A Divide-and-Conquer Parallel Pattern Implementation for Multicores

Marco Danelutto, Tiziano De Matteis, Gabriele Mencagli, and Massimo Torquati
Department of Computer Science, University of Pisa, Italy
{marcod, dematteis, mencagli, torquati}@di.unipi.it

Abstract

Divide-and-Conquer (DaC) is a sequential programming paradigm which models a large class of algorithms used in real-life applications. Although suitable to extract parallelism in a straightforward way, the parallel implementation of DaC algorithms still requires some expertise in parallel programming tools by the programmer.

In this paper we aim at providing to non-expert programmers a high-level solution for fast prototyping parallel DaC programs on multicores with minimal programming effort.

Following the rationale of *parallel design pattern* methodology, we design a C++11-compliant template interface for developing parallel DaC programs. The interface is implemented using different back-end frameworks (i.e. OpenMP, Intel TBB and FastFlow) supporting source code reuse and a certain amount of performance portability.

Experiments on a 24-core Intel server show the effectiveness of our approach: with a reduced programming effort the programmer easily prototypes parallel versions with performance comparable with hand-made parallelizations.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent Programming

Keywords High-level parallel patterns, Divide and Conquer

1. Introduction

Divide and conquer (or *divide et impera*) (briefly DaC in the sequel) is a well-known problem solving strategy that divides the original problem into smaller sub-problems each one recursively solved. Sub-problems solutions are properly combined in order to obtain the solution of the original problem. The fundamental idea is that the solution of sub-problems and the combination of their results is much more easier than finding the solution of the initial problem directly. The strategy consists in three steps applied at each level of recursion [7]:

- a ***divide*** phase in which the problem is subdivided into a number of smaller and easier to solve sub-problems;
- a ***conquer*** phase where the sub-problems are solved recursively, while simple problems (not large enough) can be solved directly without further recursion;
- a ***combine*** step in which the solutions of the sub-problems are merged to obtain the solution of a bigger problem.

A vast set of problems in different application domains can be solved with this method: typical examples are sorting algorithms (e.g., merge- and quick-sort), numerical problems (e.g., matrix multiplication, Fast Fourier Transform), computational geometry (e.g., convex hull calculation), computational biology (e.g., block alignment of sequences), preference queries (e.g., skyline queries) and many others. From the parallel computing standpoint, DaC algorithms have been widely investigated in the past owing to their intrinsic nature to be suitable for parallel computations. In fact, the executions on different sub-problems are usually independent and can be performed by different processors in parallel. Furthermore, DaC algorithms tends to be *cache oblivious* [3], i.e. they are able to take advantage of both shared and private caches for any cache size available on the target machine. Despite their pronounced tendency for parallelism, parallel implementations of DaC algorithms require a certain expertise in parallel programming and a good knowledge of parallel programming tools and frameworks to obtain the desired level of performance on today's multi-/many-core architectures.

The aim of this work is to provide to non-expert parallel programmers a high-level tool for fast prototyping of parallel DaC algorithms implementations with a reduced time-to-solution and, thus, a minimal programming effort. To achieve this goal, we will rely on the basic parallel pattern definition as provided by Mattson et al. in [19]. In this way the programmer is only involved in identifying the key components of the algorithms, and "forced" to reason directly in terms of the sequential problem instance (which is naturally the one closer to the programmer's thinking). On the underlying parallel implementation level, our high-level parallel pattern exploits different runtime systems (notably OpenMP [23], Intel TBB [25] and FastFlow [8]) in a way that is completely transparent to the programmer. The final outcome is that our parallel pattern supports the non-expert parallel programmer in the development, with a reduced effort, of good parallel implementations of DaC problems having performance at least comparable to the ones that can be produced by an expert programmer with a hand-made parallelization.

The paper contributions can be summarized as follows:

- we propose a high-level definition of a parallel template for DaC problems. The pattern is written using the C++11 syntax and code style. The interface is intentionally simple, straightforward to use, and does not require any particular parallel expertise;
- the proposed pattern can be implemented in various back-end environments (or runtimes) to target different execution scenarios. We propose its implementation for shared-memory architectures only (multicores) using the OpenMP compiler directives, the Intel TBB framework and the FastFlow parallel programming library;
- the experimental section will analyze the implementations with different runtimes and will compare the obtained performance against those obtained from more optimized hand-made parallel versions of the same code.

The rest of this paper is organized as follows. Sect. 1 shows the interface of the high-level parallel pattern for DaC. Sect. 3 describes the different implementations of the pattern with various runtimes. Sect. 4 shows a large set of experimental evaluations. Finally, Sect. 5 shows the related work and Sect. 6 states the conclusion of this work.

2. The Divide-and-Conquer Pattern

The starting point of our work is inspired by the *parallel design patterns* proposed in [19]. They describe solutions to exploit parallelism in recurrent problems. Authors recognized that the divide and conquer strategy is employed in many sequential algorithms, and they motivated the need of a high-level parallel pattern for DaC problems by observing that while parallelism in these algorithms is obvious, the implementation techniques required to efficiently exploit it are not always straightforward.

Our proposed C++11 implementation provides to the programmer a simple and effective interface (called `DAC` in the following) to easily derive a parallel implementation from a divide-and-conquer algorithm. The same interface can be implemented in different parallel programming frameworks (we will study OpenMP, Intel TBB and FastFlow; see Section 3). This allows the exploitation of different frameworks and target architectures while maintaining the same source code. Unlike [9, 22] we do not consider the possibility to automatically detect parallelizable regions and automatically produce code for the pattern instantiation under particular conditions.

To instantiate the pattern the programmer needs to provide the data type of the input problem and the type of the output result as template arguments. In the following we will refer to them as `ProblemType` and `ResultType` respectively; in the description we consider them different types, although in specific cases they can coincide. To be utilized in the interface, the types must provide a default constructor. In addition, the programmer must provide the input object and the output object where the final result will be stored. These parameters are easily identifiable from the sequential code and, as indicated by Mattson et al. [19], they are sufficient to fully characterize the algorithm behavior:

- a `divide` function takes as input a problem and produces a set of sub-problems. It has the following interface:

```
void divide(const ProblemType &p,
            std::vector<ProblemType> &subps)
    ;
```

  The function must fill the `subps` vector passed by reference. The use of the vector container enables the divide function to generate a set of sub-problems with different cardinality, i.e. not only two, at each recursion level;
- a `baseCase` solution for the base case problem. It takes as input a problem and produces the corresponding result. Both of them are passed by reference:

```
void base(const ProblemType &p,
          ResultType &res);
```

- a `combine` function that builds the result of a problem starting from the solution of its sub-problems:

```
void combine(std::vector<ResultType>& subres
    ,
             ResultType &res);
```

- a `condition` to test whether a problem is a base case problem:

```
bool cond(const ProblemType &p);
```

In the sequential algorithm the recursion continues until the sub-problems can be solved directly. In a parallel program it could be more convenient to stop recursion at an optimal level of computation granularity, and solve the problem sequentially. This may result in a better use of the memory hierarchy. However, it may also limit the number of concurrent activities. The optimal *cutoff* size depends both on the specific problem and on target architecture as studied in recent research work [13]. As a future extension of this work, we planned to add at the pattern interface level a flag that allows to relieve the user from the burden of manually set the optimal cutoff value, and at the same time, to tell the lower-level runtime to automatically derive the cutoff technique that is best suited for the application as proposed in [11].

The different functional parameters must be provided as `std::function` i.e. they can be any callable C++ object such as function pointer, lambda expression or function objects. An example of instantiation of the `DAC` interface is shown in Listing 1.

```
// functions aliases
using divide_f_t =
  std::function<void(const ProblemType&,
                     std::vector<ProblemType>&)
    >;
using combine_f_t =
  std::function<void(std::vector<ResultType>&,
                     ResultType&)>;
using base_f_t =
  std::function<void(const ProblemType&,
                     ResultType&)>;
using cond_f_t =
  std::function<bool(const ProblemType&)>;

// D&C pattern constructor
template <typename ProblemType,
          typename ResultType>
DAC(const divide_f_t& divide,
    const combine_f_t& combine,
    const base_f_t& base, const cond_f_t& cond,
    const ProblemType& p, ResultType& res,
    int par_degree = available_cores())
```

Listing 1: The `DAC` interface.

The programmer provides the reference to the starting problem (`p`), i.e. the input of the original algorithm, and a reference to the final result (`res`) where the result will be stored at the end of the parallel processing. Furthermore, the programmer may indicate also the desired number of parallel executors (`par_degree`) that by default is set to the number of available CPU cores. The call to the `compute()` method on the `DAC` object will start the computation. Once returned, the result will be found in the `res` variable.

2.1 Usage Examples

The advantage of using a high-level pattern-based approach is that all the parameters required to instantiate the pattern can be easily derived from the sequential algorithm. In addition, all the details concerning the parallel implementation are completely hidden to the programmer. As an exemplification, in the following we will show how to express two common DaC problems: the computation of the Fibonacci numbers and the standard Mergesort algorithm.

Fibonacci numbers. As a first example we consider the naïve computation of the n^{th} number of the Fibonacci sequence (with $n > 0$). Based to its definition, this can be computed using a DaC approach as shown in Listing 2.

```
1 unsigned int Fib(unsigned int n) {
2    if ( n <= 2 ) return 1;
3    return Fib(n-1) + Fib(n-2);
4 }
```

Listing 2: Sequential Fibonacci.

To instantiate the `DAC` pattern we have to specify the `Problem` and `Result` types, which are in this case just `unsigned integers`. The aforementioned functional parameters are immediately derived by the sequential algorithm definition. The `divide` function simply splits the problem of computing the n^{th} Fibonacci number in the problems of computing the $(n-1)^{th}$ and $(n-2)^{th}$ (line 3 in Listing 2). The base case (line 2) regards the computation of the first two numbers of the sequence. Finally, the `combine` function sums up the partial results to generate the desired Fibonacci number (line 3). The code to instantiate the pattern is reported in Listing 3; all the functions are expressed as C++11 lambda expressions.

```
using uint = unsigned int;
DAC<uint, uint> dac(
    // divide
   [](const uint &p,std::vector<uint> &subps) {
      subps.push_back(p-1);
      subps.push_back(p-2);},
   // combine
   [](std::vector<uint>& res,
      uint &ret) {   ret=res[0]+res[1];},
   // sequential base case
   [](const uint &p, uint &res) { res=1;},
   // condition
   [](const uint &p) { return (p<=2); },
   n, res);
dac.compute(); //starting the computation
```

Listing 3: `DAC` Fibonacci number computation.

Once executed the result will be available in the `res` variable.

Mergesort. Listing 4 shows a sequential implementation of the Mergesort algorithm.

```
void merge_sort(std::vector<int>::iterator left,
                std::vector<int>::iterator right
    ){
  if(right-left>1) {
    std::vector<int>::iterator m=
            left+(right-left)/2;
    merge_sort(left, m);
    merge_sort(m,right);
    merge(left,right,m);
  }
}
void merge(std::vector<int>::iterator left,
           std::vector<int>::iterator right,
13            std::vector<int>::iterator mid) {
14    int size=right-left;
15    std::vector<int> tmp(size);
16    std::vector<int>::iterator i=left, j=mid;
17    //merge in order
18    for(int k=0;k<size;k++) {
19       if(i<mid && (j>=right || *i<=*j)) {
20          tmp[k]=*i; i++;
21       } else {
22          tmp[k]=*j; j++;
23       }
24    }
25    //copy back
26    std::copy(tmp.begin(),tmp.end(),left);
27 }
```

Listing 4: Sequential Mergesort algorithm

To use the DAC interface the programmer defines a Problem type that encapsulates the information needed to describe the problem: in this case the two iterators indicating the vector portion to be sorted are sufficient.

```
struct Problem {
   vector<int>::iterator left;
   vector<int>::iterator right;
};
```

The same definition can be used as the Result type.

In the divide phase the problem of sorting an n-element sequence is divided into the problem of sorting two sub-sequences of $n/2$ elements (line 4 of Listing 4). The combine phase is essentially managed by the merge function (defined in lines 11-27). In this case the programmer has to properly build the problem and the result data structures. Furthermore, in the parallel implementation we have to consider the case that eventually the sub-problems generated by the divide function become small enough that they can be computed sequentially. Therefore, it could be more convenient to stop the recursion before reaching the base case of the sequential algorithm. This is captured in the cond function: the sequential version is used when the remaining size of the vector to be sorted has length smaller than a given *cut-off* parameter, e.g., 1000 or 2000 elements. The code of the DAC version of the mergesort is reported in Listing 5.

```
void divide(const Problem &p,
            std::vector<Problem> &subps) {
   std::vector<int>::iterator mid=p.left+(p.
    right-p.left)/2;
   Problem a;
   a.left=p.left;
   a.right=mid;
   subps.push_back(a);
   Problem b;
   b.left=mid;
   b.right=p.right;
   subps.push_back(b);
}
void seq(const Problem &p, Result &ret) {
   ret=p;
   std::sort(ret.left,ret.right);;
}
void merge(std::vector<Result>& res,Result& ret)
     {
    int size=res[1].right-res[0].left;
    std::vector<int> tmp(size);
    std::vector<int>::iterator i=res[0].left;
    std::vector<int>::iterator mid=res[0].right;
   std::vector<int>::iterator j=mid;
   //merge in order
   for(int k=0;k<size;k++) {
      if(i<mid && (j>=res[1].right || *i<=*j)) {
         tmp[k]=*i; i++;
      } else {
         tmp[k]=*j; j++;
      }
   }
   //copy back
   std::copy(tmp.begin(),tmp.end(),res[0].left);
   //build the result
   ret.left =res[0].left;
   ret.right=res[1].right;
}
bool cond(const Problem &p) {
   return (p.right-p.left<=CUT_OFF);
}
```

Listing 5: DAC for the Mergesort algorithm.

f After that, the programmer can istantiate a DAC object as shown in the previous example.

3. Pattern Implementations

The proposed pattern interface can be implemented using different back-end frameworks for parallel programming on multicores. This allows the programmer to exploit the different frameworks and, more in general, different target architectures without requiring code rewriting and by having a reasonable expectation of the actual performance of the parallel code, i.e. the so-called *performance portability*.

A generic divide and conquer algorithm can be defined as shown in Listing 6:

```
void DACAlgo(const ProblemType &p,
             ResultType &ret)  {
   if(!cond(op)) { //not the base case
      //divide
      std::vector<ProblemType> ps;
      divide(p,ps);
      std::vector<ResultType> res(ps.size());
      //conquer recursive phase
      for(size_t i=0;i<ps.size();i++)
         DACAlgo(ps[i],res[i]);
      //combine results
      combine(res,ret);
      return;
   }
```

```
    seq(p,ret); //base case
}
```

Listing 6: DAC algorithm.

We provide three different implementations of the DAC pattern:

- DAC_OPENMP: an implementation of the pattern that uses the OpenMP framework based on *pragma*-based preprocessor directives;
- DAC_TBB: an implementation based on the Intel Threading Building Block (TBB) library;
- DAC_FF: an implementation entirely based on the FastFlow framework.

In the following we will describe in detail the specific implementation strategies used in developing the different versions of the pattern.

3.1 OpenMP

OpenMP is a specification for a set of compiler directives, library routines, and environment variables for shared-memory parallelism in C, C++ and Fortran programs [23].

Starting from version 3.0, OpenMP has introduced a support for *task parallelism* as a way to express units of work with dependencies. Tasks found immediate application in recursive algorithms that were traditionally difficult or inefficient to express using classical OpenMP constructs [2]. The programmer has to specify where the tasks are defined, their code, if they share context and when they synchronize with each other. Since OpenMP is based on compiler directives, this is done by using proper *pragmas* in the source code.

In the case of DaC algorithms it is relatively easy to recognize where these pragmas should be placed. Owing to the wide OpenMP diffusion, we decided to provide a ready-to-use implementation of our pattern. Internally, we adopted the basic algorithm of Listing 6 by creating tasks at each recursive call: independent calls can go through the recursion tree in parallel but they have to be synchronized before performing the combine phase, in order to be sure that all the partial results have been computed. Therefore, the recursive calls to DACAlgo (line 10 in Listing 6) are defined as tasks using the proper pragma, and, before performing the combine phase, a synchronization is placed to wait for their completion.

How tasks will be scheduled and executed by the different thread of a parallel region will depend on the particular OpenMP implementation. As pointed out in [12], in OpenMP the use of cutoff thresholds is important to limit the overhead of task creation and management. For this reason this parameter plays a fundamental role in achieving good performance when using the OpenMP back-end.

3.2 TBB

The Intel *Thread Building Blocks* (TBB) library [25] aims at supporting programmers in developing parallel programs for multicore without having to specify directly thread creation and management. TBB provides the implementation of a set of parallelism design patterns useful for defining parallel programs. When these patterns do not match the programmer needs, TBB provides proper interfaces for accessing its tasking abstraction which allows the definition of arbitrary parallel programs. A Divide and Conquer pattern does not appear among the ready-to-use TBB high-level patterns, therefore we rely on generic task graphs to express this kind of problems in TBB.

As pointed out by the official documentation [17], if performance is a major concern it is more convenient to use the low-level interface (i.e. the tbb::task class) for expressing parallel programs. This interface represents a non-user friendly approach and requires a certain level of expertise in using the TBB features. For this reason, we decided to provide an implementation for the TBB library that internally exploits the task feature through this interface. The rationale followed in the development is very similar to the one used for OpenMP: recursive calls spawn independent tasks that are synchronized before the combine phase.

Tasks are scheduled to a set of threads for execution. To achieve better load balancing among worker threads, TBB uses a non-preemptive cooperative scheduling based on work stealing, inspired by the task scheduler of Cilk5 [14]. This enables efficiently handling of very irregular divide and conquer problems, i.e. computations that generate highly variable sub-problem sizes such as in the quick-sort and quick hull problems.

3.3 FastFlow

FastFlow [1, 8] is a structured parallel programming environment for multicore implemented in C++ on top of POSIX threads. The framework presents a layered design: it provides high-level parallel patterns to the programmers, implemented on top of core skeletons (*pipeline* and *farm*), easily composable and nestable to obtain complex computations. FastFlow does not natively offer a divide and conquer skeleton. To implement it we exploit the *macro-data flow* pattern (ffMDF) [4]. This pattern implements a dynamic macro-dataflow interpreter processing direct acyclic graphs (DAG) of tasks generated at run-time. Its runtime is in charge of scheduling tasks to the processing units as they become available (fireable), i.e. all input data-dependencies are satisfied.

The definition of tasks and of their dependencies requires a certain level of knowledge of the algorithm and the framework itself. The proposed implementation handles this work in a transparent way, relieving the programmer from this burden. For divide and conquer algorithms the DAG can be identified by considering the recursion tree. As already pointed out in the previous sections, the recursive calls and combine phases can be represented as tasks. Differently than OpenMP and TBB, the `ffMDF` pattern does not offer primitives for task synchronization. To guarantee the correctness, proper DAG dependencies are enforced among the graph nodes representing the sub-problems and the ones representing the partial results.

4. Experiments

In this section we describe a set of experiments of the `DAC` pattern implemented with the three different parallel frameworks. We have chosen three different and classic divide and conquer algorithms: the merge- and quick-sort algorithms and the Strassen algorithm for matrix multiplication. Besides being very well-known, these problems fully characterize the variety of possible situations that may occur in divide and conquer algorithms, thus they represent a minimal yet representative set of benchmark applications useful to test the pattern. Specifically:

- merge-sort is characterized by a divide phase with a negligible computation cost, while most of the running time is spent in the combine phase (i.e. the merge of two ordered sub-arrays);
- the quick-sort algorithm is symmetric with respect to merge-sort. In fact, here the combine phase is totally absent and the entire work is essentially performed in the divide phase;
- in the Strassen algorithm both the divide and combine phases represent relatively coarse-grain computations. In addition, differently from the previous two cases, at each recursion step the problem is divided into seven sub problems, rather than two.

As a first evaluation, we will show how the different backend implementations perform with these problems. The programs are tested with different sizes of the input data: merge- and quick-sort are tested with arrays having size equal to $10M$, $20M$, $50M$ and $100M$ (integer elements). Strassen is tested using square dense matrices of double elements having sizes equal to $1K \times 1K$, $2K \times 2K$, $4K \times 4K$ and $8K \times 8K$.

These experiments are aimed at highlighting an important result of this paper: *the use of our high-level pattern interface by a non-expert programmer in parallel programming allows an easy development of parallel DaC programs with performance comparable to the one achieved by a expert programmer using directly the low-level mechanisms offered by the underlying framework.* To show this result, we compare the pattern-based parallelizations with *hand-made* or third-party OpenMP and TBB parallelizations written using the low-level mechanisms (i.e. explicit task creation and synchronization) provided by those frameworks. The source code of all the implementations discussed in this section are freely available. [1].

The target platform used for the experiments is a dual socket Intel Xeon Ivy Bridge running at 2.40GHz with 24 cores (12 per socket) and 64GB of RAM. Each core has 32KB private L1d, 256KB private L2 and 30MB shared L3. The *Turboboost* and *Hyperthreading* facilities have been disabled. For the compilation we use `gcc 4.8.3` with the `-O3` optimization flag.

Concerning the libraries, for OpenMP we use the `gcc` implementation (interface `v. 3.1`), for the Intel TBB and FastFlow library we use the version `4.1` and `2.1.2` respectively. All the measurements are performed multiple times and average values are shown: in general, the difference between the standard deviation and the average values reported is less than 3.5%.

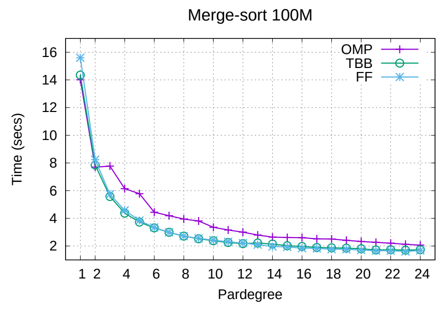
4.1 Comparison between Various Back-Ends

In this first evaluation we compare the behavior of the different backends. Figure 1 shows the completion time of the three applications with the biggest input data instances using different parallelism degrees. With parallelism degree equal to one, the FastFlow back-end usually has a completion time higher with respect to the OpenMP and TBB versions, however it approaches the other two solutions as long as the parallelism degree increases.

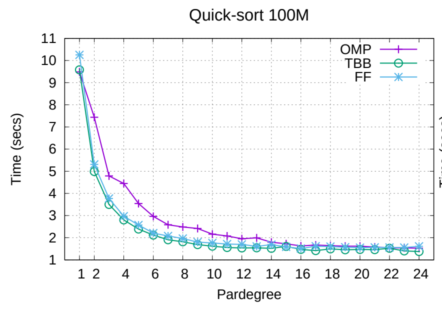
In general, all the implementations behave similarly, with OpenMP slightly slower when low values of the parallelism degree are used. The merge- and quick-sort plots show a plateau when we use a parallelism degree equal or greater to $12 - 14$, which means that the performance does not steadily increase if we use more cores. For Strassen we are able to reduce the completion time using higher degrees of parallelism. The main reason for this different behavior is due to the fact that the sorting problems are essentially memory-bound, hence the overall scalability of the parallel implementation using many cores is bounded by the memory bandwidth provided by the machine; the Strassen algorithm is a more compute-bound application and better scalability can be achieved.

Figure 2 compares the best time achieved by the different implementations for different problem sizes. In the merge-sort case FastFlow obtains the best time, while OpenMP obtains the worst one. The TBB implementation results the best option for the quick-sort problem: we argue that this is due to the TBB task scheduler, which is known to be able to efficiently handle situations of unbalanced computations. This is exactly the situation characterizing quick-sort, where the divide phases can produce sub-problems with substantially different sizes. Finally, for Strassen TBB exhibits the best completion time, while OpenMP and FastFlow perform similarly and slightly slower than TBB.

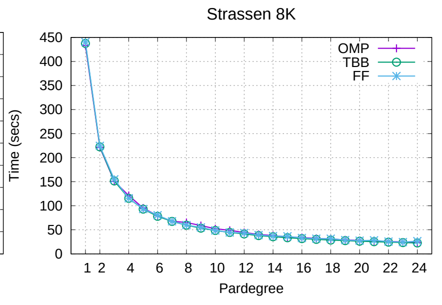
[1] The source code can be downloaded at: `https://github.com/ParaGroup/DAC`



(a) Merge-sort

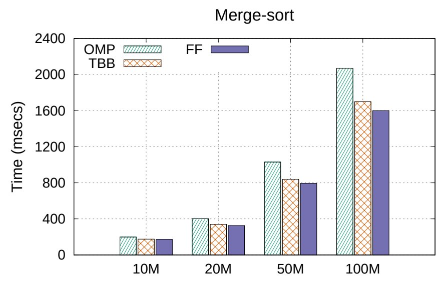


(b) Quick-sort

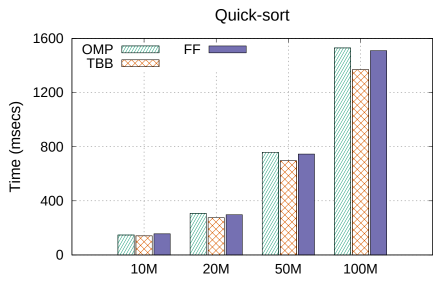


(c) Strassen

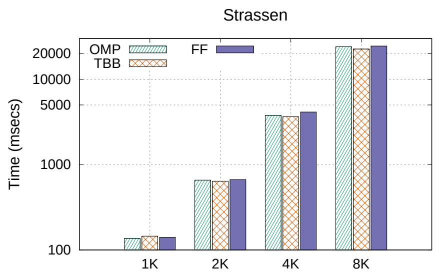
Figure 1: Behavior of the different back-end implementations of the DAC pattern for the merge-sort, quick-sort and Strassen problems with the biggest tested problem size. The plots show the completion time with different parallelism degrees (Pardegree).



(a) Merge-sort



(b) Quick-sort



(c) Strassen

Figure 2: Comparison of the different backends with different problem sizes. The Strassen plot is shown in logarithmic scale.

All in all, these experiments show that the three back-ends provide similar performance figures for the DAC pattern.

4.2 Comparison with Hand-Made Versions

In this section we show the comparison between the pattern-based and hand-made/third party parallelizations of the considered applications. For the quick-sort and the Strassen problems we use a hand-made parallelization developed by ourselves. For merge-sort we use the stable-sort version provided by Intel [2]: for this evaluation, the same algorithm has been implemented using the `DAC` pattern maintaining the same interfaces and data types.

Figure 3 shows the ratio of the completion time of the pattern-based implementation to the completion time of the hand-made version for the considered applications. A value greater than 1 indicates a performance loss, while a value lower than 1 shows that the pattern-based version is faster than the hand-made counterpart.

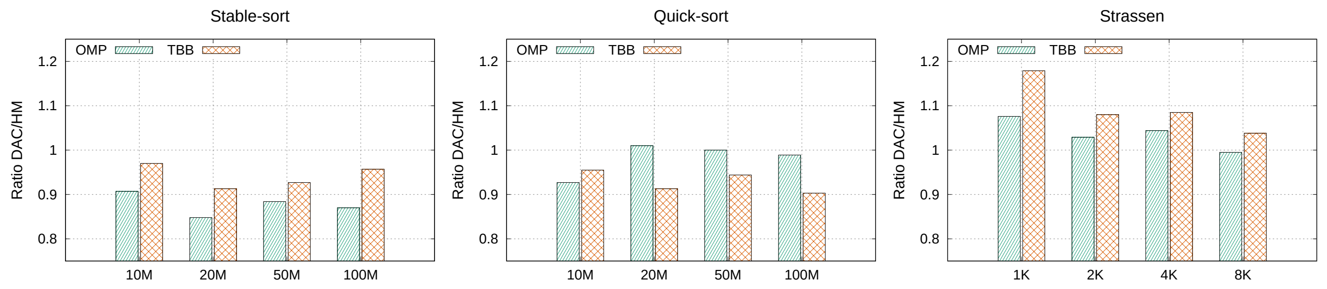
For the stable-sort and quick-sort problems, the implementation written with the high-level pattern interface achieves better performance for any input data size. This is especially true for the stable-sort problem using the OpenMP back-end. This may be due to the compiler optimizations and would require a complete profiling of the applications to be fully explained. We left this analysis in our future investigations in order to confirm our intuition. In Strassen, we experience a performance loss especially using the TBB back-end. The main reason of this difference derives from the parallelization of the matrix multiplication base case (through a `parallel for`) in the hand-made implementation version, while in the pattern-based implementations the base case is handled sequentially. In all the considered scenarios, the experienced performance loss is less than 17%, and the performance benefit, when present, is about 7% for the considered problems on average.

5. Related Work

Over the years divide-and-conquer algorithms have attracted the attention of the parallel community, due to their wide diffusion in different domains and their attractive properties in terms of parallelism and memory hierarchy exploitation [3, 15, 16, 24].

Authors in [15] observe that `parallel-for` and `parallel-reduce` algorithmic templates offered by the Intel TBB library could be used to parallelize this kind of prob-

[2] https://software.intel.com/en-us/articles/a-parallel-stable-sort-using-c11-for-tbb-cilk-plus-and-openmp



(a) Stable-sort (b) Quick-sort (c) Strassen

Figure 3: The plots show the ratio of the completion time of the pattern implementations over the time of the corresponding hand-made parallelization of the same problem.

lems. However, this needs an important code restyling and refactoring phase and it is actually limited to the possibility of having only binary divisions of the problem and associative reductions. To overcome this limitation they propose an additional algorithm template for TBB (`parallel recursion`). Its implementation relies on the `parallel-for` but in a more pragmatic way. The programmer has to define two objects (`Info` and `Body`), which encapsulate the information needed for the parallel execution. The execution is based on the `parallel-for`, while the combine phase is performed by a sequential call instead of using the `parallel-reduce`.

The performance achieved by this implementation is similar to, and in some cases better than, the one obtained with the native `parallel-for` implementation. This solution is strictly bounded to the use of the Intel TBB framework and still requires an important code refactoring. Moreover, the authors do not consider other methods that Intel suggests to parallelize divide-and-conquer problems using its TBB library [17], which are based on the more general and powerful concept of *task parallelism* [19]. According to these methods, recursive calls are managed as different tasks and a fork-join approach is used to handle their life-cycle. As stated in the documentation [17], to obtain the best performance it is suggested to use the low-level interface to the tasking capabilities of TBB, which requires a certain level of expertise to be used efficiently. In our TBB implementation (Sect. 3) this solution has been adopted *for the backend implementation of our high-level pattern template*, hiding all the low-level programming aspects to the user.

Another approch, which follows the high-level patternized view of parallel code, consists in using parallel patterns to express parallelism. In these frameworks the patterns for recurrent problems are provided to programmers as *algorithmic skeletons*, i.e. pre-defined programming building blocks with an efficient implementation for the given platform. Algorithmic skeletons were firstly introduced by Cole [5]. He proposed four basic skeletons including the *Fixed Degree Divide & Conquer*. The fixed degree requirement imposes that every divide phase produces a constant number of sub-problems. This practically holds in many real-life problems. Hovewer, to be more general and flexible, we removed this constraint in our parallel pattern.

Starting from Cole's proposal, various skeleton frameworks have introduced their own implementation of the divide-and-conquer skeleton. Examples can be found in the eSkel [6] and Skandium [18]. In [10] authors show in detail the implementation of a divide-and-conquer skeleton for the `MaLLBa` framework, while in [24] the detailed implementation on the *Muesli* library is presented. In both the cases the programmer specifies four different operators, which are essentially equivalent to the ones of our template.

Abstracting from their interface details, our proposed pattern template shares many common principles with these previous works. However, while these proposals consider a different skeleton for each particular framework, in this paper we propose a more *general* pattern template which can be easily implemented in different back-end frameworks by maintaining the same interface and, almost, the same level of performance. This makes it possible to achieve high code (re-)usability and portability.

6. Conclusions and Future Work

In this paper we proposed a parallel template for divide-and-conquer problems which represent a notable class of recurrent algorithms. The pattern aims at simplifying the parallel implementation on multi-core platforms providing a ready- and easy-to-use solution to the user that does not require any particular parallel expertise.

The parallel pattern has been implemented in various backend environments: in this way, by maintaining the same source code, the programmer can exploit the potential of different frameworks and target architectures. We proposed three different implementations for multi-core architectures based on OpenMP compiler annotations, Intel TBB and FastFlow parallel programming libraries. The experimental analysis, performed on a 24-cores Intel server, showed that the reduced effort in programming does not come at the expense of significant performance penalties. The experimental study has been done by comparing the pattern-based solution with hand-made parallelizations using the same backend runtime.

These results pave the way to further development of this work. First, the set of backend implementations can be further extended, including a MPI implementation for targeting distributed systems, and a CUDA/OpenCL-based implementation for GPUs. Second, we recognize that an important role in achieving good level of performance is played by the cutoff value, i.e. the point at which we stop the recursion and solve the problem sequentially to better exploit the cache hierarchy and/or limit the runtime support overhead. This value depends on the structure of the specific parallelized application and on the kind of platform used. As proposed in [11], using information from the application collected at runtime (without relying on any user hints), it is possible to automatically derive the cutoff technique that is best suited for the application. This technique can be performed automatically, by regulating the optimal cut-off value based on updated measurements while the code is running. Therefore, this approach can be studied integrately with autonomic supports like the ones previously studied in [20, 21].

Acknowledgments

This work has been partially supported by the EU H2020-ICT-2014-1 project RePhrase (No. 644235).

References

[1] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 662–673, Rhodes Island, Greece, Aug. 2012. Springer. doi: 10.1007/978-3-642-32820-6_65. URL `http://calvados.di.unipi.it/storage/paper_files/2012_spsc_europar.pdf`.

[2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. *A Proposal for Task Parallelism in OpenMP*, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-69303-1. doi: 10.1007/978-3-540-69303-1_1.

[3] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '08, pages 501–510, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[4] D. Buono, M. Danelutto, T. D. Matteis, G. Mencagli, and M. Torquati. A lightweight run-time support for fast dense linear algebra on multi-core. In *Proc. of the 12th International Conference on Parallel and Distributed Computing and Networks (PDCN 2014)*. IASTED, ACTA press, Feb. 2014.

[5] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-53086-4.

[6] M. Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, Mar. 2004. ISSN 0167-8191. doi: 10.1016/j.parco.2003.12.002. URL `http://dx.doi.org/10.1016/j.parco.2003.12.002`.

[7] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001. ISBN 0070131511.

[8] M. Danelutto and M. Torquati. Structured parallel programming with "core" fastflow. In V. Zsók, Z. Horváth, and L. Csató, editors, *Central European Functional Programming School*, volume 8606 of *LNCS*, pages 29–75. Springer, 2015. ISBN 978-3-319-15939-3. doi: 10.1007/978-3-319-15940-9_2. URL `http://dx.doi.org/10.1007/978-3-319-15940-9_2`.

[9] D. del Rio Astorga, M. F. Dolz, L. M. Sanchez, and J. D. García. Discovering Pipeline Parallel Patterns in Sequential Legacy C++ Codes. In *Procs of the 7th Int'l Workshop on Progr. Models and Applications for Multicores and Manycores*, PMAM'16, pages 11–19, NY, USA, 2016. ACM. ISBN 978-1-4503-4196-7. doi: 10.1145/2883404.2883411. URL `http://doi.acm.org/10.1145/2883404.2883411`.

[10] I. Dorta, C. Leon, C. Rodriguez, and A. Rojas. Parallel skeletons for divide-and-conquer and branch-and-bound techniques. In *Parallel, Distributed and Network-Based Processing, 2003. Proceedings. Eleventh Euromicro Conference on*, pages 292–298, Feb 2003. doi: 10.1109/EMPDP.2003.1183602.

[11] A. Duran, J. Corbalán, and E. Ayguadé. An adaptive cut-off for task parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 36:1–36:11, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.

[12] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of openmp task scheduling strategies. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 100–110, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-79560-X, 978-3-540-79560-5. URL `http://dl.acm.org/citation.cfm?id=1789826.1789838`.

[13] A. Fonseca and B. Cabral. Evaluation of runtime cut-off approaches for parallel programs. In *Proceedings of the 12th International Meeting on High Performance Computing for Computational Science (VECPAR 2016)*. Springer, 2016. to appear.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998. ISSN 0362-1340. doi: 10.1145/277652.277725. URL `http://doi.acm.org/10.1145/277652.277725`.

[15] C. H. Gonzalez and B. B. Fraguela. A generic algorithm template for divide-and-conquer in multicore systems. In *Proceedings of the 2010 IEEE 12th International Conference on High Performance Computing and Communica-*

tions, HPCC '10, pages 79–88, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4214-0. doi: 10.1109/HPCC.2010.24. URL `http://dx.doi.org/10.1109/HPCC.2010.24`.

[16] C. A. Herrmann and C. Lengauer. A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(02n03): 239–250, 2000.

[17] *Intel®Threading Building Blocks (Intel®TBB) Developer Guide*. Intel ®, 2016. Available at: `https://www.threadingbuildingblocks.org/docs/help/tbb_userguide/Design_Patterns/Divide_and_Conquer.html`.

[18] 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*, pages 289–296, Feb 2010. doi: 10.1109/PDP.2010.26.

[19] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004. ISBN 0321228111.

[20] G. Mencagli, M. Vanneschi, and E. Vespa. Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 11–18, July 2013. doi: 10.1109/HPCSim.2013.6641387.

[21] G. Mencagli, M. Vanneschi, and E. Vespa. A cooperative predictive control approach to improve the reconfiguration stability of adaptive distributed parallel applications. *ACM Trans. Auton. Adapt. Syst.*, 9(1):2:1–2:27, Mar. 2014. ISSN 1556-4665. doi: 10.1145/2567929. URL `http://doi.acm.org/10.1145/2567929`.

[22] K. Molitorisz, T. Müller, and W. F. Tichy. Patty: A pattern-based parallelization tool for the multicore age. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, pages 153–163, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3404-4. doi: 10.1145/2712386.2712392. URL `http://doi.acm.org/10.1145/2712386.2712392`.

[23] OpenMP Architecture Review Board. Openmp application program interface, 2011. URL `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`.

[24] M. Poldner and H. Kuchen. Task parallel skeletons for divide and conquer. In *Proceedings of the Workshop of the Working Group Programming Languages and Computing Concepts of the German Computer Science Association GI*, Bad Honnef, 2008. URL `http://danae.uni-muenster.de/lehre/kuchen/PUBLICATIONS/Honnef08.pdf`. Publication status: Published.

[25] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.