

# Skandium: Multi-core Programming with Algorithmic Skeletons

Mario Leyton

NIC Labs, Universidad de Chile,  
Miraflores 222, Piso 14, 832-0198, Santiago, Chile.  
mleyton@niclabs.cl

José M. Piquer

DCC, Universidad de Chile,  
Av. Blanco Encalada 2120, Santiago, Chile.  
jpiquer@dcc.uchile.cl

**Abstract**—This paper argues that algorithmic skeletons are a suitable programming model for multi-core architectures. The high-level abstractions offered by algorithmic skeletons provide a simple way for non-parallel programmers to address parallel programming.

Previous algorithmic skeleton frameworks and libraries have addressed distributed computing environments such as Clusters and Grids. This paper proposes a parallel skeleton library, Skandium; and concludes, after an experimental evaluation, that algorithmic skeletons are an effective methodology to program multi-core architectures.

**Keywords**-parallel programming; multi-core; patterns; algorithmic skeletons

## I. INTRODUCTION

The relevance of parallel and distributed computing is evident in a variety of scientific and engineering domains such as Astrophysics, Meteorology, Bioinformatics, Particle Physics, etc. All of them require increasing computation capacities in order to solve their problems. Parallel and distributed computing offer the possibility to address increasingly complex problems now, which would otherwise take decades or be unfeasible to solve. The applications are many and diverse such as sequence alignment (BLAST) or protein folding in Bioinformatics, astrophysics data analysis like SETI@home, particle physics analysis at CERN's Large Hadron Collider, Google's web crawling and page ranking, etc.

For the last few decades the computer industry has been a cycle where new faster hardware enables the development of more complex and resource demanding software, which in turn promotes the development of even faster hardware. To increase the speed of an application, the best strategy has been simply to buy newer hardware.

Nevertheless, this paradigm has been challenged as of 2006 when Intel, for the first time, introduced new processors with slower speeds than previous ones, but having multiple cores. The free ride is over: old software runs slower on new machines. The multi-core crisis states that an increase of performance, with newer hardware, will only happen if the application is a parallel one, which was never required before.

Furthermore, this trend will continue over the following decades as Intel, the leading CPU manufacturer, will continue to develop multi-core systems. We shall soon see processors featuring 80 cores and eventually possibly hundreds and even thousands of cores [1]. The danger is that parallel programming, which has up to now been

reserved for an elite of programmers, will be mandatory for all types of programmers world wide, along with its complexities and pitfalls.

Algorithmic skeletons (*skeletons* for short) are a high level programming model for parallel and distributed computing, introduced by Cole in [2]. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones. All the non-functional aspects regarding parallelization and distribution are implicitly defined by the composed parallel structure. Once a structure has been defined, programmers complete the program by providing the application's sequential blocks, called *muscle* functions. Muscle functions correspond to the sequential functions of the program. We classify muscle functions using the following categories: execution (denoted  $f_e$ ), evaluation of conditions ( $f_c$ ), split of data ( $f_s$ ), and merge of results ( $f_m$ ).

Traditionally, algorithmic skeletons have been used in distributed environments such as Clusters and Grids. We believe that algorithmic skeletons' high-level abstractions are also suitable for multi-core programming. In this paper we present an algorithmic skeleton library, Skandium [3], for parallel programming of multi-core architectures in Java [4]. Skandium is a complete reimplementaion of Calcium [5], [6], [7] (part of the ProActive Middleware [8]), but focused on parallel multi-core computing.

This paper is organized as follows. Section II discusses the related work, section III describes the programming model, section IV show how to implement QuickSort with the algorithmic skeleton programming, and section V provides experimental results on up to 8 cores. Finally, conclusions and future work are presented in section VI.

## II. RELATED WORK

TBB (Threading Building Blocks) is a C++ library for parallel programming developed by Intel to take advantage of multi-core architectures [9]. TBB offers parallel patterns such as: *for*, *reduce*, *scan*, *do*, *sort*, and *pipeline*; and concurrent data structures as: *hashmap*, *vector*, and *queue*. To our knowledge, TBB does not label itself as a skeleton framework, but we provide a description here because of the evident relationship with skeleton programming: parallelism is abstracted through patterns. Compared with skeleton frameworks, TBB provides lower level abstractions with more control on lower level parallelism aspects

such as granularity, the possibility to combine with other thread libraries, and direct access to the task scheduler.

Cilk++ is an extension of C++ with its own runtime and compiler [10]. Cilk++ extensions consist of three parallelism constructs: *cilk\_for* which allows parallel execution of iterations, *cilk\_spawn* which executes a function as a new thread, and *cilk\_sync* which blocks and waits for spawned threads to finish. The main drawback of Cilk++'s approach is that it requires a custom compiler which reduces compatibility with other libraries and frameworks. On the otherhand, the overhead introduced by Cilk++ parallelism is small when compared with pthread library primitives.

OpenMP is an API for multiprocessing in C, C++ and Fortran [11]. It consists of a set of compiler directives which annotate regular sequential code to exploit parallelism. At runtime, when a parallel section is encountered, a master thread spawns a specified number of slave threads to compute a task. Additionally a set of clauses can be used to specify the scope of data between threads: *shared*, *private*, etc; synchronization such as: *critical section*, *atomic*, *ordered*, *barrier*, *nowait*; among others. OpenMP applications use the regular compiler, but the annotation approach leads to an extensive tangling of functional and parallelism concerns in the code.

In Java, the Fork/Join framework [12] provides a recursive parallel programming model. Problems are solved by recursively splitting them, solving the sub-problems and then merging the results. There are many similarities between the Fork/Join approach and Skandium. For example, the recursive approach in Fork/Join resembles the well known divide and conquer skeleton pattern, where subtasks are generated and scheduled for execution. There are however some important differences. The first is that Skandium provides other patterns besides divide and conquer. Also that these patterns can be nested together to represent more complex problems. However the biggest difference is that in Skandium the program flow is not explicit, in the sense that it is not the programmers responsibility to explicitly call the *split*, *fork*, *join* and *compose* operations.

Since algorithmic skeletons were first introduced by Cole in [2], [13], their research has been mostly focused on Cluster and Grid computing. To name a few skeleton programming environments, eSkel [13], Lithium [14], [15], [16], [17], Muskel [18], Muesli [19], [20], [21], QUAFF [22], have targeted Cluster computing; while HOC [23] and Behavioral Skeletons [24] have focused on adapting skeletons for Grid computing.

Skandium is mainly inspired by Lithium [16], [17] and Muskel [25] frameworks, developed at University of Pisa. In all of them, skeletons are provided to the programmer through a Java API. Recent research for Muskel has focused on multi/many core clusters which exploits macro data flow. However, Skandium differs from Muskel by supposing a shared memory architecture, thus allowing stateful skeletons.

The SkeTo [26] project is a C++ library which achieves

parallelization using MPI. SkeTo is different to other skeleton libraries because instead of providing nestable parallelism patterns, SkeTo provides parallel skeletons for parallel data structures such as: lists [26], trees [27], [28], and matrices [29]. The data structures are typed using templates, and several parallel operations can be invoked on them. For example the list structure provides parallel operations such as: *map*, *reduce*, *scan*, *zip*, and *shift*. Recent research has also focused on multi-core clusters [30], using a two step scheduling

BlockLib [31] provides skeleton abstractions on C++ for the IBM Cell processor such as: *map*, *reduce*, and *map-with-overlap*. BlockLib is based on NestStep (a BSP platform) to interact with the Cell architecture. Hence BlockLib could be used with other multi-core architectures by using any other NestStep platform, but to our knowledge, no such attempt has been performed.

### III. A SHARED MEMORY ALGORITHMIC SKELETON MODEL

#### A. The Algorithmic Skeleton Language

In Skandium [3], skeletons are provided as a Java library. The library can nest task and data parallel skeletons in the following way:

$$\begin{aligned} \Delta ::= & \text{seq}(f_e) \mid \text{farm}(\Delta) \mid \text{pipe}(\Delta_1, \Delta_2) \mid \text{while}(f_c, \Delta) \mid \\ & \text{if}(f_c, \Delta_{\text{true}}, \Delta_{\text{false}}) \mid \text{for}(i, \Delta) \mid \text{map}(f_s, \Delta, f_c) \mid \\ & \text{fork}(f_s, \{\Delta_i\}, f_m) \mid \text{d\&c}(f_c, f_s, \Delta, f_m) \end{aligned}$$

Each skeleton represents a different pattern of parallel computation. All the communication details are implicit for each pattern, hidden away from the programmer.

#### B. Muscle (Sequential) Blocks

The nested skeleton pattern ( $\Delta$ ) relies on sequential blocks of the application. These blocks provide the business logic and transform a general skeleton pattern into a specific application. We denominate these blocks *muscles*, as they provide the *real* (non-parallel) functionality of the application.

In Skandium, muscles come in four flavors:

Execution	$f_e : P \rightarrow R$
Split	$f_s : P \rightarrow \{R\}$
Merge	$f_m : \{P\} \rightarrow R$
Condition	$f_c : P \rightarrow \text{boolean}$

Where  $P$  is the parameter type,  $R$  the result type, and  $\{X\}$  represents a list of elements of type  $X$ .

For the skeleton language, muscles are black boxes invoked during the computation of the skeleton program. Multiple muscles may be executed either sequentially or in parallel with respect to each other, in accordance with the defined  $\Delta$ . The result of a muscle is passed as a parameter to other muscle(s). When no further muscles need to be executed, the final result is delivered to the user.

### C. The Skeleton Patterns

Skandium skeletons have the following hypotheses to simplify the programming model and avoid programming errors.

- **Single input/output** *Skeletons can only receive/produce single inputs/outputs.* Therefore the proposed programming model does not provide skeleton such as *reduce* or *split*. Instead, these parallel behaviors are embedded directly into higher-level skeleton such as *map*, *d&c*, and *fork*. This hypothesis is relevant to simplify skeleton nesting and avoid programming errors. At the same time this hypothesis is not limiting, since inputs and outputs are objects which can encapsulate multiple data.
- **Passive Skeletons** *Each skeleton output is directly related to a previously received input.* Therefore *heartbeat* like skeletons, where a skeleton can produce outputs without receiving an input, are not allowed. This hypothesis simplifies termination detection of an application.

The currently supported skeleton patterns in Skandium are described as follows.

- **SEQ** terminates a recursive nesting of skeletons, and therefore does not really exploit parallelism. The *seq* skeleton wraps execution muscles which are then nested into the skeleton program as terminal leaves of the skeleton nesting tree.
- **FARM** represents task replication and is also known as master-slave scheme. Skeletons nested inside a *farm* are meant to be replicated for task parallelism.
- **PIPE** is one of the most classical skeletons in the literature. The *pipe* skeleton represents staged computations, where parallelism can be achieved by computing different stages simultaneously on different inputs. The number of stages provided by *pipe* can be variable or fixed, but it is worth noting that fixed staged *pipes* can be nested inside other fixed staged *pipes* to create a *pipe* with any number of stages.
- **IF** provides conditional branching. Two sub-skeletons are provided as parameters, along with a condition muscle. When an input arrives at the *if* skeleton, either one or the other sub-skeleton is executed, depending on the result of the condition muscle.
- **FOR** receives a sub-skeleton and an integer as parameters. The sub-skeleton will be executed the number of times specified by the integer parameter. The result of one invocation of a sub-skeleton is passed as parameter to the following invocation of the sub-skeleton. Eventually, the result of the last sub-skeleton is provided as the *for* skeleton's result.
- **WHILE** skeleton is analogous to the *for* skeleton, but instead of iterating a fixed number of times, a condition muscle decides whether the iteration must continue or stop. On each iteration, the result of the previous execution of the sub-skeleton is passed as input to the same sub-skeleton.
- **MAP** is the most classical skeleton representing data

parallelism. Its origin is closely related with functional languages. The semantics behind *map* specify that a function (or sub-skeleton) can be applied simultaneously to all the elements of a list to achieve parallelism. The data parallelism occurs because a single data element can be split into multiple data, then the sub-skeleton is executed on each data element, and finally the results are united again into a single result.

- **FORK** behaves like *map*. The difference is that instead of applying the same function (sub-skeleton) to all elements of a list, a different function (sub-skeleton) is applied to each list element. This solves the problem of having to apply the same function to all data elements in the *map* skeleton. With *fork* a different function can be applied to each data element.
- **D&C** is another classical skeleton in the literature. To put it simply, *d&c* is a generalization of the *map* skeleton, where *maps* are recursively applied while a condition is met. The semantics of *d&c* are as follows. When an input arrives, a condition muscle is invoked on the input. Depending on the result two things can happen. Either the parameter is passed on to the sub-skeleton, or the input is split with the split muscle into a list of data. Then, for each list element the same process is applied recursively. When no further recursions are performed, the results obtained at each level are merged using a merge muscle. Eventually, the merged results yields one result which corresponds to the final result of the *d&c* skeleton. Thus, in our case, the depth of the recursion and width of the division are not fixed, but will depend on the muscles and data provided by the user.

### D. Moving Back From Distributed Programming

From our experience on the design and implementation of Calcium and now Skandium, it is easier to design and implement a skeleton library on a shared memory environments than distributed ones. Many concerns can be discarded such as: deployment, heterogeneity, and fault tolerance. Unfortunately, there is a loss of control, for example in memory locality and task scheduling. Specially when designing a Java library.

For us, the interesting aspects of moving back to a shared memory model are not the differences in the design and implementation of the library. But, the restrictions which can be relaxed from the algorithmic skeleton programming model. The most relevant being *stateless muscles*, a fundamental restriction on Calcium to achieve good performance on distributed memory environments. However, on a shared memory environment it is natural to allow for both stateless and stateful muscles. This eases the programming of algorithmic skeletons applications, but raises the issue critical sections synchronization.

### E. Synchronization of Critical Sections

Two threads which simultaneously access shared data may create an inconsistent data state, this is known as a data race. Synchronization tools such as monitors, locks, semaphores, and others can be used to prevent concurrent access to data within a critical section.

As a library, Skandium does not impose synchronization primitives but allows programmers to use their preferred synchronization tools to define critical sections. The simplest tool in Java is the *synchronized* idiom which does not allow two synchronized method/block invocations to interleave on the same object.

Skeleton patterns will try to execute muscle codes concurrently whenever possible. So programmers must implement muscles with the inherent supposition that they will be executed concurrently. The best mechanism is to program, as much as possible, stateless muscles which cannot have critical sections.

However, when a muscle requires state and has a critical section, the simplest and worst approach is to add the *synchronized* keyword to the muscle method:

```
BadExecute implements Execute<P,R>{
    int counter = 0;
    public synchronized R execute(P p){
        counter++;
        //... Intensive computation here
    }
}
```

Unfortunately, this approach serializes all parallel invocations to a muscle and can render ineffective all parallelism for some skeletons. For example, consider  $farm(f_e)$ , which replicates the execution of  $f_e$ . If  $f_e$  is synchronized, then all replication is effectively serialized and the parallelism is destroyed. The problem is further aggravated when the same object is used to implement several muscle functions, all sharing the same object monitor.

A better approach is to reduce the span of critical sections, and allow parallel execution of computation intensive blocks:

```
BetterExecute implements Execute<P,R>{
    int counter = 0;
    public R execute(P p){
        synchronized(this){
            counter++;
        }
        //... Intensive computation here
    }
}
```

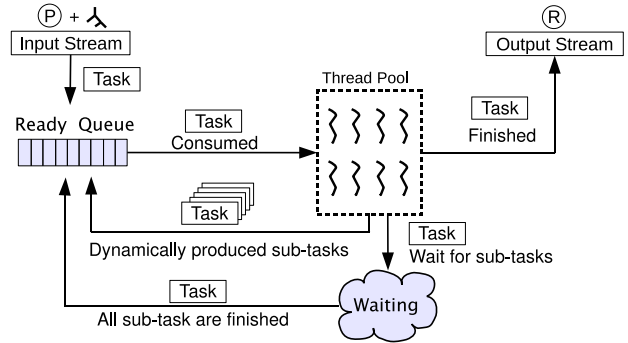


Figure 1. Thread Pool Execution

```
}
}
```

*Pitfall: Wait/Notify:* A fixed amount of threads are used to execute muscle codes. Therefore, it is not advised to use synchronizations tools which suspend the execution of the current thread, such as *wait*, from inside muscles.

```
Dangerous implements Execute<P,R>{
    public R execute(P p){
        synchronized(...){
            while(condition)
                wait(); //Dangerous idea
        }
        //...
    }
}
```

The current thread, belonging to the thread pool, is suspended and will not be used to execute other muscles, which would eventually have called *notify*. Thus, the execution of the application is likely to completely halt, as waiting threads may never be woken up.

### F. Execution Environment

From a general perspective, parallelism or distribution of an application in Skandium is a producer/consumer problem, where the shared buffer is a task queue and the produced/consumed data are tasks, as shown in Figure 1.

A ready queue stores ready tasks. Root tasks are entered into the ready queue by users, who provide the initial parameter and the skeleton program. The skeleton program undergoes a transformation process into an internal stack of instructions which harness the parallelism behavior of each skeleton, as detailed in [32].

Interpreters threads, all on the same Java Virtual Machine, consume tasks from the ready queue and compute tasks according to their skeleton instruction stack. When the interpreters cannot compute a task any further, the task is either in finished or waiting state. If the task is in the finished state its result is delivered to the user. If the task is in the waiting state, then the task has generated new sub-tasks which are inserted into the ready queue. Sub-tasks represent data parallelism for skeletons such as *map*,

*fork*, *d&c*. A sub-task may in turn produce new sub-tasks. A task will exit the waiting state and be reinserted into the ready queue when all of its sub-tasks are finished.

The execution of a skeleton program is thus reduced to a producer/consumer problem.

#### IV. EXAMPLE: QUICKSORT

##### A. Skandium Input/Output

The code in Listing 1 shows how simple it is to interact with the Skandium API to input data and retrieve the result.

The first thing to do is define the skeleton program (1). For QuickSort the best skeleton corresponds to a Divide and Conquer (DaC), which requires four muscles  $f_c$ : ShouldSplit,  $f_s$ : SplitList,  $f_e$ : Sort,  $f_m$ : MergeList. The code of each muscle is shown later in this section. Note that  $f_e$  could be replaced by some other skeleton, hence allowing nested compositions.

Once the skeleton program is defined, we can input parameters for computation (2). In this case Range is a simple class which represents an array, a start and a finish index. The input is performed via the input method which returns a Future, used to cancel the computation and await the results.

Futures also allow for asynchronous computation. Hence in (3) other computation tasks can be performed by the main thread before actually requiring the results.

Finally, in (4) we retrieve the results, block until they are available, or an Exception is raised.

##### B. Muscles

The code in Listing 2 shows the muscle’s implementation for QuickSort. Some methods have been omitted for the sake of clarity.

**Condition** The ShouldSplit condition returns true if the array’s range should be subdivided. This happens when the range’s size is bigger than a given threshold, and the number of maximum subdivisions has not been reached. Note that the condition method has been synchronized to serialize access to the times variable. Since the ShouldSplit condition is composed into a DaC skeleton, it will be used to determine if further splits are required.

**Split** In this example, SplitList takes an initial range and partitions it into two sub ranges using a partition method. The partition method follows the quicksort algorithm: all elements on the left of *i* are smaller than all elements on the right of *i*. Afterwards two new Range objects are created to represent each partition and returned.

**Execute** Once the DaC splitting is done, a sort method is applied to each Range of the array. The implementation is simple, we call Java’s Arrays.sort method to on the specific block. Once this method returns the array represented by the Range will be sorted.

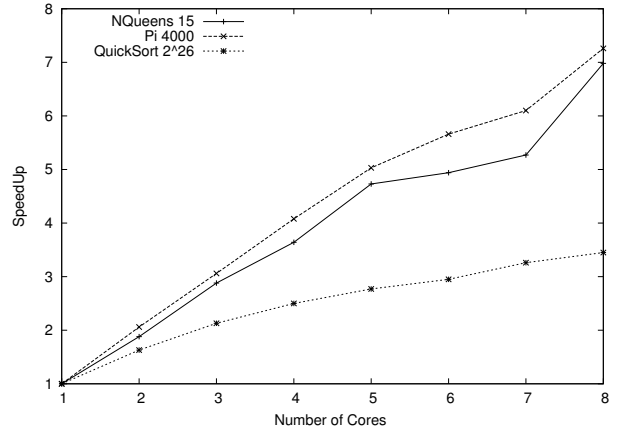


Figure 2. SpeedUp

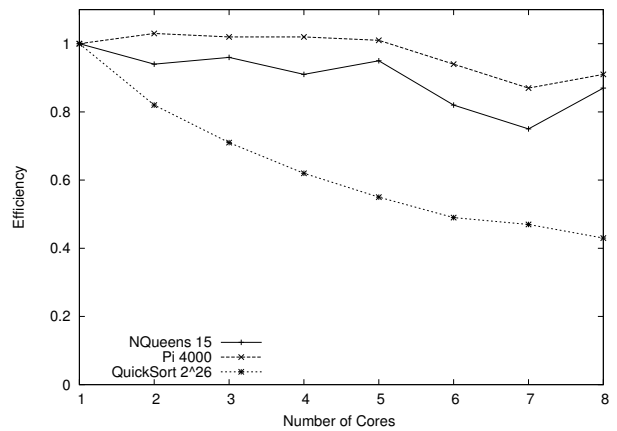


Figure 3. Efficiency

**Merge** The final step is to merge back all the Range objects into a single one. No actual sorting is required here, as all sub Ranges are already sorted, so we simply reconstruct a Range representing the union of both sub-Ranges.

#### V. EXPERIMENTATION

We tested our Skandium implementation on a 2 x Intel Xeon Quad E5420 2.5Ghz, for a total of 8 cores, with 4x4GB RAM at 667Mhz. The 64 bit Java Runtime Environment version 1.6.0\_14 was used, and configured with a maximum heap size of 512MB. Three applications were tested: NQueens, Pi, and QuickSort. The NQueens problem counted the number of ways  $n$  queens can be placed on a chessboard of size  $n \times n$ , and was tested for  $n = 15$ . The Pi problem computed the first 4000 decimals of  $\pi$ , and the QuickSort problem sorted an array of  $2^{26}$  integer elements.

The results are shown in Figures 2 and 3 for [1, 8] cores. The figures show that both NQueens and Pi have good efficiencies. Pi even experience super linear speedups for [2, 5] processors, because of Java’s BigDecimal class implementation, which performs faster when fewer decimals are used. On the other hand, the figures show that QuickSort’s efficiency dramatically decreases as the number of

```

// 1. Define the skeleton program
Skeleton<Range, Range> sort = new DaC<Range, Range>(
    new ShouldSplit(threshold, maxTimes),
    new SplitList(),
    new Sort(),
    new MergeList());

// 2. Input parameters
Future<Range> future = sort.input(new Range(generate(...)));

// 3. Do something else here.
// ...

// 4. Block for the results
Range result = future.get();

```

Listing 1. Skandium: Data Input & Result Output

cores increases. The difference between QuickSort and the other problems is that NQueens and Pi work on small and localized memory, while QuickSort requires more memory bandwidth to read array elements. This evidences one of the main problems with multi-core programming: the *memory access bottleneck*. Which, for sorting, can be improved with algorithms such as mergesort, that reduce the bandwidth transfer [33].

## VI. CONCLUSIONS AND FUTURE WORK

This paper has introduced Skandium [3], an algorithmic skeleton programming Java library for multi-core architectures. Skandium takes the high-level abstractions of algorithmic skeleton programming to the multi-core context where parallelism is the only way to provide speedups in new hardware.

We have described the programming model; provided an example; and benchmarked the NQueens counting problem, Pi decimals computation, and QuickSort on an 8 core machine.

In the future we would like to improve the programming model, for example the exception and event management. Also, on the computation side we want to explore the possibility of using algorithmic skeletons to program GPGPU processors.

## REFERENCES

- [1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from Berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006.
- [2] M. Cole, *Algorithmic skeletons: structured management of parallel computation*. Cambridge, MA, USA: MIT Press, 1991.
- [3] Skandium, <http://skandium.niclabs.cl/>.
- [4] S. Microsystems, "Java," <http://java.sun.com>.
- [5] D. Caromel and M. Leyton, "Fine tuning algorithmic skeletons," in *13th International Euro-Par Conference: Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 4641. Springer-Verlag, 2007, pp. 72–81.
- [6] —, "A transparent non-invasive file data model for algorithmic skeletons," in *22nd International Parallel and Distributed Processing Symposium (IPDPS)*. Miami, USA: IEEE Computer Society, March 2008, pp. 1–8.
- [7] D. Caromel, L. Henrio, and M. Leyton, "Type safe algorithmic skeletons," in *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-based Processing*. Toulouse, France: IEEE CS Press, Feb. 2008, pp. 45–53.
- [8] D. Caromel, C. Delbé, A. di Costanzo, and M. Leyton, "Proactive: an integrated platform for programming and running applications on grids and p2p systems," *Computational Methods in Science and Technology*, vol. 12, 2006.
- [9] Intel, "Threading building blocks," <http://www.threadingbuildingblocks.org>.
- [10] Cilk++, <http://www.cilk.com/>.
- [11] OpenMP, <http://openmp.org/>.
- [12] D. Lea, "A java fork/join framework," in *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*. New York, NY, USA: ACM, 2000, pp. 36–43.
- [13] 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.
- [14] M. Aldinucci and M. Danelutto, "Stream parallel skeleton optimization," in *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, IASTED. Cambridge, Massachusetts, USA: ACTA press, Nov. 1999, pp. 955–962.
- [15] M. Aldinucci, M. Danelutto, and J. Dünneweber, "Optimization techniques for implementing parallel skeletons in grid environments," in *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, S. Gorlatch, Ed. Stirling, Scotland, UK: Universität Münster, Germany, Jul. 2004, pp. 35–47.

```

public class ShouldSplit implements Condition<Range>{

    int threshold, maxTimes, times;

    public ShouldSplit(int threshold, int maxTimes){
        this.threshold = threshold;
        this.maxTimes = maxTimes;
        this.times = 0;
    }

    @Override
    public synchronized boolean condition(Range r){
        return r.right - r.left > threshold &&
            times++ < this.maxTimes;
    }
}

public class SplitList implements Split<Range, Range>{

    @Override
    public Range[] split(Range r){

        int i = partition(r.array, r.left, r.right);

        Range[] intervals = {new Range(r.array, r.left, i-1),
            new Range(r.array, i+1, r.right)};

        return intervals;
    }
}

public class Sort implements Execute<Range, Range> {

    @Override
    public Range execute(Range r){

        if (r.right <= r.left) return r;

        Arrays.sort(r.array, r.left, r.right+1);

        return r;
    }
}

public class MergeList implements Merge<Range, Range>{

    @Override
    public Range merge(Range[] r){

        Range result = new Range( r[0].array, r[0].left, r[1].right);

        return result;
    }
}

```

Listing 2. QuickSort's DaC Skeleton Muscles

- [16] M. Aldinucci, M. Danelutto, and P. Teti, "An advanced environment supporting structured parallel programming in Java," *Future Generation Computer Systems*, vol. 19, no. 5, pp. 611–626, Jul. 2003.
- [17] M. Danelutto and P. Teti, "Lithium: A structured parallel programming environment in Java," in *Proc. of ICCS: International Conference on Computational Science*, ser. LNCS, vol. 2330. Springer Verlag, Apr. 2002, pp. 844–853.
- [18] M. Danelutto and P. Dazzi, "Joint structured/unstructured parallelism exploitation in Muskel," in *Proc. of ICCS 2006 / PAPP 2006*, ser. LNCS. Springer Verlag, May 2006, to appear.
- [19] H. Kuchen and J. Striegnitz, "Higher-order functions and partial applications for a c++ skeleton library," in *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*. New York, NY, USA: ACM Press, 2002, pp. 122–130.
- [20] H. Kuchen, "A skeleton library," in *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*. London, UK: Springer-Verlag, 2002, pp. 620–629.
- [21] H. Kuchen and J. Striegnitz, "Features from functional programming for a c++ skeleton library," *Concurrency - Practice and Experience*, vol. 17, no. 7-8, pp. 739–756, 2005.
- [22] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté, "Quaff: efficient c++ design for parallel skeletons," *Parallel Computing*, vol. 32, no. 7, pp. 604–615, 2006.
- [23] J. Dünneweber, A. Benoit, M. Cole, and S. Gorlatch, "Integrating mpi-skeletons with web services," vol. 33, pp. 787–794, 2005.
- [24] M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Kilpatrick, P. Dazzi, D. Laforenza, and N. Tonello, "Behavioural skeletons in gcm: Autonomic management of grid components," in *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 54–63.
- [25] M. Danelutto, "Qos in parallel programming through application managers," in *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 282–289.
- [26] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, "A library of constructive skeletons for sequential style of parallel programming," in *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*. New York, NY, USA: ACM, 2006, p. 13.
- [27] K. Matsuzaki, Z. Hu, and M. Takeichi, "Parallelization with tree skeletons," in *Euro-Par*, ser. Lecture Notes in Computer Science, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds., vol. 2790. Springer, 2003, pp. 789–798.
- [28] —, "Parallel skeletons for manipulating general trees," *Parallel Comput.*, vol. 32, no. 7, pp. 590–603, 2006.
- [29] K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi, "A compositional framework for developing parallel programs on two dimensional arrays," Department of Mathematical Informatics, University of Tokyo, Tech. Rep., 2005.
- [30] Y. Karasawa and H. Iwasaki, "A parallel skeleton library for multi-core clusters," in *Proceedings of 38th International Conference on Parallel Processing (ICPP 2009)*, 2009.
- [31] M. Alind, M. V. Eriksson, and C. W. Kessler, "Blocklib: a skeleton library for cell broadband engine," in *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*. New York, NY, USA: ACM, 2008, pp. 7–14.
- [32] M. Leyton, "Advanced features for algorithmic skeleton programming," Ph.D. dissertation, Université de Nice-Sophia Antipolis, October 2008.
- [33] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core simd cpu architecture," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1313–1324, 2008.