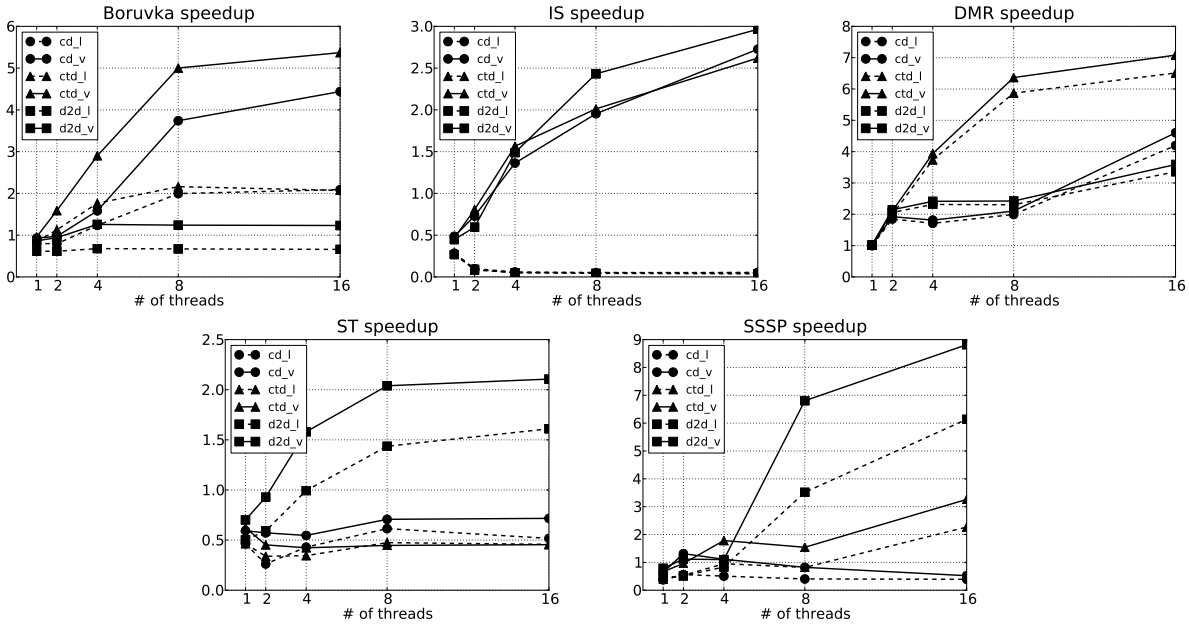

Fig. 4. Speedups using different domains and containers

and it checks its neighbors and adds to the tree those not already added. The processing continues from each one of these nodes, until all the nodes have been checked and added to the tree.

- **Single-Source Shortest Path** (SSSP) solves the single-source shortest path problem with non-negative edge weights using a demand-driven modification of the Bellman-Ford algorithm. Each node maintains an estimate of its shortest distance from the source. Initially, this value is infinity for all nodes except for the source, whose distance is 0. The algorithm proceeds by iteratively updating distance estimates starting from the source and maintaining a work list of nodes whose distances have changed and thus may cause other distances to be updated.

Figure 3 shows a possible implementation of the Boruvka algorithm using our library. In Line 3 it declares the domain it is going to use, in this case the class provided for the clustered domain explained in the preceding Section. Our library domain classes are parameterized with the type of work list to use. In this case the programmer has chosen `std :: vector`, but other containers can be used just providing their name here. Similarly, in order to explore other partitioning strategies, the user only needs to change the domain class name, using any of the other domain classes in our library, or a new one provided by the programmer. It is left to the programmer how her custom domain will work internally, as long as it fulfills the interface and semantics explained in the previous section. The initial worklist is initialized in Line 6, using the method `push`, that will store a reference to each work item from the input list in the corresponding local worklist of its subdomain. Lines 9 to 17 call the parallel skeleton with the operation implemented as a C++11 lambda function. This function uses the `Context&` `ctx` parameter to check whether the elements fall in the current subdomain using method `check_node_and_neighbours` in Line 13, which will

TABLE I. BASELINE TIMES FOR THE ALGORITHMS

Benchmark	Serial time (s)
Boruvka	17.95
IS	0.34
DMR	13.66
ST	1.61
SSSP	0.63

return inmediately the control from the user operation to the skeleton if this is not the case. The use of the context to add new elements to the work lists is also illustrated in Line 15. On average, despite the complexity of these irregular applications, our parallel versions based on `parallel_domain_proc` are just two lines of code longer than the serial ones, which gives an idea of the power of the skeleton-based approach.

The experiments were performed in a system with 2 Intel Xeon E5-2660 Sandy Bridge-EP CPUs (8 cores/CPU) at 2.2 GHz and 64 GB of RAM, using g++ 4.7.2 with optimization flag `-O3`. The graph, node and edge classes used were taken from the Galois system [3], as they were found to be more efficient than the locally developed ones used in [5] and the skeleton transparently supports any classes. The inputs were a road map of the USA with 24 million nodes and 58 million edges for Boruvka, IS and ST, a road map of New York City with 264 thousand nodes and 733 thousand edges for SSSP –both maps taken from [19]– and a mesh with 1 million triangles taken from the Galois project for DMR. Since Spanning Tree and Single-Source Shortest Path begin their operation with a single node from which the computation spreads to the whole graph, their parallelization has been performed activating the `redicted` optional feature of `parallel_domain_proc`, which is not used in the other benchmarks.

Figure 4 shows the speedups obtained with respect to a sequential execution for each benchmark using different

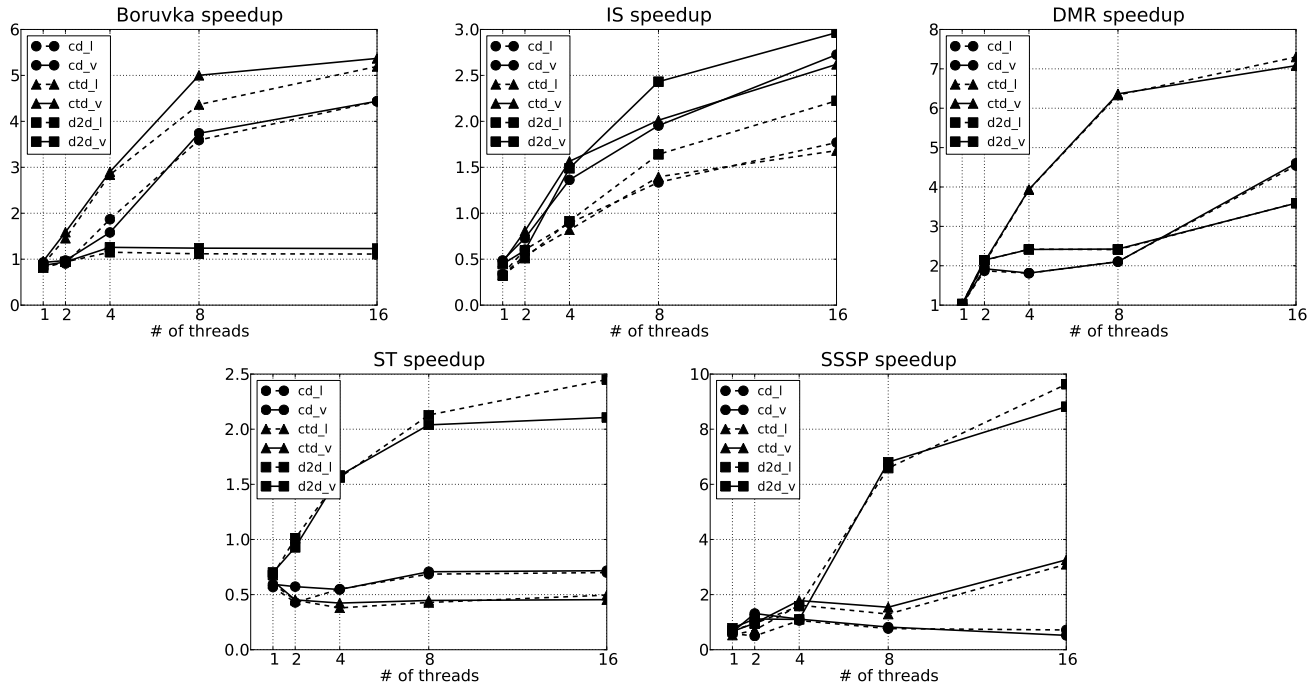

Fig. 5. Speedups using different domains and containers. In this figure, lists use our pool allocator

number of cores. The baselines, whose runtimes are shown in Table I, are pure sequential implementations, and therefore they are neither based on our skeleton nor on any other parallelization mechanism that could add any overhead, and they use the best data structures for graphs and work lists, which are vectors, as we will soon see. We tried six combinations based on the usage of the three domain decomposition strategies described in Sect. III (cd for Clustered Domain, ctd for Clustered Tree Domain and d2d for the DomainND in two dimensions) and two work list containers, namely standard (std::)lists (l) and vectors (v), both using the default C++ allocators. The executions followed the default policy of generating one bottom level task per core. The slowdown for a single core gives an idea of the overhead of the skeleton, which can be up to three times slower than the sequential version in these runs.

Vector-based work lists clearly perform better than list-based ones despite being more memory greedy, as they do not remove already processed elements (see description in Sect. III). Thanks to the reduced memory management cost and better locality, the traversal of the worklist is much more efficient when using vectors than when using lists. While in some algorithms the extra cost of lists is relatively small (DMR, or ST for some domain partitionings), and lists are in fact negligibly faster in ST with domain ctd for 8 and 16 cores, in others the disadvantage of lists is enormous. The best example is IS, where the versions that use lists obtain much worse speedup than those with vectors.

We suspected that the most important reason for the bad performance of lists is their requirement to continuously allocate and deallocate items. This operation is even more expensive in a multithreaded program, where the memory management provided by the C++ runtime is thread-safe, with

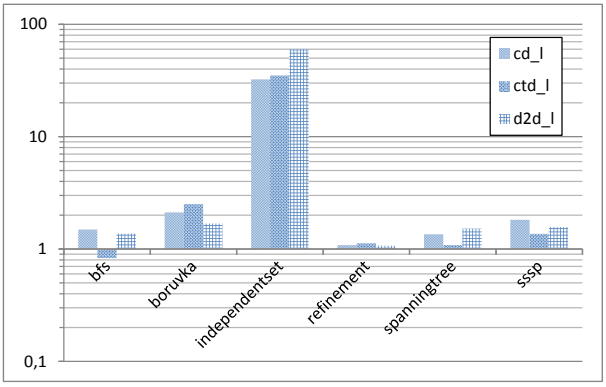

Fig. 6. Speedups of using the pool allocator with respect to the standard allocator

the associated synchronization costs. In order to prove this, we wrote a customized allocator class that acts as a pool, thus minimizing the number of invocations to the underlying thread-safe memory manager. Our allocator has a thread-safe and a faster non-threadsafe version. We could use the last one thanks to the fact that each worklist is always accessed by a single thread of the skeleton. The results are shown in Fig. 5, where lists use our allocator. Using our pool allocator greatly reduces the gap in performance between vectors and lists, up to the point of almost achieving the same speedups with both containers and in some cases, like ST and SSSP, improving them. The most significant case is IS, which did not show any speedup when using lists with the standard allocator (in Fig. 4) but now presents a reasonable scalability with the list container. Thanks to the easy configurability of our skeleton, using a list

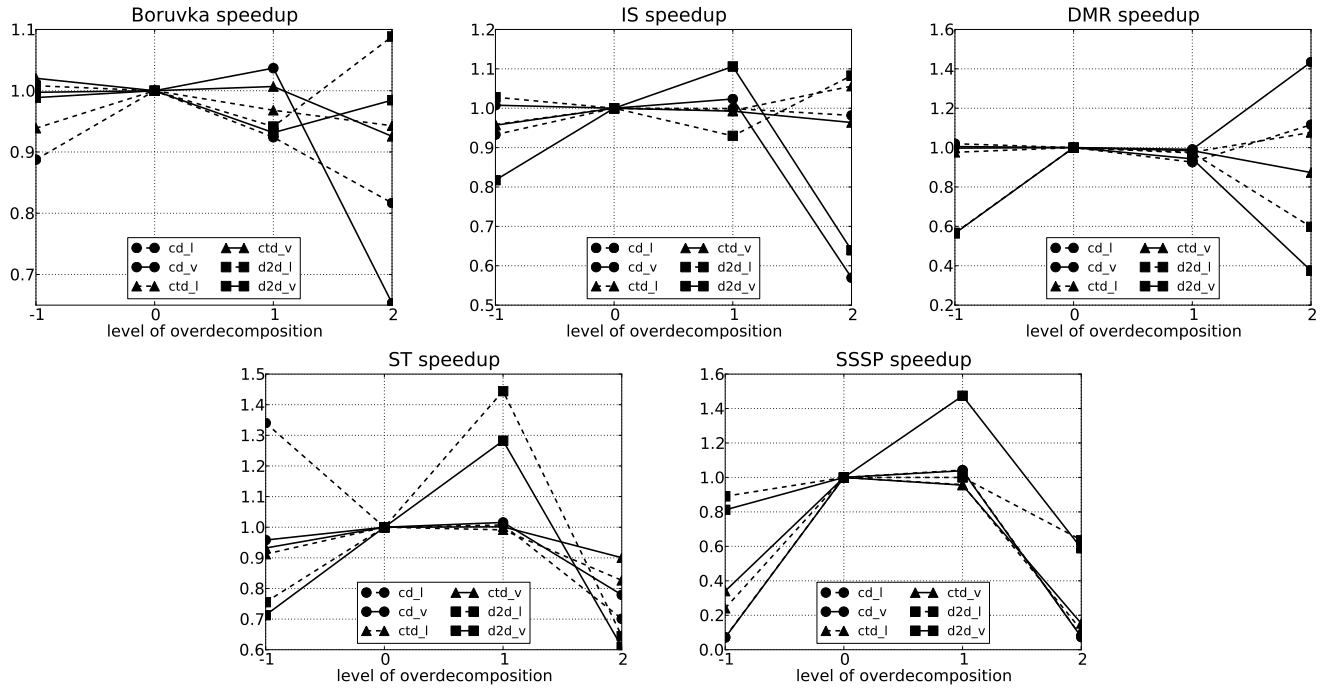

Fig. 7. Speedups using different levels of decomposition with respect to no over-decomposition in runs with 16 cores

based on our allocator was straightforward, requiring changes only in a couple of lines. The improvement in performance is shown in Fig. 6. Almost all tests give some runtime improvement, with the most dramatic case reaching a 60x speedup.

The type of domain decomposition also plays a critical role in performance, there being not a clear winner. The DomainND strategy is usually the best one for IS, ST and SSSP, while the Clustered Tree Domain offers the best performance for Boruvka and DMR. The need to allow programmers to easily test different configurations of the skeleton execution is further supported by the fact that while in some applications a decomposition algorithm is always the best across the board, this is not the case in others. For example, while for 8 and 16 cores SSSP achieves the best performance with DomainND, the best speedups for 2 and 4 cores are achieved with the Clustered Domain and the Clustered Tree Domain strategies, respectively. Similary, while DomainND is also the best strategy for IS for runs with 8 and 16 cores, it is the worst partitioning when we only have 2 cores. Also, in some specific situations all the partitioning algorithms can provide a very similar speedup. This is the case of DMR with two threads. This algorithm performs a lot of work per workitem, so it tends to scale linearly if the domains are balanced in terms of work, and the number of conflicts due to neighborhoods that extend outside the local domain are minimized. When only two domains are used, the number of conflicts is minimal due to the small number of subdomains. If in addition, the domains are built using reasonable heuristics like the ones considered in this paper, they will probably be reasonably balanced. Both ciscumstances favor the behavior observed. Another reason for the complex behavior of these applications in terms of performance is that in many of them the amount of work per workitem is highly variable, and sometimes impossible to predict in advance. This is the case of the DMR bad triangle cavities, whose extent can only be known when they are explored, and where the number of new bad triangles generated can only be known after the re-triangulation. Another example is Boruvka, whose amount of work per node is proportional to the number of edges to be contracted. This number not only depends on the initial number of edges of each node, but also on the sequence of nodes contracted before the one considered.

All in all, the best decomposition strategy depends on the application, the number of cores, and the kind of input graph, as it can favor a specific partitioning strategy [14]. Given the complexity of the subject, it is difficult to make a priori selections of the domain decomposition algorithm, and although the generic algorithms we propose can obtain good results, a better understanding of the application can allow users to create domains that can obtain better results.

The impact of over-decomposition on performance is analyzed in Fig. 7 with experiments using 16 cores. It shows the relative speedup of each one of the six possibilities tested in Fig. 4 with respect to their own execution with no over-decomposition, that is, one in which one bottom level task is created per core. As we explained in Sect. III, a level of decomposition `i` means generating $2^{\texttt{i}}$ tasks per core, thus for $\texttt{i} = 0$ the speedup in the figure is always 1. The -1 level, which generates 8 tasks, was tried to test if the lower number of task merges could improve the performance, which happened very seldom. Over-decomposition, which is very easy to apply with our skeleton, can largely improve performance, even when we consider the choices that achieved the best performance in Fig. 4. This way, d2d_v, which was the best strategy for 16 cores for IS, ST and SSSP, further increases its performance by 10%, 30% and 50%, respectively, when 2 tasks per core are generated.

Overall the skeleton achieves performance similar to that found in the bibliography for manually-tuned parallel implementations of these applications. This is the case for example for DMR in [20], although this is only a qualitative observation given the different hardware and inputs tested. Regarding the absolute speedups achieved, we must note that the performance of this kind of applications is more limited by memory latency and bandwidth than that of applications with regular access patterns and more CPU operations per input data item.

V. Related work

While we have not found any other skeleton-based approach oriented to the parallelization of this kind of applications, there are proposals with this aim. The Galois system [3] is a framework for this kind of algorithms that relies on user annotations that describe the properties of the operations. Its interface can be simplified though, if only cautious and unordered algorithms are considered. Galois has been enhanced with abstract domains [21], defined as a set of abstract processors optionally related to some topology, in contrast to our concept of set of values for a property of the items to process. Our domains are completely configurable and can be related to the problem at hand, so the programmer can use the better fitted domain for her program.

Chorus [4] defines an approach for the parallelization of irregular applications based on object assemblies, which are dynamically defined local regions of shared data structures equipped with a short-lived, speculative thread of control. Chorus follows a bottom-up strategy that starts with individual elements, merging and splitting assemblies as needed. These assemblies have no relation to property domains and their evolution, i.e., when and with whom to merge or split, must be programmatically specified by the user. We use a top-down process based on an abstract property, and only a way to subdivide its domain and to check the ownership are needed. Also, the evolution of the domains is automated by our library and it is oblivious to the algorithm code.

Partitioning has also been applied to an irregular application in [20]. Their partitioned code is manually written and it is specifically developed and tuned for the single application they study, Delaunay mesh generation. Additionally, their implementation uses transactional memory for synchronizations.

The concept of hierarchical partitioning has also been studied, for example in [22], as a means to improve data locality through several processing elements, and in the memory hierarchy of each processing element. But their solution is only applicable to regular algorithms, and as in the case of the Galois system, their domain is just an abstraction of the number of processing elements, and consecutive elements in the input data, but unlike ours, it is not defined from the properties of such data and it is not configurable.

VI. Conclusions

In this paper we have extended and evaluated the configurability of the first skeleton for amorphous data-parallel applications as far as we know. These applications offer a large number of implementation possibilities based on the use of different data structures, levels of work decomposition and work partitioning algorithms, which are richer than those of regular algorithms. The ability to easily experiment with these possibilities is very important for irregular applications because deriving heuristics to decide the best configuration for each given algorithm, input and computer to use is much more difficult than in the case of regular applications, or even impossible.

Our experience has shown that programmers can explore many alternative configurations with this skeleton with very little effort. We have also observed that the impact on performance of each implementation decision, even taken isolatedly, can be enormous and that the best alternative depends on the algorithm and the number of cores available. Also, although the generic options provided in the library provide reasonable performance, users can easily define and use their own decomposition algorithms, data structures, etc. specifically targeted to their particular problem to achieve better performance.

As for future work, we plan to enable providing more hints to improve load balancing and performance. Providing more options is also interesting, as for example, the usage of domains that rely on well-known graph partitioners [23][24] for their splitting process is a promising approach to explore the generation of balanced tasks, particularly when the user lacks information on the structure of the input. Other interesting possibilities are automating, at least partially, the parameter tuning of the skeleton, applying techniques such as those used in behavioural skeletons [25] to manage skeleton parameter sweeping at run time in case of long running applications, and developing performance models to support these tasks.

Acknowledgements

This work was supported by the Xunta de Galicia under the Consolidation Program of Competitive Reference Groups, cofunded by FEDER funds of the EU (Ref. GRC2013/055), by the Spanish Ministry of Science and Innovation, cofunded by the FEDER funds of the EU (Ref. TIN2013-42148-P), and by the FPU Program of the Ministry of Education of Spain (Ref. AP2009-4752).

References

[1] K. Pingali, M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-lojo, D. Prountzos, X. Sui, and Z. Zhong, "Amorphous data-parallelism in irregular algorithms," The Univ. of Texas at Austin, Dpt. of Computer Sciences, Tech. Rep. TR-09-05, Feb. 2009.

[2] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: Why is it only a research toy?" *Queue*, vol. 6, no. 5, pp. 46–58, 2008.

[3] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *SIGPLAN Not.*, vol. 42, no. 6, pp. 211–222, Jun. 2007.

[4] R. Lublinerman, S. Chaudhuri, and P. Cerný, "Parallel programming with object assemblies," in *OOPSLA*, 2009, pp. 61–80.

[5] C. H. Gonzlez and B. B. Fraguela, "An algorithm template for domain-based parallel irregular algorithms," *International Journal Parallel Programming*, pp. 1–20, 2013.

[6] M. Cole, *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1991.

[7] S. Gorlatch and M. Cole, "Parallel skeletons," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer US, 2011, pp. 1417–1422.

[8] M. Danelutto, "Efficient support for skeletons on workstation clusters," *Parallel Processing Letters*, vol. 11, no. 1, pp. 41–56, 2001.

[9] P. Ciechanowicz, M. Poldner, and H. Kuchen, "The Münster Skeleton Library Muesli - A Comprehensive Overview," Univ. of Münster, Tech. Rep. Working Papers, ERCIS No. 7, 2009.

[10] M. Steuwer and S. Gorlatch, "Skelcl: Enhancing opencl for high-level programming of multi-gpu systems," in *Parallel Computing Technologies*, ser. Lecture Notes in Computer Science, V. Malyshkin, Ed. Springer Berlin Heidelberg, 2013, vol. 7979, pp. 258–272.

[11] M. Danelutto and M. Torquati, "Loop parallelism: A new skeleton perspective on data parallel patterns," in *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, Feb 2014, pp. 52–59.

[12] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming," in *Infoscale*, 2006.

[13] J. Enmyren and C. Kessler, "SkePU: a multi-backend skeleton programming library for multi-GPU systems," in *4th intl. workshop on High-level parallel programming and applications*, ser. HLPP '10, 2010, pp. 5–14.

[14] A. Pothen, "Graph partitioning algorithms with applications to scientific computing," in *Parallel Numerical Algorithms*. Springer Netherlands, 1997, pp. 323–368.

[15] J. Reinders, *Intel Threading Building Blocks*, 1st ed. O'Reilly & Associates, Inc., 2007.

[16] M. Leyton and J. Piquer, "Skandium: Multi-core programming with algorithmic skeletons," in *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, Feb 2010, pp. 289–296.

[17] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," *SIGPLAN Not.*, vol. 46, no. 8, pp. 3–12, Feb. 2011.

[18] L. P. Chew, "Guaranteed-quality mesh generation for curved surfaces," in *9th Symp. on Computational Geometry*, ser. SCG '93, 1993, pp. 274–280.

[19] University of Rome, "Dimacs implementation challenge," http://www.dis.uniroma1.it/challenge9/.

[20] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe, "Delaunay triangulation with transactions and barriers," in *IEEE Intl. Symp. on Workload Characterization*, ser. IISWC '07, 2007, pp. 107–113.

[21] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew, "Optimistic parallelism benefits from data partitioning," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 233–243, 2008.

[22] J. Meng, S. Che, J. W. Sheaffer, J. Li, J. Huang, and K. Skadron, "Hierarchical domain partitioning for hierarchical architectures," Univ. of Virginia Dept. of Computer Science, Tech. Rep. CS-2008-08, June 2008.

[23] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[24] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *High-Performance Computing and Networking*. Springer, 1996, pp. 493–498.

[25] M. Aldinucci, S. Campa, M. Danelutto, P. Dazzi, D. Laforenza, N. Tonellotto, and P. Kilpatrick, "Behavioural skeletons for component autonomic management on grids." in *CoreGRID Workshop - Making Grids Work*, 2007, pp. 3–15.