

# A Skandium based Parallelization of DNSSEC

Mario Leyton<sup>†</sup> and José M. Piquer<sup>‡</sup>

<sup>†</sup>NIC Labs, Universidad de Chile

Miraflores 222, Piso 14, 832-0198, Santiago, Chile.

mleyton@niclabs.cl

<sup>‡</sup>Departamento de Ciencias de la Computación, Universidad de Chile,

Av. Blanco Encalada 2120, Santiago, Chile.

jpiquer@dcc.uchile.cl

## Abstract

*This paper shows an effective methodology to parallelize DNSSEC using an algorithmic skeleton library: Skandium.*

*DNSSEC is the Domain Name System (DNS) Security Extensions. DNSSEC provides a set of backward compatible extensions to the DNS that secure the domain name resolution process. The application of these extensions to a domain zone introduces new CPU intensive tasks, where parallelism can be applied to reduce the digital signing time.*

*The high-level abstractions offered by algorithmic skeletons provide a simple way for non-parallel programmers to address parallel programming. The objective of this paper is to show how algorithmic skeletons can be used to successfully parallelize a real world problem, DNSSEC, on a multi-core hardware.*

*Keywords:* dns, dnssec, parallel programming, multi-core, patterns, algorithmic skeletons

## 1 Introduction

The Domain Name System (DNS) is a hierarchical, world wide distributed database which provides a lookup service for resources on the Internet and local private networks. A variety of information can be associated with a unique lookup key called a *domain name*, the most important being the resolution of a human readable domain names into a network address identifiers (IPv4 or IPv6).

The responsibility of answering for a particular domain name is assigned to *authoritative servers*, starting with the root servers in charge of the “.” domain. An authoritative server can answer domain and sub-domain lookups, or delegate authority of *sub-domains* to other authoritative name servers. For a given domain, a *domain zone* corresponds to the group of domain records, undelegated sub-domains, and

the sub-domain authority delegation records.

The Domain Name Security Extensions (DNSSEC) is a set of backward compatible extensions to the DNS that secure the domain name resolution process [6]. The security provided by DNSSEC features: origin authentication of DNS data, data integrity, and authenticated denial of existence. The objective of DNSSEC is to protect resolvers (clients) from forged DNS data. Thanks to digitally signed DNS answers, a resolver is capable of checking if received data is complete and correct with respect to the authoritative name server.

The top level domains (TLD) are classified into generic (gTLD) such as .com, .net, .org, and country code TLD (ccTLD) such as: .cl, .fr, .de. The process of digitally signing each answer is of course CPU intensive, and imposes a new burden on authoritative servers, and in particular to TLDs. The time required to digitally sign domain zones is proportional to the number of records in the domain zone and the size of the signing key.

For the last decades, the standard strategy to increase the speed of an application has been to buy new hardware. The computer industry has been in 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.

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 with newer hardware, an increase of performance will only happen if the application is parallel.

Furthermore, this trend will continue over the following decades as CPU manufacturers develop multi-core systems with an increasing number of cores. For example, we shall soon see processors featuring 80 cores, such as Intel’s

Polaris, and eventually possibly hundreds and even thousands of cores [7]. 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.

Therefore, to improve the performance of DNSSEC's zone signing, parallelization must be applied.

Algorithmic skeletons (*skeletons* for short) are a high level programming model for parallel and distributed computing, introduced by Cole in [13]. Skeletons take advantage of recurrent 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 show how a Java [27] based algorithmic skeleton library, Skandium [29], can be used to parallelize DNSSEC's digital signing of domain zones. Skandium is a complete reimplement of Calcium [10, 11, 9] (part of the ProActive Middleware [8]), but focused on parallel multi-core computing.

This paper is organized as follows. Section 2 discusses the related work. Section 3 provides an overview of the DNS and in particular DNSSEC. Section 4 briefly describes Skandium's programming model. Section 5 shows how to parallelize DNSSEC zone signing with Skandium. Section 6 provides experimental results on up to 4 cores. Finally, conclusions and future work are presented in section 7.

## 2 Related Work

TBB (Threading Building Blocks) is a C++ library for parallel programming developed by Intel to take advantage of multi-core architectures [22]. 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: paral-

lelism 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.

Cylk++ is an extension of C++ with its own runtime and compiler [12]. Cylk++ 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 the Cylk++ approach is that it requires a custom compiler which reduces compatibility with other libraries and frameworks. On the other hand, the overhead introduced by Cylk++ parallelism is small when compared with pthread library primitives.

OpenMP is an API for multiprocessing in C, C++ and Fortran [28]. 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 [26] 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. It is not the programmer's responsibility to explicitly call the split, fork, join and compose operations.

Since algorithmic skeletons were first introduced by Cole in [13, 14], their research has been mostly focused on Cluster and Grid computing. To name a few skeleton programming environments, eSkel [14], Lithium [2, 3, 4, 18], Muskel [17], Muesli [24, 23, 25], QUAFF [20], have targeted Cluster computing; while HOC [19] and Behavioral Skeletons [1] have focused on adapting skeletons for Grid computing.

The only other algorithmic skeleton library we are aware of for multi-core infrastructures corresponds to BlockLib [5]. BlockLib 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.

Skandium is mainly inspired by Lithium [4, 18] and Muskel [15] frameworks, developed at University of Pisa. In all of them, skeletons are provided to the programmer through a Java API. Research related to Lithium and Muskel has focused mainly on providing optimizations based on: skeleton rewriting techniques [4, 2], task lookahead, and server-to-server lazy binding [3]. Recently, additional research has been performed concerning the combination of unstructured and structured programming [16].

QUAFF [20] is a recent skeleton library written in C++ and MPI. QUAFF relies on template-based meta-programming techniques to reduce runtime overheads and perform skeleton expansions and optimizations at compilation time. Skeletons can be nested and sequential functions are stateful. Besides type checking, QUAFF takes advantage of C++ templates to generate, at compilation time, new C/MPI code. QUAFF is based on the CSP-model, where the skeleton program is described as a *process network* and production rules (single, serial, par, join) [21].

## 3 DNS & DNSSEC Background

### 3.1 The Domain Name System (DNS)

The Domain Name System is a hierarchical, distributed lookup service, where a domain name is used as a key to obtain information for a given record type. The most used lookup service corresponds to the translation of domain names into IP addresses, but is not limited to this as other information may be served by the DNS.

The following DNS concepts are relevant in the context of this paper.

**Domain Names** A domain name consists of a set of labels separated by dots, such as `niclabs.cl`. The first label (right to left) is the top level domain (TLD). Each label represents a subdivision or sub-domain of the parent label.

**Domain Zone** The DNS is a hierarchical system which can be represented as a tree. A domain zone corresponds to a branch of the tree for which a server is authoritative.

**Authoritative Servers** Are responsible for answering queries for a given zone. The authoritative servers for the top most domain “.” are called root name servers, which are responsible for delegating the authority of TLDs.

**Resolvers** A resolver is a client of the DNS. Its role is to query authoritative servers, and possibly follow the delegation chain to fully resolve a domain name query.

### 3.2 DNSSEC Extensions

The Domain Name System Security Extensions (DNSSEC) is a specification which extends the traditional DNS to improve its security. DNSSEC improves three main features:

**Origin authentication** Resolvers can verify the data’s source, to check if it was generated by an authoritative server.

**Data Integrity** Resolvers can verify the data’s integrity, to check that it has not been corrupted or forged.

**Authenticated denial of existence** Responses stating that a domain name does not exist can be verified by resolvers.

Currently, the process of signing a domain zone with DNSSEC is the following:

1. **Clean & Sort** If present, remove existent DNSSEC records and sort.
2. **Remove duplicates** Remove duplicate records in the zone.
3. **Generate DS** Generate the DS records for the public key signing key (KSK) and public zone signing key (ZSK).
4. **Generate NSEC3 & Sort** Generate the NSEC3 records, responsible for providing authenticated denial of existence, and sorting the zone.
5. **Sign RRSets** Group the records into RRSets and sign them.

Table 1 shows the time distribution and number of records after each stage for a recent .CL domain zone provided by NIC Chile. The zone consists of approximately 250.000 sub-domain delegations. The most time consuming stage corresponds to the signing of the RRSets with 99.47% of the computation time, and should concentrate the parallelization efforts. Additionally, it is worth noticing that the final size of the zone, in number of records, is roughly 2x the initial size; and represents approximately 4x the number of delegated sub-domains.

Action	Time %	Num. Records
1. Clean & Sort	0.06	563 909
2. Remove duplicates	0.01	563 905
3. Generate DS	0.01	563 907
4. Generate NSEC3 & Sort	0.45	800 890
5. Sign RRsets	99.47	1 037 880

**Table 1. .CL DNSSEC Zone Signing Process**

## 4 Skandium’s Algorithmic Skeleton Programming Model

In Skadium [29], skeletons are provided as a Java library. The library can nest task and data parallel skeletons in the following way:

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

Each skeleton represents a different pattern of parallel computation. All the communication details are implicit for each pattern, hidden away from the programmer. The task-parallel skeletons are: *seq* for wrapping execution functions; *farm* for task replication; *pipe* for staged computation; *while/for* for iteration; and *if* for conditional branching. The data-parallel skeletons are: *map* for single instruction multiple data; *fork* which is like *map* but applies multiple instructions to multiple data; and *d&c* for divide and conquer.

### 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.

## 5 DNSSEC Parallelization

Following the suggestion of Table 1, we concentrated the parallelization efforts in the signing of resource records sets (RRsets). We replaced the call to a sequential signing method with the Skandium parallel implementation shown on Listing 1.

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 to define the skeleton program (1). For DNSSEC’s signing stage, the best skeleton corresponds to a *map* which divides the list of records into sublists and signs each sublist in parallel. A *map* requires three muscles  $f_s$ : `SplitRecords`,  $f_e$ : `SignRecords`,  $f_m$ : `MergeRecords`. The code of each muscle is described 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 parallel computation (2). In this case `SignData` is a simple class which contains relevant signing information such as keys and the array of records to sign. The input is performed via the input method and a `Future` is returned. The `Future` can be used to cancel the computation and await the results, blocking if necessary.

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

Finally, in (4) we return the results, block until they are available, or raise an `Exception`.

### 5.1 Muscles

The code in Listing 2 shows the muscle’s implementation for DNSSEC’s signing stage.

**Split** This interface is implemented by the `SplitRecords` class, which takes an initial `SignData` and partitions the list of records into sublists. Finally, a `SignData` array of length `parts` is returned.

**Execute** This interface is implemented by the `SignRecords` class. This method performs the actual signing of records, by invoking the `signRecordsSequentialMethod` which receives the signing parameters and a sublist of records. This method returns a list of records with their signatures.

### Listing 1. Skandium: Data Input & Result Output

```
public List signRecordsInParallel(SignData data) throws Exception{

    // 1. Define the skeleton program
    Map<SignData,List<Record>> skeleton = new Map<SignData,List<Record>>(
        new SplitRecords(),
        new SignRecords(),
        new MergeRecords());

    // 2. Input parameters
    Future<List<Record>> future = skeleton.input(data);

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

    // 4. Block for the results and return
    return future.get();
}
```

**Merge** The final step is to merge back all the records into a single list `MergeRecords`. All sublists containing the records and their signatures are merged into a single list.

## 6 Experimentation

We tested the Skandium DNSSEC parallelization on an Intel Xeon Quad E5420 2.5Ghz, for a total of 4 cores, with 1x4GB 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.

We used a recent .CL domain zone of approximately 250.000 sub-domain delegations. The signing keys were RSASHA1, with a Key Signing Key (KSK) of 1024[bits] and a Zone Signing Key (ZSK) of 512[bits].

The library that was parallelized corresponds to `jdnssec-tools-0.9.0`, which is built on top of Verisign's `dnsjava-2.0.6` library.

The results for [1, 4] cores are shown in Figures 1 and 2. The Figures show a linear speedup. This implies that DNSSEC zone signing is highly parallel, and therefore its performance should continue to increase as the number of cores per CPU increases. The linear speedup may be explained because of DNSSEC's highly localized memory access. In other words, there is no memory access bottleneck, and a very good usage of the cores' caches.

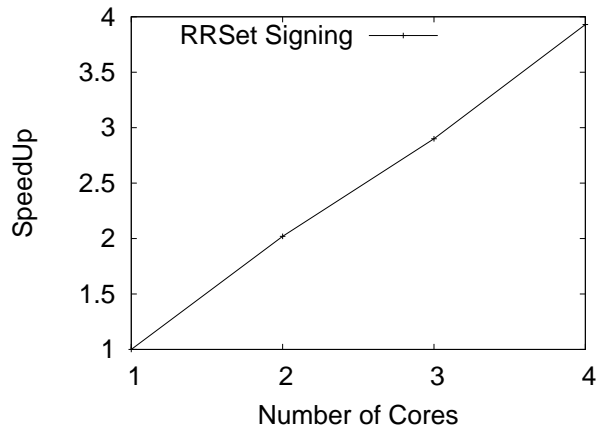


Figure 1. SpeedUp

## Listing 2. DNSSEC's Map Muscles: Split, Execute , Merge

```
public class SplitRecords implements Split<SignData, SignData>{

    @Override
    public SignData[] split(SignData sd) throws Exception {

        SignData[] sds = new SignData[parts];

        int size = sd.records.size()/parts;

        for(int i=0;i<sd.parts;i++){
            int first = i*size
            int last = i != sd.parts-1? (i+1)*size: sd.records.size()

            List<Record> sublist = sd.records.subList(first,last);
            sds[i]= new SignData(sd, sublist);
        }

        return sds;
    }
}

public class SignRecords implements Execute<SignData, List<Record>>{

    @Override
    public List<Record> execute(SignData sd) throws Exception {

        return JCEDnsSecSigner.signRecordsSequential(
            sd.zonename,
            sd.records,
            sd.kskpairs,
            sd.zskpairs,
            sd.start,
            sd.expire,
            sd.fullySignKeyset );
    }
}

public class MergeRecords implements Merge<List<Record>, List<Record>>{

    @Override
    public List<Record> merge(List<Record>[] lists) throws Exception {
        ArrayList<Record> merged = new ArrayList<Record>();

        for(List<Record> l:lists){
            merged.addAll(l);
        }
        return merged;
    }
}
```

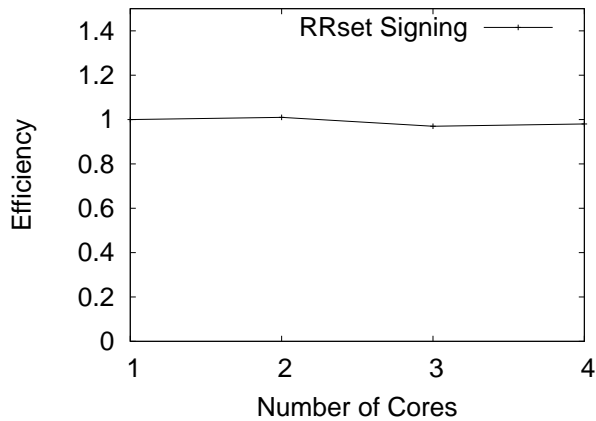


Figure 2. Efficiency

## 7 Conclusions and Future Work

This paper has shown how DNSSEC’s zone signing can be parallelized with algorithmic skeleton technology, in particular with the Skandium [29] Java library for multi-core architectures. Skandium provides a high-level abstractions based on algorithmic skeleton programming, but applied to multi-core, where parallelism must be used to provide speedups in new hardware.

DNSSEC’s zone signing is highly parallelizable activity, and does not suffer from a memory access bottleneck. Therefore, as the number of cores per CPU continues to increase, we should expect a near linear speedup of DNSSEC’s record signing.

In the future we would like to continue improving the algorithmic skeleton programming model and the Skandium library. On the DNSSEC side we plan to develop parallel incremental signing of records which would not require to sign the complete zone but only the differences from previous versions.

## References

- [1] Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Peter Kilpatrick, Patrizio Dazzi, Domenico Laforenza, and Nicola 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)*, pages 54–63, Washington, DC, USA, 2008. IEEE Computer Society.
- [2] Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED, ACTA press.
- [3] Marco Aldinucci, Marco Danelutto, and Jan Dünneweber. Optimization techniques for implementing parallel skeletons in grid environments. In S. Gorlatch, editor, *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 35–47, Stirling, Scotland, UK, July 2004. Universität Münster, Germany.
- [4] Marco Aldinucci, Marco Danelutto, and Paolo Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.
- [5] Markus Alind, Mattias V. Eriksson, and Christoph W. Kessler. Blocklib: a skeleton library for cell broadband engine. In *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*, pages 7–14, New York, NY, USA, 2008. ACM.
- [6] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. RFC 4033: DNS Security Introduction and Requirements. Technical Report RFC 4033, Network Working Group, 2005.
- [7] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.
- [8] Denis Caromel, Christian Delbé, Alexandre di Costanzo, and Mario Leyton. Proactive: an integrated platform for programming and running applications on grids and p2p systems. *Computational Methods in Science and Technology*, 12, 2006.
- [9] Denis Caromel, Ludovic Henrio, and Mario Leyton. Type safe algorithmic skeletons. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 45–53, Toulouse, France, February 2008. IEEE CS Press.
- [10] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *13th International Euro-Par Conference: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 2007.
- [11] Denis Caromel and Mario Leyton. A transparent non-invasive file data model for algorithmic skeletons. In

22nd International Parallel and Distributed Processing Symposium (IPDPS), pages 1–8, Miami, USA, March 2008. IEEE Computer Society.

- [12] Cilk++. <http://www.cilk.com/>.
- [13] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.
- [14] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.
- [15] Marco 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)*, pages 282–289, Washington, DC, USA, 2005. IEEE Computer Society.
- [16] Marco Danelutto and Patrizio Dazzi. Joint structured/non structured parallelism exploitation through data flow. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *Proc. of ICCS: International Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming*, LNCS, Reading, UK, May 2006. Springer Verlag.
- [17] Marco Danelutto and Patrizio Dazzi. Joint structured/unstructured parallelism exploitation in Muskel. In *Proc. of ICCS 2006 / PAPP 2006*, LNCS. Springer Verlag, May 2006. to appear.
- [18] Marco Danelutto and Paolo Teti. Lithium: A structured parallel programming environment in Java. In *Proc. of ICCS: International Conference on Computational Science*, volume 2330 of LNCS, pages 844–853. Springer Verlag, April 2002.
- [19] Jan Dünneweber, Anne Benoit, Murray Cole, and Sergei Gorlatch. Integrating mpi-skeletons with web services. 33:787–794, 2005.
- [20] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. Quaff: efficient c++ design for parallel skeletons. *Parallel Computing*, 32(7):604–615, 2006.
- [21] Joel Falcou and Jocelyn Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. In G. R. Joubert, C. Bischof, F. J. Peters, T. Lippert, M. Bcker, P. Gibbon, and B. Mohr, editors, *Parallel Computing: Architectures, Algorithms and Applications (Proc. of PARCO 2007, Julich, Germany)*, volume 38 of NIC, pages 243–252, Germany, September 2007. John von Neumann Institute for Computing.
- [22] Intel. Threading building blocks. <http://www.threadingbuildingblocks.org>.
- [23] Herbert Kuchen. A skeleton library. In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, London, UK, 2002. Springer-Verlag.
- [24] Herbert Kuchen and Jörg 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*, pages 122–130, New York, NY, USA, 2002. ACM Press.
- [25] Herbert Kuchen and Jörg Striegnitz. Features from functional programming for a c++ skeleton library. *Concurrency - Practice and Experience*, 17(7-8):739–756, 2005.
- [26] Doug Lea. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43, New York, NY, USA, 2000. ACM.
- [27] Sun Microsystems. Java. <http://java.sun.com>.
- [28] OpenMP. <http://openmp.org/>.
- [29] Skandium. <http://skandium.niclabs.cl/>.