*October 2017*

# Towards Distributed Parallel Programming Support for the SPar DSL

Dalvan Griebler [a,1], Luiz Gustavo Fernandes [a]
[a] *Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil*

**Abstract.** SPar was originally designed to provide high-level abstractions for stream parallelism in C++ programs targeting multi-core systems. This work proposes distributed parallel programming support for SPar targeting cluster environments. The goal is to preserve the original semantics while source-to-source code transformations will be turned into MPI (Message Passing Interface) parallel code. The results of the experiments presented in the paper demonstrate improved programmability without significant performance losses.

**Keywords.** Parallel Programming Languages, Domain-Specific Language, Stream Parallelism, Algorithmic Skeletons, Parallel Patterns, Parallel Programming.

## 1. Introduction

Parallel programming for cluster architectures remains a challenging task for application programmers. For the most part, they need to deal with source code rewriting and low-level programming libraries when they try to achieve high-performance in distributed parallel processing. The current standard is the Message Passing Interface (MPI), which requires code modeling with explicit communication implementation, load balancing, processes synchronization, and data serialization (except MPI data types). Programmers must also understand the underlying architecture to efficiently exploit the parallelism.

This problem is well documented in the literature and though there are state-of- the-art initiatives such as the X10 [18], Chapel [5] and Charm++ [6], there is still a lack of high-level and productive alternatives such as proposed in SPar [13]. For instance, Charm++ is a machine independent programming system based on C++. The programmer exploits the parallelism in a object-oriented manner, and the runtime can run object and classes in parallel while the communication must be done through message passing [1]. On the other hand, X10 is a new object-oriented parallel programming language designed to work with the Asynchronous Partitioned Global Address Space (APGAS) model [15]. Moreover, targeting heterogeneous parallel architectures (CPU and GPU), there are related approaches that use C++ attributes (the same annotation mechanism of SPar) to abstract parallelism as part of a software engineering methodology [8,9]. However, they do not provide support for distributed parallel programming.

In contrast to these initiatives, we are proposing support for distributed parallel programming without changing the original semantics of SPar. In fact, our approach enables application programmers to add standard C++ annotations rather than having to rewrite their sequential code on shared memory architectures (multi-core) [12,11]. Our goal in

[1] Corresponding Author: dalvan.griebler@acad.pucrs.br

this paper is to present the proposal and demonstrate its high-level and structured parallelism abstraction as a new alternative for application programmers targeting cluster systems. In addition, we assess its programmability and performance through benchmarks that extend the initial experiments of the thesis dissertation [11], in which we used a different runtime library to generate parallel code without the support of ordering and on-demand scheduler optimizations.

In Section 2 of this paper we briefly highlight and compare related works. Next, we introduce SPar and explain our proposal for extending it with the support of distributed parallel programming on cluster systems (Section 3), including the language, skeleton library, and run-time parallelism. Finally, Section 4 discusses the results of our experiments and Section 5 concludes our work.

## 2. Related Work

MPI and OpenMP are the most well-known and widely used frameworks in parallel programming. MPI provides a low-level network message communication library for cluster environments. On the other hand, OpenMP provides higher level abstractions through annotations, where the compiler generates multi-threaded code for either multi-core or accelerators hardware [16]. In our case, as OpenMP is not suitable to exploit stream parallelism, we used FastFlow's runtime library [2] in the original multi-core version of SPar due to its functionality, flexibility, efficiency, and options. This runtime support is similar to that offered by MPI for distributed memory architectures.

Although FastFlow also supports distributed programming, its template-based library interface differs from those provided for shared memory architectures [17]. Yet, it does not have a processes launcher, which requires the application programmer to manually start the process in the cluster machines. Consequently, we chose MPI, which is considered the "*de-facto*" standard for message passing in parallel programming systems.

Charm++ [6], Chapel [5], and X10 [18] are examples of frameworks to increase productivity in high-performance computing that differ from our approach. Charm++ [1] is an extension of the C++ language that implements parallelism on top of a message passing programming model. Unlike other frameworks, the language is based on object migration and programmers interact through asynchronous object invocations. Chapel [4] is a completely new programming language with a block-imperative programming style. It provides parallelism abstractions by using the anonymous threads concept, which are implemented by the compiler and runtime system. X10 [15] on the other hand, is a new language based on Java on top of an APGAS (Asynchronous Partitioned Global Address Space) programming model, which offers a set of constructions to deal with a single memory space. Thus, these alternatives may be considered another way to exploit parallelism and may be used in place of MPI to generate parallel code.

## 3. SPar: a DSL for High-Level and Productive Stream Parallelism

SPar is an internal DSL embedded in the C++ language, capable of modeling high-level parallelism for streaming application [13,12,11]. It was implemented with the standard C++ attributes annotation mechanism [14]. Therefore, the programmer only needs to introduce annotations in the sequential source code rather than having to actually rewrite

it to exploit the parallelism of multi-core systems. The following sections will describe the SPar language, skeletal library, and runtime.

### *3.1. The SPar Language*

When using SPar, the programmer will deal with user-friendly abstractions that are closer to the streaming application domain's vocabulary. All properties are represented by language attributes in the annotated regions. An annotation is performed by using double brackets `[[id-attr, aux-attr, ..]]`, where a list of attributes could be specified if necessary. When at least the first attribute of the annotation is specified in the attributes' list, it is considered a SPar annotation. We named the first attribute identifier (ID) and the others auxiliary (AUX). A short description of SPar's available attributes are presented below (`ToStream` and `Stage` are ID and the others are AUX):

- `ToStream` is used to tag the beginning of the stream region. We can put it before any loop statement or compound statement code region.
- `Stage` is used to annotate inside of the `ToStream` annotation scope in the regions that perform computations for each stream item. There can be as many annotations as necessary.
- `Input(<list-var>)` is used to specify the data items that a given stream region will consume to compute. We can have one or more data variables as arguments.
- `Output(<list-var>)` is used to specify the data items that a given stream region will produce after computing. We can have one or more data variables as arguments.
- `Replicate(<val-int>)`is used to replicate a given stage region. This is only possible when stream items may be computed independently. The attribute may eventually receive an integer value as a parameter, but it can also remain empty. In this case, the programmer may use the environmental variable `SPAR_NUM_WORKERS` to determine the degree of parallelism.

In Listing 1, we can see SPar's usability with the Prime Numbers algorithm. Semantically, each `ToStream` must have at least one `Stage` annotation. In SPar, the code left between `ToStream` and the first `Stage` becomes a stream management stage, where the programmer needs to manage the end of the stream. We can easily achieve this by introducing a stop condition that breaks the loop. In the case of Listing 1, the end of stream is dictated by the annotated loop in line 3. Also, this is the only piece of code that can be left out of the `Stage` scope limits, which is inside a `ToStream` region. Both the ID attributes can use the loop body to define their scope as we performed the annotation in line 5. Another restriction is to use `Replicate` only with `Stage`.

```
1  int prime_numbers(int n){
2   int total=0;
3   [[spar::ToStream, spar::Input(n)]] for(int i=2; i<=n; i++){
4    int prime=1;
5    [[spar::Stage, spar::Input(i,prime), spar::Output(prime), spar::
       Replicate(workers)]] for (int j=2; j<i; j++){
6     if (i%j==0){ prime=0; break; }
7    }
8    [[spar::Stage,spar::Input(prime),spar::Output(total)]]
9    { total=total+prime; }
10  }
11  return total;
```

```
12 }
```

Listing 1: Prime Numbers algorithm annotated with SPar.

To ensure correct parallel code generation for cluster systems, we imposed two restrictions for input and output data specification, which are expressed by the application programmer with SPar's `Input` and `Output` attributes. The first one addresses vectors and arrays. It is necessary to specify their data size right after the variable name, as we have expressed in the following:

```
int data[SIZE];
[[spar::Stage, spar::Input(data[SIZE])]]
```

where `data` is the variable name and `SIZE` is the amount of memory allocated for the array. This is valid for the `Output` attribute as well. The second limitation is the accepted data types. They are listed in Table 1 where the first column describes the data types and others are the respective C/C++ data types tabulated. Observe in the third column that SPar is able to serialize standard C++ vector and array containers for all data types listed, which is not supported by default trough the MPI library. SPar simply abstracts these details, making it easier for the application programmer.

**Table 1.** Current data supported for `Input` and `Output` attributes.

| **Data** | **C Types** | **C++ Types/Containers** |
|---|---|---|
| Character | `char`, `unsigned char` | `std::string`, `std::vector`, `std::array` |
| Integer | `int`, `unsigned int` | `std::vector`, `std::array` |
| Floating points | `float` | `std::vector`, `std::array` |
| Double precision | `double`, `long double` | `std::vector`, `std::array` |
| Complement integer | `long`, `unsigned long` | `std::vector`, `std::array` |
| Boolean | | `bool`, `std::vector`, `std::array` |

The SPar compiler is designed to recognize our language and generate parallel code. It was developed by using the CINCLE (A Compiler Infrastructure for New C/C++ Language Extensions) support tools [11]. The compiler parses the code (which is specified by a compiler flag named `spar_file`) and builds an AST (Abstract Syntax Tree) to abstractly represent the C++ source code. Subsequently, all code transformations are made directly in the AST, where calls to our skeleton library are implemented with MPI (Section 3.2). Once all SPar annotations are properly transformed, another C++ code is generated, which is then compiled by invoking the GCC compiler to produce a binary output. The compiler interprets the proposed transformation rules of Section 3.3, aiming to convert SPar annotations into parallel patterns (Pipeline, Farm, and a combination of the two). Therefore, we present our skeleton library, source-to-source transformation rules, and runtime in the following sections.

### *3.2. The SPar Skeleton Library*

Our skeleton library was created to simplify parallel code generation for clusters, supporting Farm, Pipeline, and composed of Pipeline with Farm(s). The implementation of the skeleton communications as well as data serialization for the clusters listed in Ta-

ble 1 were performed using the C++ MPI Library. Both Farm and Pipeline are provided through a C++ template-based library, which was inspired by the FastFlow design principles and other skeleton libraries in the literature [7,10]. Also, our library provides similar patterns as the ones provided by FastFlow for multi-cores, and SPar sentences will be transformed into library calls.

The **Farm** pattern has three nodes called Emitter ($E$), Worker ($W$), and Collector ($C$), where each E and $C$ is associated with a process while $W$ may have more than one process running the same business logic code. $E$ is the stream item scheduler that will distribute the items to a given number of $W$s. Its default behavior is to send items in a round-robin fashion to the $W$s. When the stream ends, the $E$ broadcasts a special end of the stream message used to implement termination. All $W$s receive stream items and perform sequential operations in the items. The results are then sent to $C$, if necessary. Finally, $C$ is then able to gather stream items from all workers and compute its own business logic code.

For the **Pipeline** pattern there is only a Stage ($S$) node type that is classified as either the first, middle, or last stage. Each $S$ is always associated with a single process running a sequential computation. The classification of $S$ a is way of organizing and implementing the correct pipeline computation. Therefore, the first $S$ is the stream generator in the Pipeline, that is always sent to the next $S$. The last $S$ simply receives results from the Pipeline computation, the middle $S$ receives items from the previous $S$ as well as sends intermediate results to the next $S$.

The nesting of patterns allows us to build more complex topologies [3], called pattern compositions. For instance, $pipe(farm(E,W,C), farm(E,W,C))$ implements the combination of a Pipeline with two Farms as stages. Another is $pipe(S, farm(E,W))$, representing a Pipeline with one sequential stage and a Farm. $pipe(farm(E,W,C),S)$ is a Pipeline where the first stage is a farm and the last is a sequential stage. As can be observed, we initially focused mostly on creating pattern variants in such a way that the Pipeline was combined with Farms. The next section provides transformation rules from SPar annotations to parallel patterns, which were implemented by our skeleton library.

### *3.3. Transformation Rules*

The transformation rules target Farm and Pipeline parallel patterns for introducing parallelism on cluster architectures. These patterns were provided by our skeleton library in Section 3.2. We also used their functional semantics to represent the generated patterns in the transformation rules. The simplest way to annotate a C++ code is by using SPar such as provided in the left side of Rule 1. The functions `f1` and `f2` represent a sequence of commands that can be executed independently. Consequently, this annotation sentence will be transformed into a Pipeline, where `f1` is the first stage and `f2` is the second.

```
[[spar::ToStream]]{
  f1();
  [[spar::Stage]]{
    f2();
  }
}
```

$\Rightarrow$ `pipe(f1,f2);` **(Rule 1)**

With respect to Rule 1, when adding the `Replicate` attribute in the `Stage` annotation, you will have Rule 2. Therefore, we transform the annotation sentence in a Farm, where `f1` is assigned to the emitter (`E(...)`) and `f2` to the worker (`W(...)`) frameworks.

```
[[spar::ToStream]]{
  f1();
  [[spar::Stage,spar::Replicate(N)]]{
    f2();
  }
}
```
⇒ `farm(E(f1),W(f2));` **(Rule 2)**

Rule 3 has one more `Stage` annotation, where inside there is another sequence of commands represented by `f3`. In this kind of sentence, we still generate the Farm pattern, however, `f3` is assigned to the collector (`C(...)`), `f2` to the worker (`W(...)`), and `f1` to the emitter (`E(...)`).

```
[[spar::ToStream]]{
  f1();
  [[spar::Stage,
  spar::Replicate(N)]]{
    f2();
  }
  [[spar::Stage]]{
    f3();
  }
}
```
⇒ `farm(E(f1), W(f2), C(f3));` **(Rule 3)**

Unlike Rule 3, in Rule 4 the last `Stage` annotation also has a `Replicate` attribute. Consequently, a more complex transformation is performed to produce a Pipeline with nested Farm stages. This rule is optimized so that the first Farm is generated according to Rule 2, while the second Farm only has assigned `f3` to the worker (`W(...)`).

```
[[spar::ToStream]]{
  f1();
  [[spar::Stage,
  spar::Replicate(N)]]{
    f2();
  }
  [[spar::Stage,
  spar::Replicate(N)]]{
    f3();
  }
}
```
⇒ `pipe(farm(E(f1), W(f2)), farm(W(f3)));` **(Rule 4)**

Finally, the programmer may produce even more robust annotation sentences with respect to those discussed in this section. They may have more `Stage` annotations with different combinations of `Replicate`. Therefore, beyond the rules presented in this section, they will generate Pipelines that may or not have nested Farm stages. Moreover, note that our transformation rules are not taking into account the `Input` and `Output` attributes, which implies a limitation of not exploiting DataFlow parallelism.

### *3.4. The SPar Runtime*

Figure 1 is a high-level representation of a given annotated code (left side) with the respective runtime parallelism behavior (right side). Note that the code annotated is reading stream items infinitely, which are subsequently filtered and then written to the standard

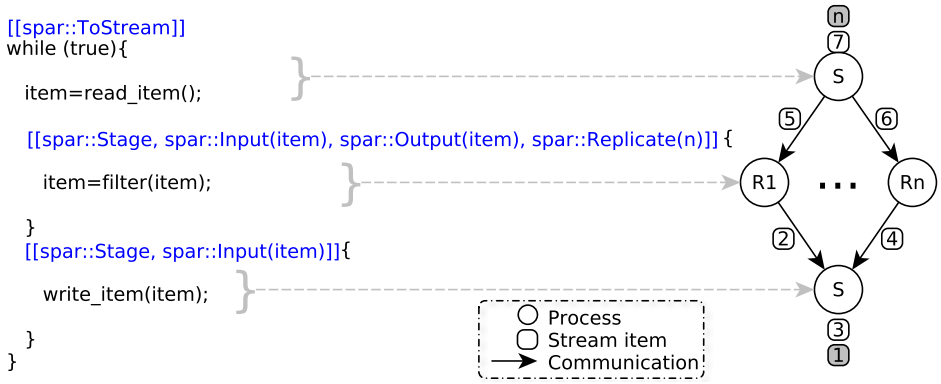



**Figure 1.** SPar runtime parallelism.

output. On the right side of Figure 1, the code between `ToStream` and the first `Stage` annotation is a single process running this portion of code.

In SPar, the `Stage` with a `Replicate` attribute is like spawning many MPI process from the skeleton library with the same portion of code (see the first stage Figure 1). The underlying runtime system will abstractly distribute stream items (specified through the `Input` and `Output` attributes) to these spawned processes, which uses the MPI library to communicate with the previous (by receiving) or next (by sending) stage. By default, the items are distributed in a round-robin fashion and no ordering is preserved. This is identified in Figure 1 through the labeled stream items, arriving unordered at the next stage when the previous one is replicated. Thus, the stream generator stage sends while the next stage is still working on the previous item that was sent. The last `Stage` is a single process, collecting items from all replicated processes of the previous stage. Observe that state-full operators are not managed by SPar, because it is up to the programmer not to replicate these kind of stages. SPar have other options through compiler flags that can be activated when desired (individually or combined) as follows:

- `spar_ondemand`: generates an on-demand stream item scheduler by switching the way processes communicate with the previous stage. Therefore, a new item will only be sent when the next stage has asked for another one.
- `spar_ordered`: makes the scheduler (on-demand or round-robin) preserve the order of the stream items.

## 4. Experiments

To assess our approach, we used two classic benchmarks. The first is the Prime Numbers algorithm previously highlighted in Listing 1. It basically receives a number as input and checks it by simply dividing it and adding up every prime number that is found. It allows us to demonstrate the importance of providing support for the on-demand scheduler and the possibility to model data parallelism as stream parallelism for cluster systems. The second is the well known program and library for data compression called `bzip2`[2]. We chose it to demonstrate SPar's feasibility with a real world streaming application. We also

[2] http://www.bzip.org/

compared the SPar generated code with `mpibzp2`[3], which is a native implementation of `bzip2` with MPI. A high-level representation of the annotated code with SPar is shown in Listing 2 for `bzip2` compression phase, since it is the most costly computation. This application first reads blocks of data from a given file. Then, the data are compressed and finally written to an output file. The stream region starts in line 4 where there is a `while` loop used to read blocks of data, which becomes the first stage of the pipeline. Subsequently, there is a stage that compresses the block of data (line 11 until line 15) and then a stage that writes it into the output file. Note that the main challenge is to find the stream region and identify the stages of the computation as well as the input and output dependencies, specifying the amount of data that a given variable stores.

```
1  int compress(...){
2   open(file);
3   [[spar::ToStream, spar::Input(hInfile, blockSize, bytesLeft)]]
4   while (bytesLeft > 0){
5    char* FileData=NULL, *CompressedData=NULL;
6    FileData = new char[inSize];
7    int rret = read(hInfile, (char *) FileData, inSize);
8    if (rret == 0) break; // end of the file
9    unsigned int outSize = (int) ((inSize*1.01)+600);
10   [[spar::Stage, spar::Input(inSize, FileData[inSize], outSize), spar::
       Output(outSize,CompressedData[outSize]), spar::Replicate(N)]]{
11    CompressedData = new char[outSize];
12    int ret = BZ2_bzBuffToBuffCompress(CompressedData, &outSize, FileData,
        inSize, BWTblockSize, Verbosity, 30);
13    if (FileData != NULL) delete [] FileData;
14   }
15   [[spar::Stage,spar::Input(outSize, CompressedData[outSize])]]{
16    int ret = write(hOutfile, CompressedData, outSize);
17    if (CompressedData != NULL) delete [] CompressedData;
18   }
19  }
20  close(file);
21  return 0;
22 }
```

Listing 2: High-level representation of `bzip2` compression function with SPar.

We used a homogeneous cluster configuration with four nodes to run the performance experiments. Each node had two Intel Xeon Quad-Core E5520 2.27 GHz Hyper-Threading processors, 16GB of memory, and two Gigabit-Ethernet network interfaces. We ran 10 repetitions for each replica size tested and plotted the standard deviation through error bars in the graphs in Figure 2. To evaluate the programmability, we measured the Source Lines of Code (SLOC), and Cyclomatic Complexity Number (CCN) (Table 2). As highlighted in Section 1, MPI requires code modeling with explicit communication implementation, load balancing, processes synchronization, and data serialization. In contrast, SPar preserves the source code semantics because it only requires standard C++ annotations to be inserted in the correct place. Another advantage is that it is only has five attributes, and all of them are flexible enough to model in different parallelization strategies. Moreover, it makes it possible to simply combine two compiler flags (`spar_ondemand` and `spar_ordered`). Thus, the SPar language can be learned quickly.

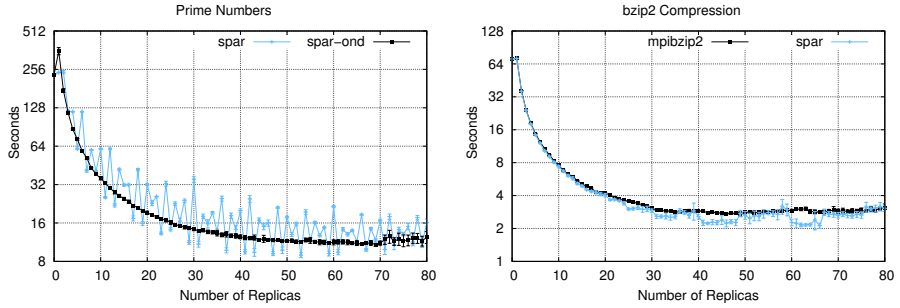
Beyond that, we also provide a quantitative analysis of the parallelized application versions using software engineering metrics such as CCN, and SLOC in Table 2. We also

[3] http://compression.ca/mpibzip2/

**Table 2.** Programmability results.

| | (a) Prime Numbers | | (b) bzip2 | |
|---|---|---|---|---|
| **Implementation** | ***SLOC*** | ***CCN*** | ***SLOC*** | ***CCN*** |
| serial | 110 | 26 | 101 | 27 |
| spar | 113 | 26 | 105 | 27 |
| mpi source | – | – | 227 | 56 |
| Generated | 250 | 45 | 209 | 50 |

measured the amount of code generated by the SPar compiler. Note that SPar provides lower SLOC and CCN than MPI in the original implementation of `bzip2`. SPar does not increase the CCN w.r.t sequential version since there is not significant code intrusion or re-factoring in Prime Numbers and `bzip2` benchmarks. As expected, the SLOC and CCN is almost double in the code generated by SPar.




(a) Execution time of the Primer Numbers algorithm. (b) Execution time of the `bzip2` application.

**Figure 2.** Performance results.

The graphs in Figure 2(a) and 2(b) present the execution times achieved by the Prime Numbers (counting the prime numbers up to 1,200,000) and `bzip2` (for compressing a text file of 104MB) benchmarks. Note that in both benchmarks the parallelization provided good scalability until reaching hyper-thread resources usage, where the number of replicas refers to the degree of parallelism. Figure 2(a) demonstrates that the `spar_ondemand` is needed to achieve better load balancing. In Figure 2(b) we used the `spar_ordered` to maintain the correct order that data are written in the compressed file. When comparing this SPar version with the native implementation of `bzip2` using MPI (`mpibzip2`), there is no significant difference.

## 5. Conclusions

This paper has presented the primary contributions of our ongoing research project. Firstly, we introduced the SPar DSL and its extended features to support distributed parallel programming for cluster systems (Section 3), which includes the skeleton library implementation and its runtime parallelism for target code generation. Lastly, Section 4 demonstrated the efficiency of the source-to-source transformation and the programmability of our solution. Therefore, our approach can be considered a suitable alternative for expressing parallelism and targeting cluster systems, because it avoids source code rewriting and provides efficient MPI code generation. In the future, we plan to support

other data types in the `Input/Output` attributes without the need to necessarily specify the data size. Furthermore, we will continue improving the parallelism support and abstraction to address current limitations (*e.g.*, DataFlow and data parallelism).

**Acknowledgements**
We thank CAPES and the Computer Science Graduate Program (PPGCC) from PUCRS for their partial financial support. Moreover, we thank Prof. Dr. Marco Danelutto for its fruitful discussions.

## References

[1] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 647–658, New Orleans, Louisana, November 2014. IEEE Press.

[2] M. Aldinucci, M. Danelutto, P. Kilpatrick, and M. Torquati. FastFlow: High-Level and Efficient Streaming on Multi-core. In *Programming Multi-core and Many-core Computing Systems*, volume 1 of *PDC*, page 14. Wiley, March 2014.

[3] A. Benoit and M. Cole. Two Fundamental Concepts in Skeletal Parallel Programming. In *International Conference on Computational Science (ICCS)*, volume 3515, pages 764–771, USA, May 2005. Springer.

[4] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[5] C. Chapel. The Chapel Parallel Programming Language. http://chapel.cray.com, July 2017.

[6] Charm++. Parallel Programming Framework. http://charmplusplus.org/, July 2017.

[7] M. Cole. Bringing Skeletons Out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, March 2004.

[8] M. Danelutto, J. D. Garcia, L. M. Sanchez, R. Sotomayor, and M. Torquati. Introducing Parallelism by using REPARA C++11 Attributes. In *24th Euromicro Inter. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, page 5. IEEE, February 2016.

[9] D. del Rio Astorga, M. F. Dolz, L. M. Snchez, J. D. Garca, M. Danelutto, and M. Torquati. Finding Parallel Patterns Through Static Analysis in C++ Applications. *The International Journal of High Performance Computing Applications*, 2017.

[10] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Software Practice & Experience*, 40(12):1135–1160, November 2010.

[11] D. Griebler. *Domain-Specific Language & Support Tool for High-Level Stream Parallelism*. PhD thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, June 2016.

[12] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. An Embedded C++ Domain-Specific Language for Stream Parallelism. In *Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing*, ParCo'15, pages 317–326, Edinburgh, Scotland, UK, September 2015. IOS Press.

[13] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters*, 27(01):20, March 2017.

[14] R. . ISO/IEC. Information Technology - Programming Languages - C++. Technical report, International Standard, Geneva, Switzerland, December 2014.

[15] J. J. Milthorpe. *X10 for High-Performance Scientific Computing*. PhD thesis, Australian National University, Australia, March 2015.

[16] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, New York, USA, 2003.

[17] A. Secco, I. Uddin, G. P. Pezzi, and M. Torquati. Message Passing on InfiniBand RDMA for Parallel Run-Time Supports. In *22th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 130–137, Torino, Italy, February 2014. IEEE.

[18] X10. Performance and Productivity at Scale. http://x10-lang.org/, July 2017.