Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

© 2016 The authors and Open Channel Publishing Ltd. All rights reserved.

Communicating Connected Components: Extending Plug and Play to Support Skeletons

Kevin CHALMERS [a,1], Jon KERRIDGE [a] and Jan Bækgaard PEDERSEN [b]
[a] *School of Computing, Edinburgh Napier University*
[b] *School of Computer Science, University of Nevada, Las Vegas*

Abstract. For a number of years, the Communicating Process Architecture (CPA) community have developed languages and runtimes supporting message passing concurrency. For these we always provide a set of reusable processes called plug and play. These components provide a rich set of functions to the new CPA programmer, enabling them to develop applications. In this paper, we describe recent work in taking the plug and play ideology and applying it to the area of algorithmic skeletons. We have based our work on the RISC-pb^{2}l specifications of Danelutto et. al. to provide a base set of skeletal components, focusing on the communication behaviours they exhibit.

Keywords. algorithmic skeletons, components, plug and play

Introduction

Skeletal frameworks provide simple methods to develop parallel software. They are aimed at domain experts who are not parallel programmers. Skeletal frameworks do this by providing mechanisms for building parallel applications through reusable blocks. These ideas have become important as multi- and many-core architectures became the norm.

The Communicating Process Architectures (CPA) community (`www.wotug.org`) has been developing parallel applications for over 30 years. The CPA manifesto focuses on communication between processes via channels (message passing). The nature of CPA promotes the same block ideas that skeletal frameworks do. The CPA community have developed plug and play libraries as reusable blocks for common problems and communication patterns. Many of these plug and play blocks share similarities with common skeletal blocks.

In this paper we present work examining the communication semantics of algorithmic skeletons. Our work is built on the RISC-pb^{2}l [1,2] block specifications, extending these specifications by exploring the different communication patterns. The contribution of our work is the extension of the RISC-pb^{2}l building blocks to consider communication patterns, typing, and parallel behaviour.

We present the rest of our paper as follows. In Section 1 we provide some background and related work. In Section 2 we discuss our extensions to the RISC-pb^{2}l specifications. In Section 3 we provide a case study (building a concordance application) with these blocks in

[1]Corresponding Author: *Kevin Chalmers, School of Computing, Edinburgh Napier University, 10 Colinton Road, Edinburgh.* Tel.: +44 131 455 2484; E-mail: `k.chalmers@napier.ac.uk`.

a Groovy parallel library with relevant performance data. Finally in Section 4 we conclude and discuss future extensions.

1. Background and Related Work

Algorithmic skeletons are a technique for non-parallel programmers (domain experts) to exploit parallelism. An example skeleton is a pipeline which provides a template into which functions can be placed by the programmer. A number of such skeleton libraries exist – *eSkel* [3], *Muesli* [4], *Skandium* [5], and *SkeTo* [6]. González-Vélez [7] surveyed the skeletal libraries in 2010.

More recently, development of description languages for skeleton programming has been undertaken. A description language allows the programmer to detail the structure of their application as a collection of components interacting with each other. This work builds on Danelutto et. al's [1] RISC-pb^{2}l specifications.

1.1. RISC-pb^{2}l Building Blocks

RISC-pb^{2}l [1,2] approaches the problem of a design language by producing a limited set of general purpose building blocks. These blocks are divided into three types – *wrappers*, *combinators*, and *functionals*. Each type supports a different part of the parallel solution.

Wrappers describe how a function is to run (e.g. *sequential*, *parallel*).

Combinators describe communication between blocks – *N-to-1*, *1-to-N* and *feedback*. *N-to-1* and *1-to-N* include a communication policy to determine, such as *unicast*, *gather*, etc. Feedback describes a feedback loop with a given condition.

Functionals run parallel computations. Included are *parallel*, *Multiple Instruction, Single Data*, *pipeline*, *spread*, and *reduce*.

The individual components use a *"pure dataflow semantics"*. Each component has the same arity on its input and output channels, and channels carry copies of data, not references/aliases.

As a short example, taken from [1], a task farm can be described in RISC-pb^{2}l as follows:

$$TaskFarm(F) = \triangleleft_{Unicast(Auto)} \bullet [|\Delta|]_n \bullet \triangleright_{Gather}$$

Reading from left to right:

$\triangleleft_{Unicast(Auto)}$ a *1-to-N* communication using an *auto* selected *unicast* policy.

$\bullet$ separates pipeline stages.

$[|\Delta|]_n$ denotes *n* Δ computations in parallel. Δ is F in $TaskFarm(F)$.

$\bullet$ separates pipeline stages.

$\triangleright_{Gather}$ a *N-to-1* communication using a *gather* policy.

Further examples are available in [1,2]. The aim of using RISC-pb^{2}l is to reason about the parallelism at a high level, therefore enabling analysis of possible optimisations.

1.2. *Message Passing Based Skeletal Frameworks*

Although message passing is considered a core concurrency construct, little work has been undertaken investigating message passing in light of skeletons. Work has been undertaken with Erlang [8], but with the rise of other message passing languages (Go, Rust, etc.) there are rich avenues of investigation that can be undertaken. The aim of this paper is to explore potential benefits of developing skeletons in a message passing style due to the data flow nature skeletons generally promote.

2. Extending RISC-pb^{2}l

In this section we show how the RISC-pb^{2}l building blocks can be expressed in a process oriented language.

2.1. *Wrappers*

A wrapper is a function computing an input based on an output. In process oriented design we can represent both as a *black box* with an in data type `X` and out data of type `Y`.

In a process oriented language, all communication between entities (processes) uses synchronous, blocking communication on unidirectional channels. Therefore, any wrapper function takes in the reading end of a channel of type `X` (`in<X>`) and the writing end of a channel of type `Y` (`out<Y>`). Reading from a channel is denoted by `?`, and writing by `!`. Figure 1 provides a pseudocode semantics of this block.

```
**procedure** WRAPPER(F, in<X>, out<Y>)
    **while** true **do**
        in ? value
        out ! F(value)
```

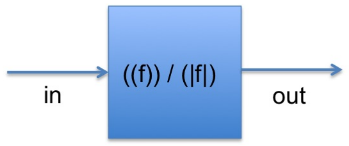

Figure 1. Wrapper Block.

Whether `f` is implemented in a parallel or sequential fashion is of no importance to the behaviour of the wrapper.

2.2. *Combinators - 1-to-N*

The $\triangleleft_{Policy}$ operator denotes a **1-to-N** data spreader. Depending on the $Policy$ it behaves differently. There are three distinct polices: $Unicast(p)$, $Broadcast$, and $Scatter$, where p is either RR (Round Robin) or $AUTO$, which directs the input to the output where a request token has been received.

2.2.1. $\triangleleft_{Broadcast}$

$\triangleleft_{Broadcast}$ can be described semantically as follows:

$$x \rightarrow \triangleleft_{Broadcast} \rightarrow \langle x, \ldots, x \rangle$$

where x is a single value. Let us assume that `in<X>` is a channel carrying values of type `X`, and `out<X>[n]` is an array of channels (0 through to n-1) also carrying values of type `X`. The API for a broadcast node can therefore be expressed as shown in Figure 2.

The `par for` is simply a regular for-loop done in parallel. Naturally, the `par for` could be a regular sequential `for`, in which case the broadcast would progress sequentially.

```
procedure BROADCAST(in<X>, out<X>[n])
    while true do
        in ? value
        par for i in 0..n-1 do
            out[i] ! value
```

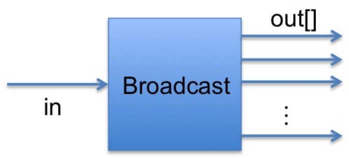

Figure 2. Broadcast Block.

2.2.2. $\lhd_{Scatter}$

The semantic description of $\lhd_{Scatter}$ is as follows:

$$V \rightarrow \lhd_{scatter} \rightarrow \langle v_1, \ldots, v_n \rangle$$

where V is a vector of size n with elements $v_1, \ldots, v_n$. We can represent that as the $Broadcast$ policy, with the difference in the types carried by the `in` channel, which has type `X[n]`. The corresponding behaviour can be seen in Figure 3.

```
procedure SCATTER(in<X[n]>, out<X>[n])
    while true do
        in ? value
        par for i in 0..n-1 do
            out[i] ! value[i]
```

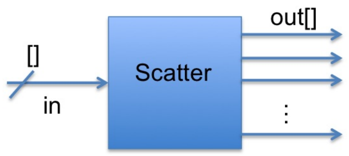

Figure 3. Scatter Array Block.

2.2.3. $\lhd_{Unicast(RR)}$

The semantics for $\lhd_{Unicast(RR)}$ is:

$$x \rightarrow \lhd_{Unicast(RR)} \langle \phi, \ldots, \phi, x, \phi, \ldots, \phi \rangle$$

A unicast using round robin simply alternates between the output channel ends in increasing order with wrap-around. The behaviour can be seen in Figure 4.

```
procedure UNICAST_RR(in<X>, out<X>[n])
    while true do
        for i in 0..n-1 do
            in ? value
            out[i] ! value
```

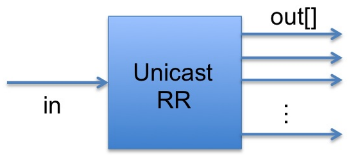

Figure 4. Unicast Round Robin Block.

Unlike $\lhd_{Scatter}$, which can send multiple inputs to multiple outputs in parallel, $\lhd_{Unicast(RR)}$ is not able to perform broadcast in parallel due to a single input value stream.

2.2.4. $\lhd_{Unicast(AUTO)}$

The AUTO option for a unicast is a little different. The receiver of the data must have sent a request first. Possible behaviours are illustrated in Figure 5.

`unicast_auto` requires a block to send a request (its index) to the unicast block to receive a message. We call this an *explicit* auto unicast. An implicit auto unicast is also possible if the language allows for output guards. An output guard is where the communication

```
procedure UNICAST_AUTO(in<X>, req<ℕ>, out<X>[n])
    while true do
        in ? value
        req ? idx
        out[idx] ! value
```

```
procedure UNICAST_AUTO_GUARDED(in<X>, out<X>[n])
    while true do
        in ? value
        select chan from out
            chan ! value
```

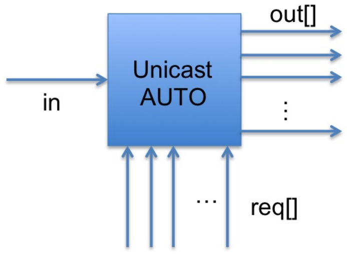

Figure 5. Unicast Auto Block.

is selected based on output availability. The requester does not send a request, but simply reads from the channel connected to the unicast block. The unicast block uses output guards to choose a ready receiver using a `select` statement, that chooses a ready output, blocking until then.

2.3. Combinators N-to-1

The $\triangleright_{Poliy}$ operator denotes an **N-to-1** collector. It gathers data from N channels and produces it, or a derivative, on the output channel. There are two policies *Gather* and *GatherAll*.

2.3.1. $\triangleright_{Gather}$

A gather function multiplexes n input channels onto an output channel:

$$\langle x_1, \ldots, x_n \rangle \rightarrow \triangleright_{Gather} \rightarrow (x_1 \succ x_2 \succ \ \ldots \succ x_n)$$

Figure 6 provides the general behaviour of the gather block.

```
procedure GATHER(in<X>[n], out<X>)
    while true do
        for i in 0..n-1 do
            in[i] ? value
            out ! value
```

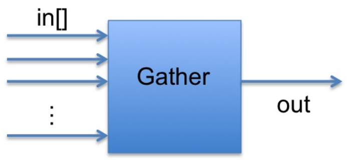

Figure 6. Gather Block.

It is not entirely clear from [1] if the semantics require **one** value from each sender (in order) or if the function should be a completely general multiplexer (implemented using a `select`). Either way is possible and both could be made available to the programmer. Note the lack of parallel in the `for` as it is not necessary to perform a gather.

2.3.2. $\triangleright_{Gatherall}$

The *Gatherall* policy requires the values be collected in a vector before the vector is written to the output:

$$\langle x_1, \ldots, x_n \rangle \rightarrow \triangleright_{Gatherall} \rightarrow [x_1, \ldots, x_n]$$

Figure 7 defines the behaviour of this block. Note that the reads can be performed in parallel.

```
procedure GATHERALL(in<X>[n], out<X[n]>)
    X value[n]
    while true do
        par for i in 0..n-1 do
            in[i] ? value[i]
        out ! value
```

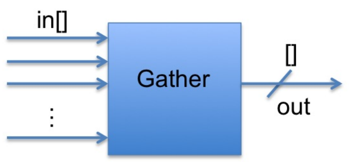

Figure 7. Gatherall Block.

2.4. Feedback Combinator $\overleftarrow{\Delta}$

The last of the combinators is **feedback**:

$$x \rightarrow \overleftarrow{\Delta} \rightarrow \begin{cases} \Delta(x) \rightarrow \mathit{feedback} \text{ if } \mathit{cond}(\Delta(x)) \\ \Delta(x) \rightarrow \mathit{out} \end{cases}$$

The feedback combinator reads from either an `input` or `feedback` channel (which has priority), passes the value to a block, (Δ), and depending on the outcome of a predicate function *cond* to the value $\Delta(x)$ writes the value to an `output` channel or back to the `feedback` channel.

We require a new helper block, `merge`, that controls the value passed to the surrounded block. `merge` and the block (Δ) are then run in parallel. Figure 8 presents the two blocks, with `block` used to denote an instantiation of Δ.

In [1] the value passed to *cond* is x and in [2] the value passed to *cond* is $\Delta(x)$. We have implemented the latter as it makes more sense, and from a process oriented design point-of-view, is a lot easier to implement. It should also be noted, that the Δ block must read and write values of the same type.

2.5. Functionals

2.5.1. $[|\Delta|]_n$

$[|\Delta|]_n$ computes the function specified by Δ on n different input values in parallel. The input arity of $[|\Delta|]_n$ is n and so is the output arity. The arity of Δ (both in and out) is 1. The semantic specification is:

$$\langle x_1, \ldots, x_n \rangle \rightarrow [|\Delta|]_n \rightarrow \langle \Delta(x_1), \ldots, \Delta(x_n) \rangle$$

Figure 9 presents the block algorithm.

```
procedure MERGE(in<X>, to_block<X>, from_block<X>, out<X>, cond)
    while true do
        in ? value
        to_block ! value
        from_block ? value
        while cond(value) do
            to_block ! value
            from_block ? value
        out ! value
procedure FEEDBACK(BLOCK, cond, in<X>, out<X>)
    to_block<X>
    from_block<X>
    par
        BLOCK(to_block, from_block)
        MERGE(in, to_block, from_block, out, cond)
```

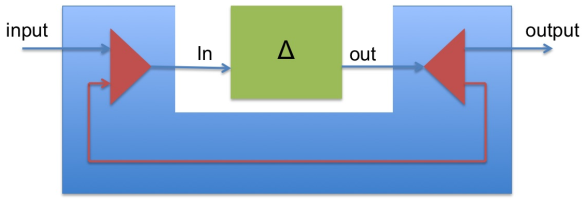

Figure 8. Feedback Block.

```
procedure PAR(BLOCK, in<X>[n], out<Y>[n])
    par for i in 0..n-1 do
        BLOCK(in[i], out[i])
```

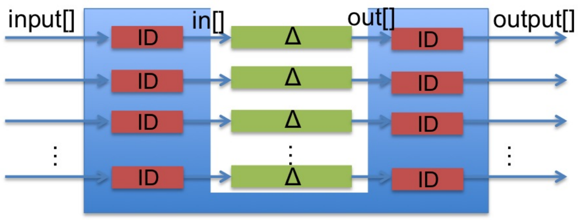

Figure 9. Par Block.

2.5.2. $[|\Delta_1, \ldots, \Delta_n|]$

This functional is no different than the $[|\Delta|]_n$ from a process oriented point-of-view; just pass `parallel-n` from the previous section a different set of channel ends – there are no restrictions on the type of computation as long as it reads an input and writes an output. For completeness, the semantic specification is:

$$\langle x_1, \ldots, x_n \rangle \rightarrow [|\Delta_1, \ldots, \Delta_n|]_n \rightarrow \langle \Delta_1(x_1), \ldots, \Delta_n(x_n) \rangle$$

The only other requirement is of course that instead of executing n copies of Δ we execute n different computations denoted by $\Delta_1, \ldots, \Delta_n$.

2.5.3. $\Delta_1 \bullet \ldots \bullet \Delta_n$

This is a pipeline computation. It combines output from one block to the input on the next:

$$x \to \Delta_1 \bullet \ldots \bullet \Delta_n \to \Delta_n(\ldots(\Delta_1(x))\ldots)$$

Figure 10 provides the behaviour. Note that the typing of the blocks has to be correct to ensure they can be connected together.

```
procedure PIPELINE(block[n], in<X>, out<Y>)
    internal[n - 1]
    par
        block[0](in, internal[0])
        par for i in 1..n-2 do
            block[i](internal[i - 1], internal[i])
        block[n-1](internal[n - 2], out)
```

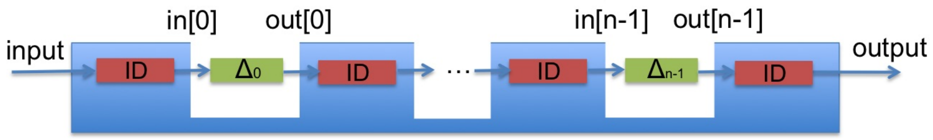

Figure 10. Pipeline Block.

2.5.4. $(f\triangleleft)$

Spread distributes the result of a function, f, on k-ary tree of blocks. There are n leaf blocks, requiring n/k applications of f. This requires a recursive block definition, provided in Figure 11. Note the recursion is parallel.

```
procedure SPREADER(F, param, k, out<X>[n])
    value ← F(param)                                        ▷ value has arity k
    if k = n then
        par for i in 0..n-1 do
            out[i] ! value[i]
    else
        par for i in 0..n-1 do
            SPREADER(F, value[i], k, out[n/k * i]...out[n/k * (i + 1)])
procedure SPREAD(F, k, in<X>, out<X<[n])
    while true do
        in ? value
        SPREADER(F, value, k, out)
```

Figure 11. Spread Block.

2.5.5. $(\triangleright f)$

Reduction performs the opposite process to `spread`, and therefore requires a recursive definition. This is provided in Figure 12.

```
procedure REDUCER(f, k, params[n])
    if k = n then
        return f(params)
    X values[n/k]
    par for i in 0..(n/k) - 1 do
        values[i] ← reducer(f, k, params[n/k * i]..params[n/k * (i + 1)])
    return f(values)
procedure REDUCE(f, k, in<X>[n], out<X>)
    X values[n]
    par for i in 0..n-1 do
        in[i] ? values[i]
    out ! reducer(f, k, values)
```

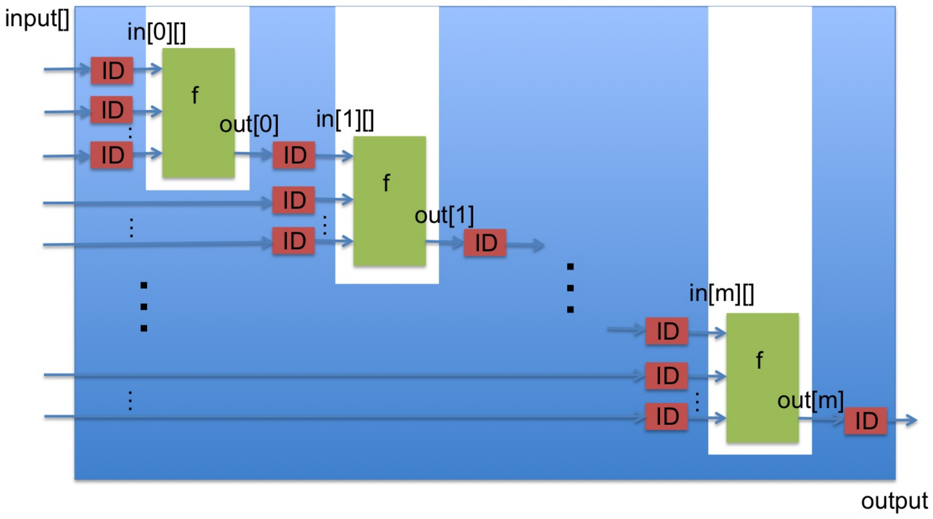

Figure 12. Reduce Block.

3. Case Study - Concordance

To test our Groovy Library based our RISC-pb^{2}l extensions (available at `https://bitbucket.org/jkerridge/org_jcsp_gpp_01`) we describe a particular case study - concordance.

3.1. The Problem

Given a text extract the location of equal word strings for strings of words of lengths 1..N in terms of the starting location of the word string in the text, provided the word string is repeated a minimum number of times.

3.2. The Solution

The solution comprises five stages as follows:

1. Processing word strings and in particular comparing them is complex and time consuming so we shall extract all the words from the text, removing unnecessary punctuation, and then calculate a corresponding integer value for each word based upon summing the letter codes for the word. We shall store these words and the list of word values.

2. Creates the sums of sequence of values for strings of length 1 to N, which generate N value lists.
3. Creates N maps each of which comprises a key based on a value from a valueList and an entry that contains all the locations where that value occurs. These maps are called indices map.
4. Disambiguates indices map because a key value may refer to different word sequences. Thus for each key in an indices map we extract the word sequence to which it corresponds, recall that the entry contains the location of the value in the text and we have stored the words. Thus we can build up a map comprising a key of a word string together with an entry comprising the locations of that specific word string. We shall call the N maps the words map.
5. The final stage is to output the words map in a human readable form ensuring that only strings that are repeated at least the minimum number of times are output. This is essentially a collect.

We provide two solutions - one that uses a group of pipelines (GoP) and another that uses a pipeline of groups (PoG). In RISC-pb^{2}l these are defined as:

$$GoP = ((emit)) \bullet \triangleleft_{Unicast(Auto)} \bullet [|2 \bullet 3 \bullet 4 \bullet 5|]_n$$

$$PoG = ((emit)) \bullet \triangleleft_{Unicast(Auto)} \bullet [|2|]_n \bullet [|3|]_n \bullet [|4|]_n \bullet [|5|]_n$$

3.3. Results

Our concordance solution is executed on the Bible, searching for strings of length N = 6 where the repeated string occurs at least twice. All the experiments were run on an Intel Core2 Quad Q8400 processor running at 2.67GHz with 8 GB memory. This provides a potential speedup of two times if the CPU is fully utilised. Our baseline sequential version of concordance executes in 28688 ms. Our results are presented in Table 1 and Table 2.

Groups	**Time (ms)**	**Speedup**
1	24281.5	1.181
2	23765.5	1.207
3	22211	1.292
4	21695.5	1.322

Table 1. Group of Pipelines Results.

Groups	**Time (ms)**	**Speedup**
1	24430	1.174
2	22984	1.248
3	21883	1.311
4	21734.5	1.320

Table 2. Pipeline of Groups Results.

As can be seen, there is little difference in performance between the GoP approach and the PoG approach. However, it should be noted that the concordance benchmark is highly dependant on I/O, and speedup reaches only 1.32×. This is due to the availability of processing blocks within the defined application. Our extended experiments do not provide a greater speedup than these figures.

4. Conclusions and Future Work

We have demonstrated that taking a process orientated view to skeleton block definition and composition provides a simple understanding of input and output typing, and the potential parallel behaviour within a block. We have also provided results of a concordance application using these blocks within a message passing Groovy library. Our results show promise for a message passing approach to skeleton composition, and other possible extensions can be made to further improve performance (for example via channel buffering).

We aim to take these definitions and implement them in other message passing languages and libraries. In particular, due to some of the typing requirements for recursive definitions, we aim to utilise C++ variadic templates to provide simple skeleton composition to the application programmer. We also aim to develop similar skeleton libraries in languages such as Go and Rust.

References

[1] M. Danelutto and M. Torquati. A RISC Building Block Set for Structured Parallel Programming. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 46–50. IEEE, February 2013.

[2] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Design patterns percolating to parallel programming framework implementation. *International Journal of Parallel Programming*, 42(6):1012–1031, December 2014.

[3] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, March 2004.

[4] P. Ciechanowicz and H. Kuchen. Enhancing Muesli's Data Parallel Skeletons for Multi-core Computer Architectures. In *2010 12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 108–113, September 2010.

[5] Mario Leyton and Jose M Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. pages 289–296. IEEE, February 2010.

[6] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A Library of Constructive Skeletons for Sequential Style of Parallel Programming. In *Proceedings of the 1st International Conference on Scalable Information Systems*, InfoScale '06, New York, NY, USA, 2006. ACM.

[7] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12):1135–1160, November 2010.

[8] Istvn Boz, Kevin Hammond, Viktoria Fords, Zoltn Horvath, Melinda Tth, Dniel Horpcsi, Tams Kozsik, Judit Kszegi, Adam Barwell, and Christopher Brown. Discovering parallel pattern candidates in Erlang. pages 13–23. ACM Press, 2014.