A Generic Algorithm Template for Divide-and-conquer in Multicore Systems

Carlos H. González and Basilio B. Fraguela
Depto. de Electrónica e Sistemas
Universidade da Coruña
A Coruña, Spain
{*cgonzalezv, basilio.fraguela*}*@udc.es*

***Abstract*—The divide-and-conquer pattern of parallelism is a powerful approach to organize parallelism on problems that are expressed naturally in a recursive way. In fact, recent tools such as Intel Threading Building Blocks (TBB), which has received much attention, go further and make extensive usage of this pattern to parallelize problems that other approaches parallelize following other strategies. In this paper we discuss the limitations to express divide-and-conquer parallelism with the algorithm templates provided by the TBB. Based on our observations, we propose a new algorithm template implemented on top of TBB that improves the programmability of many problems that fit this pattern, while providing a similar performance. This is demonstrated with a comparison both in terms of performance and programmability.**

***Keywords*-productivity; programmability; parallel skeletons; template meta-programming; libraries; patterns**

I. Introduction

The divide-and-conquer strategy appears in many problems [1]. It is applicable whenever the solution to a problem can be found by dividing it into smaller subproblems, which can be solved separately, and merging somehow the partial results to such subproblems into a global solution for the initial problem. This strategy can be often applied recursively to the subproblems until a base or indivisible one is reached, which is then solved directly. The recursivity of an algorithm sometimes is given by the data structure on which it works, as is the case of algorithms on trees, and very often it is the most natural description of the algorithm. Just to cite a few examples, cache oblivious algorithms [2], many signal-processing algorithms such as discrete Fourier transforms, or the linear algebra algorithms produced by FLAME [3] are usually recursive algorithms that follow a divide-and-conquer strategy. As for parallelism, the independence in the resolution of the subproblems in which a problem has been partitioned leads to concurrency, giving place to the divide-and-conquer pattern of parallelism [4].

Fostered by the increase of processors available in current systems, extensive research is being made on the best ways to express parallelism. The large base of existing legacy codes, the inherent learning curve and the requirement of compiler support have traditionally made difficult the widespread adoption of new languages focused on parallelism. As for compiler directives, OpenMP [5] is well established in the field of multicore systems. While it is mainly designed to parallelize regular loops, its scope of application has been extended thanks to the addition of a task-enqueuing mechanism in its latest specification [6]. Compiler directives unfortunately require compiler support and they do not provide as much structure and functionality to applications as libraries of skeletal operations [7]. Skeletons build on parallel design patterns, which provide a clean specification to the flow of execution, parallelism, synchronization and data communications of typical strategies for the parallel resolution of problems. Divide-and-conquer has been in fact identified as one of the basic skeletons of parallel programming [8]. Libraries of parallel skeletons can be thus a very good approach to develop parallel applications thanks to the higher degree of abstraction they provide. A recent example of library that provides parallel skeletons for multicore systems is Intel Threading Building Blocks (TBB) [9], which relies on recursive decomposition and task stealing. Although there have been other libraries of skeletal operations before TBB, this library has become the most popular and widely adopted. Thus we think it is interesting to analyze how TBB adapts to the parallelization of typical classes of problems and propose ways to improve it. This way, in this paper (1) we discuss the weaknesses of TBB algorithm templates to parallelize applications that are naturally fit for the divide-and-conquer pattern of parallelism, (2) we propose a new template built on TBB to express these problems, and (3) we perform a comparison both in terms of programmability and performance.

The rest of this paper is organized as follows. The next section introduces the TBB, focusing on its ability to express the divide-and-conquer pattern of parallelism, which is exemplified with small codes in Section III. Our proposal to express this pattern of parallelism is presented in Section IV and evaluated in Section V. Related work is discussed in Section VI, followed by our conclusions in Section VII.

II. The Intel TBB library

Intel Threading Building Blocks (TBB) [9] is a C++ library developed by Intel for the programming of multithreaded applications. It provides from atomic operations and mutexes to containers specially designed for parallel operation. Still, its main mechanism to express parallelism are algorithm templates that provide generic parallel algorithms. The most important TBB algorithm templates are parallel_for and parallel_reduce, which express element-by-element independent computations and a parallel reduction, respectively. These algorithm templates have two compulsory parameters. The first one is a *range* that defines a problem that can be recursively subdivided into smaller subproblems that can be solved in parallel. The second one, called *body*, provides the computation to perform on the range. The requirements of the classes of these two objects are now discussed briefly.

The ranges used in the algorithm templates provided by the TBB must model the Range concept, which represents a recursively divisible set of values. The class must provide

- a copy constructor
- an empty method to indicate when a range is empty,
- an is_divisible method to inform whether the range can be partitioned into two subranges whose processing in parallel is more efficient than the sequential processing of the whole range,
- a splitting constructor that splits a range r in two. By convention this constructor builds the second part of the range, and updates r (which is an input by reference) to be the first half. Both halves should be as similar as possible in size in order to attain the best performance.

TBB algorithm templates use these methods to partition recursively the initial range into smaller subranges that are processed in parallel. This process, which is transparent to the user, seeks to generate enough tasks of an adequate size to parallelize optimally the computation on the initial range. Thus, TBB makes extensive usage of a divide-and-conquer approach to achieve parallelism with its templates. This recursive decomposition is complemented by a task-stealing scheduling that balances the load among the existing threads, generating and moving subtasks among them as needed.

The body class has different requirements depending on the algorithm template. This way, parallel_for only requires that it has a copy constructor and overloads the operator() method on the range class used. The parallel computation is performed in this method. parallel_reduce requires additionally a splitting constructor and a join method. The splitting constructor is used to build copies of the body object for the different threads that participate in the reduction. The join method has as input a rhs body that contains the reduction of a subrange just to the right of (i.e following) the subrange reduced in the current body. The method must update the object on which it is invoked to represent the accumulated result for its reduction and the one in rhs, that is, left.join(right) should update left to be the result of left reduced with right. The reduction operation should be associative, but it need not be commutative. It is important that a new body is created only if a range is split, but the converse is not true. This means that a range can be subdivided in several smaller subranges which are all reduced by the same body. When this happens, the body always evaluates the subranges in left to right order, so that non commutative operations are not endangered.

TBB algorithm templates have a third optional parameter, called the *partitioner*, which indicates the policy followed to generate new parallel tasks. When not provided, it defaults to the simple_partitioner, which recursively splits the ranges giving place to new subtasks until their is_divisible method returns false. Thus, with it the programmer fully controls the generation of parallel tasks. The auto_partitioner lets the TBB library decide whether the ranges must be split to balance load. The library can decide not to split a range even if it is divisible because its division is not needed to balance load. Finally, affinity_partitioner applies to algorithms that are performed several times on the same data and these data fit in the caches. It tries to assign the same iterations of loops to the same threads that run them in a past execution.

III. Divide-and-conquer with the TBB

This Section analyzes the programmability of the divide-and-conquer pattern of parallelism using the TBB algorithm templates through a series of examples of increasing complexity. This analysis motivates and leads to the design of the alternative that will be presented in the next Section.

A. Fibonacci numbers

The simplest program we consider is the recursive computation of the nth Fibonacci number. While this is an inefficient method to compute this value, our interest at this point is on the expressiveness of the library, and this problem is ideal because of its simplicity. The sequential version is

```
int fib(int n) {
  if (n < 2) return n;
  else return fib(n − 1) + fib(n − 2);
}
```

which clearly shows all the basic elements of a divide-and-conquer algorithm:

- the identification of a base case (when n < 2)
- the resolution of the base case (simply return n)
- the partition in several subproblems otherwise (fib(n - 1) and fib(n - 2))
- the combination of the results of the subproblems (here simply adding their outcomes)

The simplest implementation of this algorithm based on TBB algorithm templates, shown in Figure 1, relies on parallel_reduce and it indeed follows a recursive divide-and-conquer approach. The FibRange class, which stores in n_ the Fibonacci number to compute, provides the range object required by this template. The initial range is built in the invocation to parallel_reduce in line 36 using the constructor in lines 4-5. The template generates internally new ranges using the splitting constructor in lines 7-9, which is identified by its second dummy argument of type split.

```
1 struct FibRange {
2   int n_;
3
4   FibRange(int n)
5   : n_(n) { }
6
7   FibRange(FibRange& other, split)
8   : n_(other.n_ - 2)
9   { other.n_ = other.n_ - 1; }
10
11  bool is_divisible() const { return (n_ > 1); }
12
13  bool empty() const { return n_ < 0; };
14 };
15
16 struct Fib {
17   int fsum_;
18
19   Fib()
20   : fsum_(0) { }
21
22   Fib(Fib& other, split)
23   : fsum_(0) { }
24
25   void operator() (FibRange& range) { fsum_ += fib(range.n_); }
26
27   int fib(int n) {
28     if (n < 2) return n;
29     else return fib(n - 1) + fib(n - 2);
30   }
31
32   void join(Fib& rhs) { fsum_ += rhs.fsum_; };
33 };
34 ...
35 Fib f();
36 parallel_reduce(FibRange(n), f, auto_partitioner());
37 int result = f.fsum_;
```

Figure 1. Computation of the n-th Fibonacci number using TBB's `parallel_reduce`

This method splits the computation of the n-th Fibonacci number in the computation of the $n - 1$th and $n - 2$th numbers. Concretely, line 8 fills in the new range built to represent the $n - 2$th number, while the input `FibRange` which is being split, called `other` in the code, in updated to represent the $n-1$th number in line 9. The class is completed with the `is_disivible` and `empty` methods required, with the semantics explained in the previous Section.

The body object belongs to the `Fib` class and performs the actual computation. It stores in `fsum_` the reduction (addition) of the values of the Fibonacci numbers corresponding to the ranges it has reduced. The initial body is built in line 35 with the default constructor (lines 19-20). The template builds new bodies so that different ranges can be evaluated and reduced in parallel using the splitting constructor in lines 22-23, which simply initializes `fsum_` to 0. The `operator()` of the body (line 25) is where the Fibonacci numbers indicated by the input `FibRange` are computed by invoking the sequential `fib` method in lines 27-30. Notice that `operator()` must support any value in the input range, and not just a not divisible one (0 or 1). The reason is that the `auto_partitioner` is being used (line 36) to avoid generating too many tasks, so bodies can receive divisible ranges to evaluate. The default `simple_partitioner` would have generated a new task for every step of the recursion, which would have been very inefficient. Finally, method `join` in line 32 accumulates the results of the current body and the `rhs` one in the current one, which is simply a matter of adding their `fsum_` fields.

The `Fib` class must have a state for three reasons. First, the same body can be applied to several ranges, so it must accumulate the results of their reductions. Second, bodies must also accumulate the results of the reductions of other bodies in their `join` method. Third, TBB algorithm templates have no return type, thus body objects must store the results of the reductions. This gives place to the invocation we see in lines 35-37. The topmost `Fib` object must be created before the usage of `parallel_reduce` so that when it finishes the result can be retrieved from it.

Altogether, even when the problem suits well the TBB algorithm templates, we have gone from 4 source lines of code (SLOC) in the sequential version to 26 (empty lines and comments are are not counted) in the parallel one.

B. Tree reduction

TBB ranges can only be split in two subranges in each subdivision, while sometimes it would be desirable to divide them in more subranges. For example, the natural representation of a subproblem in an operation on a tree is a range that stores a node. When this range is processed by the body of the algorithm template, the node and its children are processed. In a parallel operation on a 3-ary tree, each one of these ranges would naturally be subdivided in 3 subtasks/ranges, one per direct child. The TBB restriction to two subranges in each partition forces the programmer to build a more complex representation of the problem so that there are range objects that represent a single child node, while others keep two children nodes. As a result, the construction and splitting conditions for both kinds of ranges will be different, implying a more complicated implementation of the methods of the range. Moreover, the `operator()` method of the body will have to be written to deal correctly with both kinds of ranges.

Figure 2 exemplifies this with the TBB implementation of a reduction on a 3-ary tree. The initial range stores the root of the tree in `r1_`, while `r2_` is set to 0 (lines 5-6). The splitting constructor operates in a different way depending on whether the range `other` to split has a single node or two (line 9). If it has a single node, the new range takes its two last children and stores that its parent is `other.r1_`. The input range `other` is then updated to store its first child. When `other` has two nodes, the new range takes the second one and zeroes it from `other`. The `operator()` of the body has to take into account whether the input range has one or two nodes, and also whether a parent node is carried.

This example also points out another two problems of this approach. Although a task that can be subdivided in $N > 2$ subtasks can always be subdivided only in two,

```
struct TreeAddRange {
  tree_t * r1_, *r2_;
  tree_t * parent_;

  TreeAddRange(tree_t *root)
  : r1_(root), r2_(0), parent_(0) { }

  TreeAddRange(TreeAddRange& other, split) {
    if(other.r2_ == 0) { //other only has a node
      r1_ = other.r1_−>child[1];
      r2_ = other.r1_−>child[2];
      parent_ = other.r1_;
      other.r1_ = other.r1_−>child[0];
    } else { //other has two nodes
      parent_ = 0;
      r1_ = other.r2_;
      r2_ = 0;
      other.r2_ = 0;
    }
  }

  bool empty() const { return r1_ == 0; }

  bool is_divisible() const { return !empty(); }
};

struct TreeAddReduce {
  int sum_;

  TreeAddReduce()
  : sum_(0) { }

  TreeAddReduce(TreeAddReduce& other, split)
  : sum_(0) { }

  void operator()(TreeAddRange &range) {
    sum_ += TreeAdd(range.r1_);
    if(range.r2_ != 0)
      sum_ += TreeAdd(range.r2_);
    if (range.parent_ != 0)
      sum_ += range.parent_−>val;
  }

  void join(TreeAddReduce& rhs) {
    sum_ += rhs.sum_;
  }
};
...
TreeAddReduce tar;
parallel_reduce(TreeAddRange(root), tar, auto_partitioner());
int r = tar.sum_;
```

Figure 2. Reduction on a 3-ary tree using TBB's `parallel_reduce`

those two subproblems may have necessarily a very different granularity. In this example, one of the two children ranges of a 3-ary node is twice larger than the other one. This can lead to a poor load balancing, since the TBB recommends the subdivisions to be as even as possible. This problem can be alleviated by further subdividing the ranges and relying on the work-stealing scheduler of the TBB, which can move tasks from loaded processors to idle ones. Still, the TBB does not provide a mechanism to specify which ranges should be subdivided with greater priority, but just a boolean flag that indicates whether a range can be subdivided or not. Moreover, when automatic partitioning is used, the library may not split a range even if it is divisible. For these reasons, allowing to subdivide in N subranges at once improves both the programmability and the potential performance of divide-and-conquer problems.

The last problem is the difficulty to handle pieces of a problem which are not a natural part of the representation of its children subproblems, but which are required in the reduction stage. In this code this is reflected by the clumsy treatment of the inner nodes of the tree, which must be stored in the `parent_` field of the ranges taking care that none is either lost or stored in several ranges. Additionally, the fact that some ranges carry an inner node in this field while others do not complicates the `operator()` of the body.

C. Traveling salesman problem

The TBB algorithm templates require the reduction operations to be associative. This complicates the implementation of the algorithms in which the solution to a given problem at any level of decomposition requires merging exactly the solutions to its children subproblems. An algorithm of this kind is the recursive partitioning algorithm for the traveling salesman in [10], an implementation of which is the tsp Olden benchmark [11]. The program first builds a binary space partitioning tree with a city in each node. Then the solution is built traversing the tree with the function

```
Tree tsp(Tree t, int sz) {
  if (t−>sz <= sz) return conquer(t);
  Tree leftval = tsp(t−>left, sz);
  Tree rightval = tsp(t−>right, sz);
  return merge(leftval, rightval, t);
}
```

which follows a divide-and-conquer strategy. The base case, found when the problem is smaller than a size `sz`, is solved with the function `conquer`. Otherwise the two children can be processed in parallel applying `tsp` recursively. The solution is obtained joining their solutions with the `merge` function, which requires inserting their parent node `t`.

This structure fits well the `parallel_reduce` template in many aspects. Figure 3 shows the range and body classes used for the parallelization with this algorithm template. The range contains a node, and splitting it returns the two children subtrees. The `is_divisible` method checks whether the subtree is smaller than `sz`, when the recursion stops. The `operator()` of the body applies the original `tsp` function on the node taken from the range.

The problems arise when the application of the `merge` function is considered. First, a stack must be added to the range for two reasons. One is to identify when two ranges are children of the same parent and can thus be merged. This is expressed by function `mergeable` (lines 22-25). The other reason is that this parent is actually required by `merge`.

Reductions take place in two places. First, a body `operator()` can be applied to several consecutive ranges in left to right order, and must reduce their results. This way, when `tsp` is applied to the node in the input range (line 36), the result is stored again in this range and an attempt to merge it with the results of ranges previously processed is done in method `mergeTSPRange` (lines 40-47). The body keeps a list `lresults_` of ranges already processed with

```
1 struct TSPRange {
2   static int sz_;
3   stack<Tree> ancestry_;
4   Tree t_;
5
6   TSPRange(Tree t, int sz)
7   : t_(t)
8   { sz_ = sz; }
9
10  TSPRange(TSPRange& other, split)
11  : t_(other.t_->right), ancestry_(other.ancestry_)
12  {
13    ancestry_.push(other.t_);
14    other.ancestry_.push(other.t_);
15    other.t_ = other.t_->left;
16  }
17
18  bool empty() const { return t_ == 0; }
19
20  bool is_divisible() const { return (t_->sz > sz_); }
21
22  bool mergeable(const TSPRange& rhs) const {
23    return !ancestry_.empty() && !rhs.ancestry_.empty() &&
24           (ancestry_.top() == rhs.ancestry_.top());
25  }
26 };
27
28 struct TSPBody {
29   list<TSPRange> lresults_;
30
31   TSPBody() { }
32
33   TSPBody(TSPBody& other, split) { }
34
35   void operator() (TSPRange& range) {
36     range.t_ = tsp(range.t_, range.sz_);
37     mergeTSPRange(range);
38   }
39
40   void mergeTSPRange(TSPRange& range) {
41     while (!lresults_.empty() && lresults_.back().mergeable(range)) {
42       range.t_ = merge(lresults_.back().t_, range.t_, range.ancestry_.top());
43       range.ancestry_.pop();
44       lresults_.pop_back();
45     }
46     lresults_.push_back(range);
47   }
48
49   void join(TSPBody& rhs) {
50     list<TSPRange>::iterator itend = rhs.lresults_.end();
51     for(list<TSPRange>::iterator it = rhs.lresults_.begin(); it != itend; ++it){
52       mergeTSPRange(*it);
53     }
54   }
55 };
56 ...
57 parallel_reduce(TSPRange(root, sz), TSPBody(), auto_patitioner());
```

Figure 3. Range and body for the Olden tsp parallelization using TBB's `parallel_reduce`

their solution. The method repetitively checks whether the rightmost range in the list can be merged with the input range. In this case, `merge` reduces them into the input range, and the range just merged is removed from the list. In the end, the input range is added at the right end of the list. Reductions also take place when different bodies are accumulated in a single one through their `join` method. Namely, left.`join`(right) accumulates in the left body its results with those of the right body received as argument. This can be achieved applying `mergeTSPRange` to the ranges in the list of results of the rhs body from left to right (lines 49-54).

IV. An algorithm template for divide-and-conquer problems

The preceding Section has illustrated the limitations of TBBs to express divide-and-conquer problems. Not surprisingly, the restriction to binary subdivisions and associative reductions impact negatively on programmability. But even problems that seem to fit well the TBB paradigm such as the recursive computation of the Fibonacci numbers have a large parallelization overhead, as several kinds of constructors are required, reductions can take place in several places, bodies must keep a state to perform those reductions, etc.

The components of a divide-and-conquer algorithm are the identification of the base case, its resolution, the partition in subproblems of a non-base problem, and the combination of the results of the subproblems. Thus we should try to enable to express these problems using just one method for each one of these components. In order to increase the flexibility, the partition of a non-base problem could be split in two subtasks: calculating the number of children, so that it need not be fixed, and building these children. These tasks could be performed in a method with two outputs, but we feel it is cleaner to use two separate methods for them.

The subtasks identified in the implementation of a divide-and-conquer algorithm can be grouped in two sets, giving place to two classes. The decision on whether a problem is the base case, the calculation of the number of subproblems of non-base problems, and the splitting of a problem depend only on the input problem. They conform thus an object with a role similar to the range in the TBB algorithm templates. We will call this object the *info* object because it provides information on the problem. Contrary to the TBB ranges, we choose not to encapsulate the problem data inside the info object. This reduces the programmer burden by avoiding the need to write a constructor for this object for most problems.

The processing of the base case and the combination of the solutions of the subproblems of a given problem are responsibility of a second object analogous to the body of the TBB algorithm templates, thus we will call it also body. Many divide-and-conquer algorithms process an input problem of type `T` to get a solution of type `S`, so the body must support the data types for both concepts, although of course `S` and `T` could be the same. We have found that in some cases it is useful to perform some processing on the input before checking its divisibility and the corresponding base case computation or recursion. Thus the body of our algorithm template requires a method `pre`, which can be empty, which is applied to the input problem before any check on it is performed. As for the method that combines the solutions of the subproblems, which we will call `post`, its inputs will be an object of type `T`, defining the problem at a point of the recursion, and a pointer to a vector with the solutions to its subproblems, so that a variable number of children subproblems is easily supported. The reason for

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

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

Figure 4. Templates that provide the pseudo-signatures for the info and body objects used by `parallel_recursion`

```
1  template<typename T, typename S, typename I, typename B, typename P>
2  S parallel_recursion(T& t, I& i, B& b, P& p) {
3    b.pre(t);
4    if(i.is_base(t)) return b.base(t);
5    else {
6      const int n = i.num_children(t);
7      S result[n];
8      if(p.do_parallel(i, t))
9        parallel_for(int j = 0; j < n ; j++)
10         result[j] = parallel_recursion(i.child(j, t), i, b, p)
11     else
12       for(int j = 0; j < n ; j++)
13         result[j] = parallel_recursion(i.child(j, t), i, b, p);
14     }
15     return b.post(t, result);
16   }
17 }
```

Figure 5. Pseudocode of the `parallel_recursion` algorithm template

requiring the input problem is that, as we have seen in Sections III-B and III-C, in many cases it has data which are not found in any of its children and which are required to compute the solution.

Figure 4 shows templates that describe the info and body objects required by the algorithm template we propose. The info class must be derived from class `Arity < N >`, where `N` is either the number of children of each non base subproblem, when this value is a constant, or the identifier `UNKNOWN` if there is not a fixed number of subproblems in the partitions. This class provides a method `num_children` if `N` is a constant. As for the body, it can be optionally derived from the class `EmptyBody < T, S >`, which provides shell (empty) methods for all the methods a body requires. Thus inheriting from it can avoid writing unneeded methods.

Figure 5 shows the pseudocode for the operation of the algorithm template we propose, which is called `parallel_recursion` for similarity with the names of the standard TBB algorithm templates. Its arguments are the representation of the input problem, the info object, the body object, and optionally a partitioner that defines the policy to spawn parallel subtasks. The figure illustrates the usage of all the methods in the info and body classes, `I` and `B` in the figure, respectively. Contrary to the TBB algorithm templates, ours returns a value which has type `S`, the type of the solution. A specialization of the template allows a return type `void`.

From the pseudocode we see that `Info :: is_base` is not the exact opposite of the `is_divisible` method of the TBB ranges. TBB uses `is_divisible` to express divisibility of the range, but also whether it is more efficient to split the range and process the subranges in parallel than to process the range sequentially. Even if the user writes `is_divisible` to return true for all non base cases, the library can ignore it and stop partitioning even if it indicates divisibility if the `auto_partitioner` is used. For these reasons, the `operator()` of a standard body should be able to process both base and non base instances of the range. This makes it different from the `Body :: base` method in Figure 4, which processes the problem if and only if `Info :: is_base` is true, as line 4 in Figure 5 shows.

The decision on whether the processing of the children subproblems is made sequentially or in parallel is up to the partitioner in `parallel_recursion` (lines 8-14 in Figure 5). The behavior of the partitioners is as follows. The `simple_partitioner` generates a parallel subtask for each child generated in every level of the recursion, very much as it does in the standard TBB templates.This is the default partitioner. The `auto_partitioner` works slightly different from the standard templates. In them this partitioner can stop splitting the range, even if `is_divisible` is true, in order to balance optimally the load. In `parallel_recursion` this partitioner also seeks to balance load automatically. Nevertheless, it does not stop the recursion in the subdivision of the problem, but just the parallelization in the processing of the subtasks. This way the problem is split always that `Info :: is_base` is false. Finally, we provide a new partitioner called `custom_partitioner` which takes its decision on parallelization based on an optional `Info :: do_parallel(const T& t)` method supplied by the user. If this method returns `true`, the children of `t` are processed in parallel, otherwise they are processed sequentially.

Let us now review the implementation of the examples discussed in Section III using this proposal.

A. Examples of usage

Figure 6 shows the code to compute the n-th Fibonacci number using `parallel_recursion`. Compared to the 26 SLOC of the implementation based on `parallel_reduce` in Figure 1, this implementation only has 9. This code has the virtue that it not only parallelizes the computation, it even makes unnecessary the original sequential `fib` function thanks to the power of `parallel_recursion` to fully express problems that are solved recursively.

The addition of the values in the nodes of a 3-ary tree, which required 41 SLOC in Figure 2, is expressed using 9 SLOC with `parallel_recursion` in Figure 7. In fact, the version in Figure 2 is a bit longer because it uses the sequential function `TreeAdd`, not shown, to perform

```
struct FibInfo : public Arity<2> {
  bool is_base(const int i) const { return i <= 1; }

  int child(const int i, const int c) const { return c - i - 1; }
};

struct Fib: public EmptyBody<int, int> {
  int base(int i) { return i; }

  int post(int i, int * r) { return r[0] + r[1]; }
};
...
int result = parallel_recursion<int> (n, FibInfo(), Fib(), auto_partitioner());
```

Figure 6. Computation of the n-th Fibonacci number using `parallel_recursion`

```
struct TreeAddInfo : public Arity<3> {
  bool is_base(const tree_t *t) const { return t == 0; }

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

struct TreeAddBody : public EmptyBody<tree_t *, int> {
  int base(tree_t * t) { return 0; }

  int post(tree_t * t, int *r) { return r[0] + r[1] + r[2] + t->val; }
};
...
int r = parallel_recursion<int> (root, TreeAddInfo(), TreeAddBody(),
          auto_partitioner());
```

Figure 7. Reduction on a 3-ary tree using `parallel_recursion`

the reduction of a subtree in the `operator()` of the body. This function is not needed by the implementation based on `parallel_recursion`, which can perform the reduction just using the template.

Our last example, the traveling salesman problem implemented in the tsp Olden benchmark, is parallelized with `parallel_recursion` in Figure 8. The facts that the `post` method that combines the solutions obtained in each level of the recursion is guaranteed to be applied to the solutions of the children subproblems generated by a given problem and that this parent problem is also an input to the method simplify extraordinarily the implementation. Concretely, the code goes from 45 SLOC using `parallel_reduce` in Figure 3 to 12 using `parallel_recursion`.

V. Evaluation

We now compare the implementation of several divide-and-conquer algorithms using `parallel_recursion`, the TBB algorithm templates and OpenMP both in terms of programmability and performance. OpenMP is not directly comparable to the skeleton libraries, as it relies on compiler support. It has been included in this study as a baseline that approaches the minimum overhead in the parallelization of applications for multicores, since the insertion of compiler directives in a program usually requires less restructuring than the definition of the classes that object-oriented skeletons use. This way the comparison of standard TBB and the `parallel_recursion` skeletons with respect to the OpenMP version helps measure the relative effort of parallelization that both kinds of skeletons imply.

```
struct TSPInfo: public Arity<2> {
  static int sz_;

  TSPInfo(int sz )
  { sz_ = sz; }

  bool is_base(const Tree t) const { return (t->sz <= sz_); }

  Tree child(int i, const Tree t) const { return (i == 0) ? t->left : t->right;}
};

struct TSPBody : public EmptyBody<Tree, Tree> {
  Tree base(Tree t) { return conquer(t); }

  Tree post(Tree t, Tree * results) { return merge(results[0], results[1], t); }
};
...
parallel_recursion<Tree>(root, TSPInfo(sz), TSPBody(), auto_patitioner());
```

Figure 8. Olden tsp parallelization using `parallel_recursion`

The algorithms used in this evaluation are the computation of the n-th Fibonacci number from Section III-A (fib), the merge of two sorted sequences of integers into a single sorted sequence (merge), the sorting of a vector of integers by quicksort (qsort), the computation of the number of solutions to the N Queens problem (nqueens) and four tree-based Olden benchmarks [11]. The first one is treeadd, which adds values in the nodes of a binary tree. It is similar to the example in Section III-B, but since the tree is binary, it is much easier to implement using TBB's `parallel_reduce`. The sorting of a balanced binary tree (bisort), a simulation of a hierarchical health system (health), and the traveling salesman problem (tsp) from Section III-C complete the list.

Table I provides the problem sizes, the number of subproblems in which each problem can be divided (arity) and whether the combination of the results of the subproblems is associative or not, or even not needed. It also shows the value of the metrics that will be used in Section V-A to evaluate the programmability for a baseline version parallelized with OpenMP. All the algorithms but nqueens and health are naturally expressed splitting each problem in two, which fits the TBB algorithm templates. Nqueens tries all the locations of queens in the i-th row of the board that do not conflict with the queens already placed in the top $i-1$ rows. Each possible location gives place to a child problem which proceeds to examine the placements in the next row. This way the number of children problems at each step varies from 0 to the board size. Health operates on a 4-ary tree, thus four is its natural number of subproblems. The subnodes of each node are stored in a vector. This benefits the TBB algorithm templates, as this enables using as range a `blocked_range`, which is a built-in TBB class that defines a one-dimensional iteration space, ideal to parallelize operations on vectors.

Table I
BENCHMARKS USED

Name	Description	Arity	Assoc	SLOC	Effort	V
fib	recursive computation of 43rd Fibonacci number	2	Yes	37	31707	5
merge	merge two sorted sequences of 100 million integers each	2	-	62	143004	6
qsort	quicksort of 10 million integers	2	-	71	119908	11
nqueens	N Queens solution count in 14×14 board	var	Yes	82	192727	17
treeadd	add values in binary tree with 24 levels	2	Yes	92	179387	9
bisort	sort balanced binary tree with 22 levels	2	No	227	822032	20
health	2000 simulation steps in 4-ary tree with 6 levels	4	No	346	1945582	34
tsp	traveling salesman problem on binary tree with 23 levels	2	No	370	2065129	40

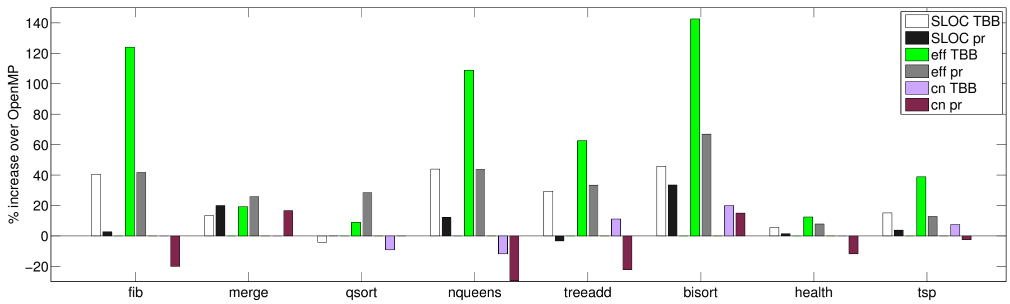

Figure 9. Productivity statistics with respect to the OpenMP baseline version of TBB based (TBB) and `parallel_recursion` based (pr) implementations. SLOC stands for source lines of code, eff for the programming effort and cn for the cyclomatic number.

A. *Programmability*

The impact of the use of an approach on the ease of programming is not easy to measure. In this section three quantitative metrics are used for this purpose: the SLOC (source lines of code excluding comments and empty lines), the programming effort [12], and the cyclomatic number [13]. The SLOC is more dependent on the user programming style than the other two metrics. The programming effort is a function of the number of unique operands, unique operators, total operands and total operators found in a program. The operands correspond to the constants and identifiers, while symbols or combinations of symbols that affect the value or ordering of operands constitute the operators. According to [12] the programming effort metric calculated from these values is approximately proportional to the programming effort required to implement an algorithm. Finally, the cyclomatic number [13] is $V = P + 1$, where P is the number of decision points or predicates in a program. The smaller V, the less complex the program is.

Figure 9 shows the SLOC, programming effort and cyclomatic number increase over an OpenMP baseline version for each code when using a suitable TBB algorithm template (TBB) or `parallel_recursion` (pr). The statistics were collected automatically on each whole application globally. Had we tried to isolate manually the portions specifically related to the parallelization, the advantage of `parallel_recursion` over TBB would have often grown to the levels seen in the examples discussed in the preceding sections. We did not do this because sometimes it may not be clear whether some portions of code must be counted as part of the parallelization effort or not, so we measured the whole program as a neutral approach.

The mostly positive values indicate that, as expected, OpenMP has the smallest programming overhead, at least when counted with SLOCs or programming effort. Nevertheless, `parallel_recursion` is the global winner for the cyclomatic number. The reason is that many of the conditionals and loops (they involve conditions to detect their termination) found in divide-and-conquer algorithms are subsumed in the `parallel_recursion` skeleton, while the other approaches leave them exposed in the programmer code more often. `parallel_recursion` requires fewer SLOC, effort and conditionals than the TBB algorithm templates in all the codes but merge and qsort. According to the programming effort indicator, programs parallelized with the TBB templates require 64.6% more effort than OpenMP, while those based on `parallel_recursion` require on average 33.3% more effort than OpenMP. This is a reduction of nearly 50% in relative terms. Interestingly, the situation is the opposite for merge and qsort, in which the average effort overhead over the OpenMP version is 13.4% for the codes that use `parallel_for` and 30.1% for the `parallel_recursion` codes. These are the only benchmarks in which there is no need to combine the result of the solution of the problems: they only require the division in subproblems that can be solved in parallel. They are also the two benchmarks purely based on arrays, where the concept of Range around which the TBB algorithm templates are designed fits better. Thus when these conditions hold, we may prefer to try the standard TBB skeletons.

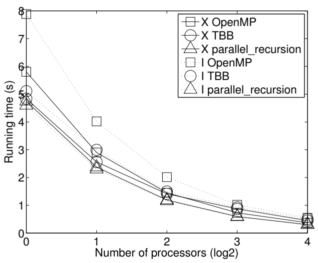

Figure 10. Performance of fib

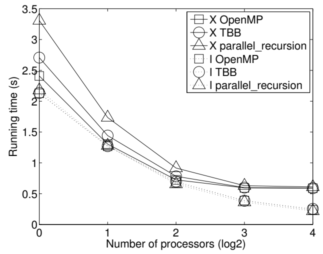

Figure 11. Performance of merge

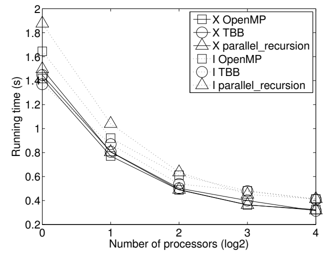

Figure 12. Performance of quicksort

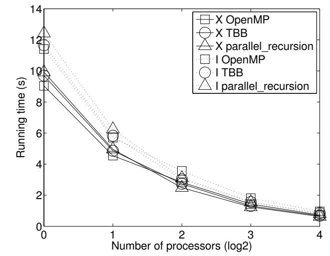

Figure 13. Performance of nqueens

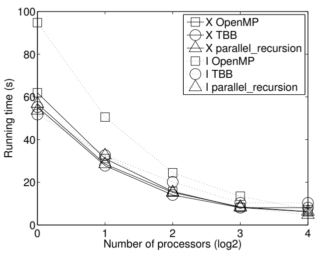

Figure 14. Performance of treeadd

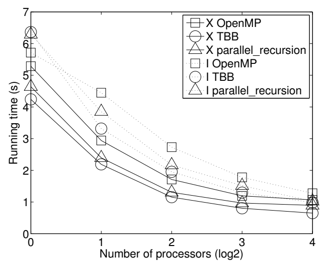

Figure 15. Performance of bisort

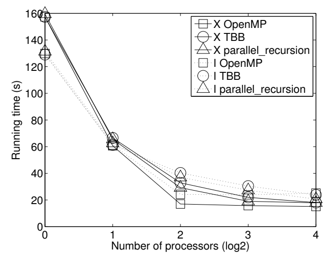

Figure 16. Performance of health

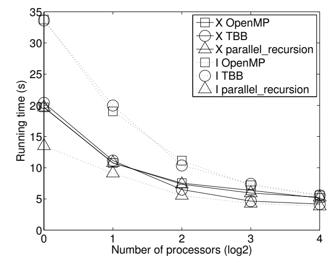

Figure 17. Performance of tsp

B. Performance

The performance of these approaches is compared now using the Intel icpc compiler V 11.0 with optimization level O3 in two platforms. One is a server with 4 Intel Xeon hexa-core 2.40GHz E7450 CPUs, whose results are labeled with X. The other is an HP Integrity rx7640 server with 8 dual-core 1.6 GHz Itanium Montvale processors, whose results are labeled with I. Figures 10 to 17 show the performance of the three implementations of the benchmarks on both systems. Automatic partitioning is used in the standard TBB and `parallel_recursion` based codes. Fib and nqueens use little memory and thus scale well in both systems. The scaling of the other benchmarks is affected by the lack of memory bandwidth as the number of cores increases, particularly in our Xeon-based system, whose memory bandwidth is up to 5 times smaller than that of the rx7640 server when 16 cores are used. This results in small to null performance improvements when we go from 8 to 16 cores in this system. Benchmark health is also affected by very frequent memory allocations with `malloc` that become a source of contention due to the associated lock.

Since `parallel_recursion` is built on top of the TBB one could expect its codes to be slower than those based on `parallel_for` or `parallel_reduce`. This is not the case because our template is built directly on the low level task API provided by the TBB. Also, it follows different policies to decide to spawn tasks and has different synchronization and data structure support requirements as we have seen. This makes it possible for `parallel_recursion` to be competitive with the native TBB version, and even win systematically in benchmarks like fib. In other benchmarks like merge in the Xeon or quicksort in the Itanium `parallel_recursion` is non negligibly slower than the standard TBB when few cores are used, but the difference vanishes as the number of cores increases. The slowdowns of `parallel_recursion` in these two codes are due to operations repeated in each invocation to `child` to generate a subproblem. Generating at once a vector with all the children tasks could avoid this. Evaluating this option is part of our future work. The behavior of tsp in the Itanium is due to the compiler, as with g++ 4.1.2 with the same flags the performance of all the implementations is very similar. Over all the benchmarks and numbers of cores, on average `parallel_recursion` is 0.3% and 19.7% faster than the TBB algorithm templates in the Xeon and in the Itanium, respectively. If tsp is ignored in the Itanium due to its strange behavior, `parallel_recursion` advantage is still 9% in this platform. Its speedups over OpenMP are 2.5% and 30.5% in the Xeon and the Itanium, respectively; 21.4% in the latter without tsp.

Finally, we also experimented with the partitioners that allow to control manually the subdivision of tasks in the runs with 16 cores. With the best parameters we found, standard TBB based codes were on average 6.5% faster than the `parallel_recursion` based ones in the Xeon, while `parallel_recursion` continued to lead the performance in the Itanium platform by 8%, or 2.4% if tsp is not counted.

VI. Related work

While TBB is probably the most widespread library of skeletal operations nowadays, it is not the only one. The eSkel library [7] offers parallel skeletal operations for C on top of MPI. Its API is somewhat low-level, with many MPI-specific implementation details. Since C is not object oriented, it cannot exploit the advantages of objects for encapsulation, polymorphism, and generic programming where available, as is the case of C++. A step in this direction was Muesli [14], which is also oriented to distributed memory, being centered around distributed containers and skeleton classes that define process topologies. Muesli relies on runtime polymorphic calls, which generate potentially large overheads. This way [15] reports 20% to 100% overheads for simple applications. Lithium [16] is a Java library oriented to distributed memory that exploits a macro data flow implementation schema instead of the more usual implementation templates, but it also relies extensively on runtime polymorphism. Quaff [17] avoids this following the same approach as the TBB and our proposal, namely relying on C++ template metaprogramming to resolve polymorphism at compile time. Quaff's most distinctive feature is that it leads the programmer to encode the task graph of the application by means of type definitions which are processed at compile time to produce optimized message-passing code. As a result, while it allows skeleton nesting, this nesting must be statically defined, just as type definitions must be. Thus tasks cannot be generated dynamically at arbitrary levels of recursion and problem subdivision as the TBBs do. This is quite sensible, since Quaff works on top of MPI, being mostly oriented to distributed memory systems. For this reason its `scm` (split-compute-merge) skeleton, which is the most appropriate one to express divide-and-conquer algorithms, differs substantially from the TBB standard algorithm templates and `parallel_recursion`.

VII. Conclusions

We have reviewed the limitations of the skeletal operations of the TBB library, a recent popular tool, to express the divide-and-conquer pattern of parallelism. This analysis has led us to design a new algorithm template that overcomes these problems. We have also implemented it on top of the task API of the TBB so that it is compatible with all the TBB library and it benefits from the load balancing of the TBB scheduler. Our implementation uses template metaprogramming very much as the standard TBB in order to provide efficient polymorphism resolved at compile time.

The examples in the paper and an evaluation using several productivity measures indicate that our algorithm template indeed adapts to a wide variety of problems and it can often improve substantially the programmer productivity when expressing divide-and-conquer parallelism. As for performance, our proposal is on average somewhat faster than the TBB templates when automatic partitioning is used. There is not a clear winner when the granularity of the parallel tasks is adjusted manually.

As future work, we want to evaluate small variations in the interface and to develop an extension that is suitable for distributed memory systems.

Acknowledgements

This work was supported by the Xunta de Galicia under project INCITE08PXIB105161PR and the Ministry of Science and Innovation, cofunded by the FEDER funds of the European Union, under the grant TIN2007-67536-C03-02. We also acknowledge the Centro de Supercomputación de Galicia (CESGA) for the usage of its supercomputers.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS '99: Procs. 40th Annual Symp. on Foundations of Computer Science*, 1999, p. 285.

[3] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn, "The Science of Deriving Dense Linear Algebra Algorithms," *ACM Trans. Math. Softw.*, vol. 31, no. 1, pp. 1–26, Mar. 2005.

[4] T. Mattson, B. Sanders, and B. Massingill, *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[5] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., 2001.

[6] OpenMP Architecture Review Board, "OpenMP Program Interface Version 3.0," May 2008.

[7] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Computing*, vol. 30, no. 3, pp. 389–406, 2004.

[8] ——, *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1991.

[9] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007.

[10] R. M. Karp, "Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane." *Math. of Operations Research*, vol. 2, no. 3, pp. 209–224, 1977.

[11] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting Dynamic Data Structures on Distributed-Memory Machines," *ACM Trans. on Programming Languages and Systems*, vol. 17, no. 2, pp. 233–263, March 1995.

[12] M. H. Halstead, *Elements of Software Science*. Elsevier, 1977.

[13] McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, 1976.

[14] 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.

[15] H. Kuchen, "A skeleton library," in *Proc. 8th Intl. Euro-Par Conf. on Parallel Processing*, 2002, pp. 620–629.

[16] M. Aldinucci, M. Danelutto, and P. Teti, "An advanced environment supporting structured parallel programming in Java," *Future Gener. Comput. Syst.*, vol. 19, no. 5, pp. 611–626, 2003.

[17] J. Falcou, J. Sérot, T. Chateau, and J.-T. Lapresté, "Quaff: efficient C++ design for parallel skeletons," *Parallel Computing*, vol. 32, no. 7-8, pp. 604–615, 2006.