High-Level Stream and Data Parallelism in C++ for Multi-Cores

Júnior Löff, Renato Barreto Hoffmann, Dalvan Griebler, Luiz Gustavo Fernandes
{junior.loff,renato.hoffmann}@edu.pucrs.br,{dalvan.griebler,luiz.fernandes}@pucrs.br
School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS)
Porto Alegre, Brazil

ABSTRACT

Stream processing applications have seen an increasing demand with the increased availability of sensors, IoT devices, and user data. Modern systems can generate millions of data items per day that require to be processed timely. To deal with this demand, application programmers must consider parallelism to exploit the maximum performance of the underlying hardware resources. However, parallel programming is often difficult and error-prone, because programmers must deal with low-level system and architecture details. In this work, we introduce a new strategy for automatic data-parallel code generation in C++ targeting multi-core architectures. This strategy was integrated with an annotation-based parallel programming abstraction named SPar. We have increased SPar's expressiveness for supporting stream and data parallelism, and their arbitrary composition. Therefore, we added two new attributes to its language and improved the compiler parallel code generation. We conducted a set of experiments on different stream and data-parallel applications to assess the efficiency of our solution. The results showed that the new SPar version obtained similar performance with respect to handwritten parallelizations. Moreover, the new SPar version is able to achieve up to 74.9x better performance with respect to the original ones due to this work.

KEYWORDS

Programming language, parallel programming, parallel patterns, algorithmic skeletons, C++ annotations, source-to-source code generation

ACM Reference Format:
Júnior Löff, Renato Barreto Hoffmann, Dalvan Griebler, Luiz Gustavo Fernandes. 2021. High-Level Stream and Data Parallelism in C++ for Multi-Cores. In *25th Brazilian Symposium on Programming Languages (SBLP'21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3475061.3475078

1 INTRODUCTION

Stream processing applications are becoming ubiquitous with the increased availability of sensors, IoT devices, digital financial systems and social networks. The viability of these class of applications is highly dependent on efficient, endless, and intense data exchange. They are executed in large scale distributed systems or single multi-core machines. However, extracting maximum efficiency from the underlying architecture demands that programmers write parallel code. This task is much more complex than writing sequential programs because it requires the developer to deal with low-level parallel details, such as schedulers, synchronizations, replications, etc. There has been an increasing trend in the computer science community towards higher-level abstractions for increasing productivity with minimal added performance overhead. For instance, C++ employed structured parallel programming as a solution to offer high-level abstractions with near negligible performance overhead. In fact, many state-of-the-art parallel libraries exploit this parallelism paradigm such as Microsoft PPL [15], Intel Threading Building Blocks (TBB) [22], and others.

Structured parallelism is based on high-level and ready-to-use parallel patterns. They are composable, parametric, and reusable abstractions that can be inter-connected to model complex data flows [3]. In the literature, parallel patterns are also known as algorithmic skeletons. The "parallel pattern" taxonomy was created when studies focusing in algorithmic skeletons [2] were unified with software engineering parallel programming methodologies [13]. There is a plethora of well-documented parallel patterns that are available for programmers to use (i.e. Map, Reduce, and Pipeline). However, efficiently applying them is still a challenge. This approach also introduced the notion of two different classes of programmers: *applications programmers* are those interested in using parallel patterns for modeling their specific-domain applications; and *system programmers* are those responsible for designing, implementing and optimizing parallel patterns. Indeed, applications programmers should not be responsible for deciding parallel implementation questions such as: which is the best parallel patterns; which is the best schedulers; which is the best degree of parallelism. These are tasks that system programmers should be in charge. Therefore, recent researches [4, 8, 14, 16, 18] are moving towards higher-level parallel abstractions.

Differently from others, SPar [8] is a domain-specific language (DSL) for expressing stream parallelism via code annotations. It provides five attributes for application programmers to learn and annotate sequential code with minimal code refactoring. The compiler decides for the parallel patterns and performs the hard work of parallelization. However, SPar is currently limited to coarse-grained stream parallelism. Modern stream processing systems may also display internal data parallelism computation. We highlight applications such as machine learning, natural resources exploration, financial systems, and others. In these cases, performance can be improved by combining stream and data parallelism. To the best of our knowledge, no previous work in the literature has combined both stream and data via code annotations. Therefore, in this work,

SBLP'21, September 27-October 1, 2021, Joinville, Brazil

ACM ISBN 978-1-4503-9062-0/21/09…$15.00
https://doi.org/10.1145/3475061.3475078

we investigate the feasibility of introducing high-level and efficient combined stream and data-parallel abstractions for Map and MapReduce using SPar as the use case. Therefore, we contribute for increasing SPar's language expressiveness, a new strategy for parallel code generation, a prototype implementation in the SPar compiler, and set of experiments with representative applications.

The remainder of this paper is organized as follows. Section 2 introduces our high-level strategy for automatic data-parallel code generation. The experimental analysis is discussed in Section 3 using applications extracted from both data and stream parallelism domains. Section 4 describes and discusses related researches. Finally, Section 5 concludes the study and presents future works.

2 INCREASING EXPRESSIVENESS

In this section, we introduce a parallel programming abstraction for automatic data-parallel code generation in C++ stream processing applications targeting multi-cores. To do so, we increase the expressiveness of SPar [8], which is a domain-specific language for expressing stream parallelism. It generates parallel code using FastFlow [1]. In the future, the strategies discussed in this work along with the compiler algorithm and transformation rules could be applied to other parallel programming abstractions besides SPar. Furthermore, the automatic parallel code generation is not bounded to FastFlow and it is possible to modify the final phase of our compiler algorithm to generate code for other runtimes like OpenMP [17] and TBB [22]. The outline of the section is the following. Section 2.1 introduces SPar and the basic concepts regarding its language and compiler. Section 2.2 presents the strategy we created for identifying data-parallel patterns in sequential code. Then, in the following we implement this strategy: (i) first by extending SPar's language in Section 2.3 and (ii) implementing a new compiler algorithm for source-to-source parallel code generation in Section 2.4.

2.1 Introduction to SPar

SPar (acronym for stream parallelism) is a domain-specific language for expressing stream parallelism in C++. The language was proposed in [6, 8] and presents a study towards parallelism higher levels of abstraction. For instance, choosing the most efficient parallel pattern or scheduling protocol should not be a high-level programmer responsibility. The intention of SPar is to offer a collection of intuitive attributes extracted from the stream processing domain and supported by C++ programming language. The programmer can use these attributes to annotate the data flow in the sequential code. Then, SPar's compiler is in charge of performing the semantic analysis of the annotated attributes and automatically generating a suitable parallel pattern using source-to-source transformations.

Currently, SPar's language contains five attributes for expressing stream parallelism in the code: (1) **ToStream** denotes where the data stream starts and ends; (2) **Stage** denotes where a stage/block of sequential code starts and ends; (3 and 4) **Input** and **Output**, as the name suggests, are the inputs and outputs of a `ToStream` or `Stage`; (5) **Replicate** is a special attribute for replicating a stateless `Stage` for parallel execution.

Listing 1 shows an example of stream processing application parallelized with SPar. The `ToStream` and `Stages` declared in lines 1, 3, and 5 represent identifier attributes. With this annotations, SPar's compiler will identify that the loop in line 1 represents a data stream. Therefore, each item of this data stream (each iteration) will be consumed by two sequential `Stages` (line 4 and 6). In this example, *i* represent the stream item. Since *i* is created outside the `Stage`, we use `Input` and `Output` to communicate data between the `Stages`. Finally, the `Replicate` attribute informs that this `Stage` can be computed in parallel.

```
1 [[spar::ToStream]] while(1){
2   i = read();
3   [[spar::Stage, spar::Input(i), spar::Output(i), spar::Replicate(4)]]
4   { i = filter(i);
5   }[[spar::Stage, spar::Input(i)]]{
6     write(i);
7 }}
```

Listing 1: Stream parallelism with SPar annotations.

SPar has been studied over the last few years. Some works already revealed that SPar is able to improve programmability with negligible performance cost [8, 9]. However, SPar targets only stream parallelism. For that, the SPar compiler generates code based on the Pipeline and Farm patterns, and their semi-arbitrary composition. The SPar language semantics are flexible and can be used to model and exploit parallelism in many application domains as shown in Listing 1. However, at the moment SPar's compiler performs automatic parallel code generation only using stream patterns. Unfortunately, in some applications the code generated becomes inefficient. In fact, we exhibit this throughout experiments later in Section 3.

In the ecosystem of stream processing applications, today's massively workloads are implying in many applications containing internal regions with intensive data processing. We highlight some applications: machine learning, which implements convolution mathematical operations; natural resources exploration, which computes CFD routines (Computational fluid dynamics), wildfires reporting, that analyzes high-definition satellite images, among others. For these applications, adding an extra internal level of parallelism can increase performance by improving resource utilization. However, SPar does not support efficient data parallelism exploitation.

Figure 1 illustrates a composition of stream and data parallelism. The Figure also highlights the goal of this work, where we investigate if it is possible to compose stream and data parallelism with a single language abstraction, and how efficient this can be. For that, we still keep stream parallelism for the coarse grain computation and internally exploit data parallelism. In this work, we target multi-core architectures, but in the future the data parallelism can target other specialized architectures, such as GPUs and FPGAs.

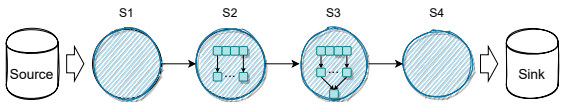

Figure 1: Composing stream and data parallelism.

2.2 High-level Data Parallelism

In this section, we describe the strategy that was created for identifying data-parallel patterns in sequential code. Later, we extend SPar using this strategy. We focus on the recurrent used data-parallel patterns: Map and MapReduce. A Map pattern can be introduced

when there is a known indexed data set locally stored in the multi-core architecture. This indexed set can be empty, but must be stored locally since data stored in network file systems or cloud storage usually is associated with severe I/O bottlenecks, which limits data parallelism. Consequently, these systems could be better modeled with stream parallelism using efficient schedulers.

For data parallelism, the Map and MapReduce patterns demand that the algorithm knows the size of the data set during execution time. For that, the algorithm must locate the boundaries of the indexed data set (start and end) and iteration step. With this information, if not explicit, the size can be calculated implicitly. For example: start=0, end=9, iteration_step+=1 (10 elements); or, start=2, end=64, iteration_step*=2 (6 elements).

The MapReduce parallel pattern is a special case of the Map in which iterations exhibit certain dependency. In this situation, all parallel replicas work towards solving a slice of the data in which results are later combined into a single output. To identify a MapReduce, first a Map has to be located and checked for dependencies. Then, data dependencies are analyzed for ensuring the computation can be implemented in terms of Reduce pattern (i.e. accumulations such as sums and multiplications). Some dependencies can not be reduced, because the order of execution modifies the result.

Listing 2 shows a high-level representation of Map (line 1) and MapReduce (lines 2 and 3) considering C++ `for` loop syntax. We have created a special notation to associate the C++ `for` with the parallel patterns. The `lhs` and `rhs` are the left- and right- hand sides data boundaries. We did not name them `start` and `end` since the loop can be both ascendant or descendant orders. The `it` stands for iteration step. The `lhs`, `rhs`, and `it` must be static and can not be modified after the parallel execution started. The `type` and `type2` can be a standard language type (i.e `int, long int, double`) or custom types (i.e. `struct, class`). Sometimes the type is not declared within the loop. Therefore, a compiler strategy should use the id (identifier) and traverse the abstract syntax tree (AST) to find where the variable was declared and extract the type. The `exp` and `op` stand for expression (i.e. `<, >, <=, >=, !=`) and operation (i.e. `-=, ++, *=, &=, |=`), respectively. These tokens are functions because they receive static values or variables (`rhs` and `it`) and modify this values according to the operator. For example, the tokens < or <= modify the `rhs` to either `rhs+0` or `rhs+1`. The loop (line 3) uses a shared variable. This is a common example of a MapReduce pattern that must implement synchronizations to avoid race-conditions and safely parallelize the code.

```
1 for(type id = lhs; id exp(rhs); id = id op(it)){ }
2 type2 id2;
3 for(type id = lhs; id exp(rhs); id = id op(it)){ id2=id2 op2(v)}
```

Listing 2: High-level Map and MapReduce representation.

Until now, we have only characterized the Map and MapReduce patterns using C++ syntax. However, this does not semantically checks if the sequential code can be safely parallelized. For example, if a loop has data dependency, as shown in previous Listing 2 at line 3, the automatic parallel code generation would be incorrect. Therefore, the data parallelism strategy in the compiler must implement checkers to test if a loop can be safely parallelized. There are two techniques that can be used in our context:

- **Auto-parallelization**: This technique uses compilers for analyzing the programming language syntax and semantic for validating functional correctness. Popular compilers for that are Cetus, Par4all, Rose, ICC and Pluto. However, a recent study [20] has performed a deep analysis to evaluate quantitative and qualitative aspects of these compilers using two bechmarks: PolyBench and NAS Parallel Benchmarks (NPB). In this study, the authors discovered and described many problems: starting from execution errors, missing parallel loops, incorrect semantics, and inefficient code generation. The authors suggest that the frameworks need more sophisticated techniques for parallel code analysis. In the mean time, frameworks and compilers can improve efficiency with user-oriented parallelism.
- **User-oriented parallelism**: This strategy is been used by almost any parallelism framework such as OpenMP, Threading Building Blocks, FastFlow, C++ Parallel STL, and others. While more sophisticated techniques for automatic parallel code detection are missing, the alternative is base the parallel abstractions in user-oriented information. Our work follows this approach with higher-level programming abstractions the programmer can use to express parallelism in the code. The compiler do not perform any static code analysis, instead we expect the programmer to provide the correct annotations.

In Sections 2.3 and 2.4, we present SPar's extension. First, we describe two new attributes we included to increase SPar's language expressiveness. Then, we describe a new compiler algorithm and transformation rules we implemented for SPar to support stream and data parallelism composition.

2.3 SPar's Language extension

We extend SPar's language proposing two new attributes to support data parallelism: `Pure` and `Impure`. The `Pure` attribute is a term already defined in functional programming for describing functions implemented in its purest shape. This means that the results (outputs) depend only on input parameters and computation has no side effect. In functional programming, the "pure" definition has many properties. In SPar, we limit pure functions to the parallelism property. Therefore, our pure definition carries the information that a block of code annotated with `Pure` can be executed in parallel with no restrictions.

A pure function must only implement a single effect: perform a processing based on input data, and return the result or store it in a non-shared location. Examples of side effects are:

- Modify an external shared variable;
- Read from a file or write to a file;
- External synchronizations, like `return`, `break`, and `socket`;

The next attribute included in SPar's language is the `Impure`. This attribute allows otherwise impure code regions to be annotated as pure. For example, if inside a function there is a single line of code with side effects, by definition all the function is considered not pure, or impure. Therefore, the `Impure` attribute shall be used to annotate that code region to purify the function, allowing parallelism transformations. In SPar, annotating a code region with `Impure` means that SPar will try to automatically implement

the required synchronization to allow parallelism. By default, an impure region is protected using locks. Before that, the compiler checks the code trying to detect ways to optimize the synchronization. In this work, we already detect reduce operations which we optimize using the MapReduce pattern. Other optimizations can be implemented in the future using speculative synchronization mechanisms or even other parallel patterns and their implications with the `Impure` attribute.

Listing 3 shows an example of parallelization using the matrix multiplication algorithm with the new attributes. Note that compared to SPar's original language (shown in Listing 1) it only requires the programmer to annotate the code using two extra attributes: `Pure` and `Impure`. Line 3 annotates a `Pure`, meaning the entire loop can be safely parallelized. However, lines 9 and 10 have side effects and we annotate this impure block of code using `Impure`.

```
1 [[spar::ToStream]]
2 for(long int i=0; i<SIZE; i+=1){
3   [[spar::Stage, spar::Pure, spar::Input(i), spar::Replicate()]]
4   for(long int j=0; j<SIZE; j++){
5     for(long int k=0; k<SIZE; k++){
6       matrix[i][j] += (matrix1[i][k] * matrix2[k][j]);
7       [[spar::Impure]]
8       {
9         sum += matrix[i][j];
10        sum_line[j] += matrix[i][j];
11      }
12  }}}
```

Listing 3: Matrix multiplication with new attributes.

Listing 4 provides a sequence of commands to show how the compiler transformations should be performed (explained in Section 2.4) from the annotation inserted by the programmer in previous Listing 3. `Pure` means the computation annotated by this attribute is in its purest form, has no side effects, and depends only on its inputs. Therefore, we encapsulate and abstract this block of code into a single function call `pure_function()`, as show in line 2. In this example, the parallelization is not safe yet, because it is conditioned to a compiler automatically purifying the impure block of code. Once the compiler purifies the impure region, it can parallelize the code using Map (line 1), and each parallel worker computes a partial result (line 2). At the end, the partial results are accumulated into a single output using the Reduce pattern (line 3).

```
1 MAP i = 0,1,...,SIZE
2   sum_local = pure_function(i);
3 REDUCE sum += sum_local;
```

Listing 4: High-level annotated attributes

2.4 SPar's Compiler Implementation

In this section, we introduced the modifications implemented in SPar's compiler to enable its extension towards data parallelism. Now, SPar supports automatic parallel code generation for different parallelism domains. Figure 2 illustrates a flowchart of our implementation methodology, which is discussed in sequence.

2.4.1 Semantic Analysis. We start by extending SPar's compiler to support the two new attributes already included in SPar's language: `Pure` and `Impure`. The first compiler step traverses the AST and performs a semantic analysis verifying the annotated attributes correctness. For example, `Impure` must be declared within a `Pure`, and `ToStream` must be the outermost attribute annotation. During the AST traverse, the compiler also gathers important information about the annotations: information where `ToStream` was declared, how many internal `Stages` or `Pures`, `Input` and `Output` variables, and others. Then, the next step of the compiler uses this information combined with pre-defined transformations rules and definitions, to converge in a suitable parallel pattern.

2.4.2 Transformation Rules. In the second compiler step, we have proposed new definitions and transformation rules to support source-to-source transformations. The current SPar definitions and transformation rules can be found in [8]. The new transformation rules we included can be classified in two groups. First, two basic transformation rules, where a code annotated with SPar can be transformed in Map or MapReduce patterns. The main goal with this data parallelism strategy is to improve SPar's automatic parallel code efficiency. Second, two composable transformation rules that target parallel code generation using arbitrary pattern composition. These are a set of transformation rules supporting the composition of stream-parallel patterns (Pipeline and Farm) with data-parallel patterns (Map and MapReduce).

2.4.3 Information Extraction. Once the compiler determines the parallel pattern, or the composition of them, it performs data extraction. This step is responsible for executing compiler routines that traverse the C++ abstract syntax tree (AST) and gather information regarding the new Map and MapReduce patterns. Data parallelism is much more restricted than stream parallelism. For example, data-parallel applications can be implemented using stream parallelism while the reverse is not possible. Also, data parallelism patterns require more information like knowing the data set. Stream parallelism is more flexible and can deal with infinite data. Therefore, in this compiler step, both the information extraction and syntax analysis routines can abort the data-parallel patterns generation. If this proceeds, our compiler algorithm resumes the original SPar execution flow, even it only generates stream parallelism. In Section 2.2, we presented our strategy and have defined the basic data required to generate the Map and MapReduce patterns based on C++ syntax. This step must extract essential data such as identifiers, variable types, indexed set size, and others. To extract these information, we have implemented parses based on standard C++17 ISO [11].

2.4.4 Parallel Code Generation. Finally, if data is correctly extracted, we generate the parallel patterns using FastFlow runtime calls. Listing 5 shows a slice of the parallel code automatically generated by our compiler algorithm. The code represents the matrix multiplication algorithm annotated with SPar in previous Listing 3. This example gives an idea how much parallelism details a programmer must deal with to parallelize an application, even if the application is as simple as a matrix multiplication. Lines 1 and 2 store a copy of the original values that require synchronization. In this case, memory copy is necessary because the reduction is performed using an array. Line 3 initializes the reduction variables since FastFlow requires the initial value and an empty reference. Lines 4 to 16 implement FastFlow's MapReduce parallel pattern schema, which is based on C++ lambda functions. We replace the lambda function by the `pure_function()` block annotated with `Pure`. Finally, in lines 17 and 18 the accumulated reduction values are assigned to the original variables.

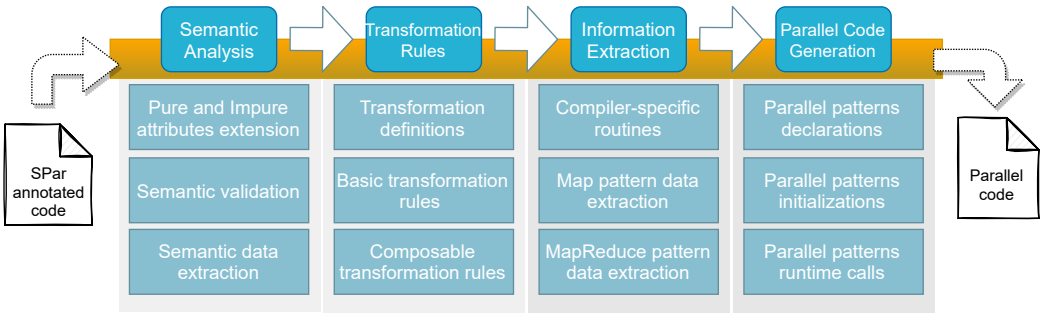

Figure 2: Implementation methodology used to implement the new compiler algorithm in SPar.

```
spar_global.sum = sum;
memcpy(spar_global.sum_line,sum_line,sizeof(int)*SIZE);
spar_reduce reduce_clean, reduce(spar_global.sum, spar_global.
    sum_line);
spar_pf->parallel_reduce(reduce,reduce_clean,0,SIZE+0,1,
[&] (long int i, spar_reduce & reduce) {
 for(long int j = 0; j < SIZE;j++){
  for(long int k = 0; k < SIZE;k++){
   matrix[i][j] += (matrix1[i][k]*matrix2[k][j]);
   reduce.sum = reduce.sum+matrix[i][j];
   reduce.sum_line[j] += matrix[i][j];
  }
 }
},
[&] (spar_reduce & reduce, const spar_reduce spar_aux_reduce) {
 reduce += spar_aux_reduce;
});
sum = reduce.sum;
memcpy(sum_line, reduce.sum_line, sizeof(int)*SIZE);
```

Listing 5: MapReduce parallel code generation.

3 EXPERIMENTS

The experiments were conducted to assess the efficiency of the compiler algorithm. The tests are divided in two parts. In Section 3.1, we evaluate how the new compiler algorithm performs when automatically generating parallel code for data parallelism applications. For that, we selected a representative benchmark for multi-cores in C++, the NAS Parallel Benchmarks (NPB) [10, 12]. The NPB contains eight benchmarks extracted from computational fluid dynamic (CFD) domain. The workload represents heavy mathematical computation that can be easily found in popular scientific HPC applications. In Section 3.2, we evaluate arbitrary pattern composition of different parallelism paradigms, composing stream and data parallelism. The applications used in this experiments are to obtain a first impression of how pattern composition behaves. We also have parallelized the Mandelbrot Set [6] and Lane Detection [9] applications. The former application belongs to the mathematical visualization set while the latter belongs to autonomous vehicle systems.

3.1 Data Parallelism

The experiments were executed in a machine equipped with 64 GB of RAM and a processor Intel(R) Xeon(R) Silver 4108 CPU @ 1.80GHz with 8 cores and hyper-threading, adding up to 16 threads. We used a machine equipped with a single processor to avoid communication bottleneck between the processors since some NPB benchmarks stress this feature. The operational system was Ubuntu 18.04 with kernel 4.15.0-123-generic. We used GCC 7.5 with -O3 flag enabled. The FastFlow version was v3.0.0. In our experimental setup, we used NPB's class B (parameters available on website [1]). The tests were executed from 1 up to 16 (maximum degree of parallelism). The execution was repeated 5 times and the graphs represent the average value. The standard deviation was plotted using error-bars and may not be visible when the value is negligible.

The graphs in Figure 3 summarize our results. In these graphs, the x axis show the degree of parallelism while the y axis shows the total execution time in seconds using logarithmic scale 2. We compare the original SPar and the new SPar+ proposed in this work against handwritten and manually optimized FastFlow versions obtained from [10, 12]. We named SPar+ with the purpose of differentiating our version with respect to original SPar. In Figure 3a, the results are similar since EP is an embarrassingly parallel application containing a single parallel loop. However, there is already a slight advantage when using data-parallel pattern over stream patterns. This can be seen in the maximum degree of parallelism (16 threads), where SPar+ is 3.9% faster than SPar using the same annotations in the code. And the main reason is that the stream patterns use an extra thread for scheduling, the Emitter. Therefore, in maximum degree of parallelism there are actually 17 threads (16 parallel workers + 1 scheduler) competing for resources.

In Figures 3b and 3c, this is more evident as it becomes clear that the stream parallelism generated by SPar is inefficient in this type of applications. The main problem is that the scheduler is a bottleneck, since it has to orchestrate a fine-grained workload. The tasks are low computational intensity and working threads finish very quickly, becoming idle until a new task arrives from the scheduler. The execution time increases with higher degrees of parallelism, considering there are more threads waiting and time to receive new tasks increases. The difference between SPar versions is up to 1.54x in FT and up to 74.9x in CG.

We were not able to implement the other three applications with SPar. The reason is SPar exhibits limitations such as code refactoring and variable capturing that preclude parallelizing NPB's benchmarks. On the other hand, the new language and compiler algorithm increased SPar+ expressiveness and flexibility, and we were able to fix those limitations while enabling parallelism for all applications. The results are illustrated in Figures 3d, 3e and 3f, and

[1] https://www.nas.nasa.gov/publications/npb_problem_sizes.html

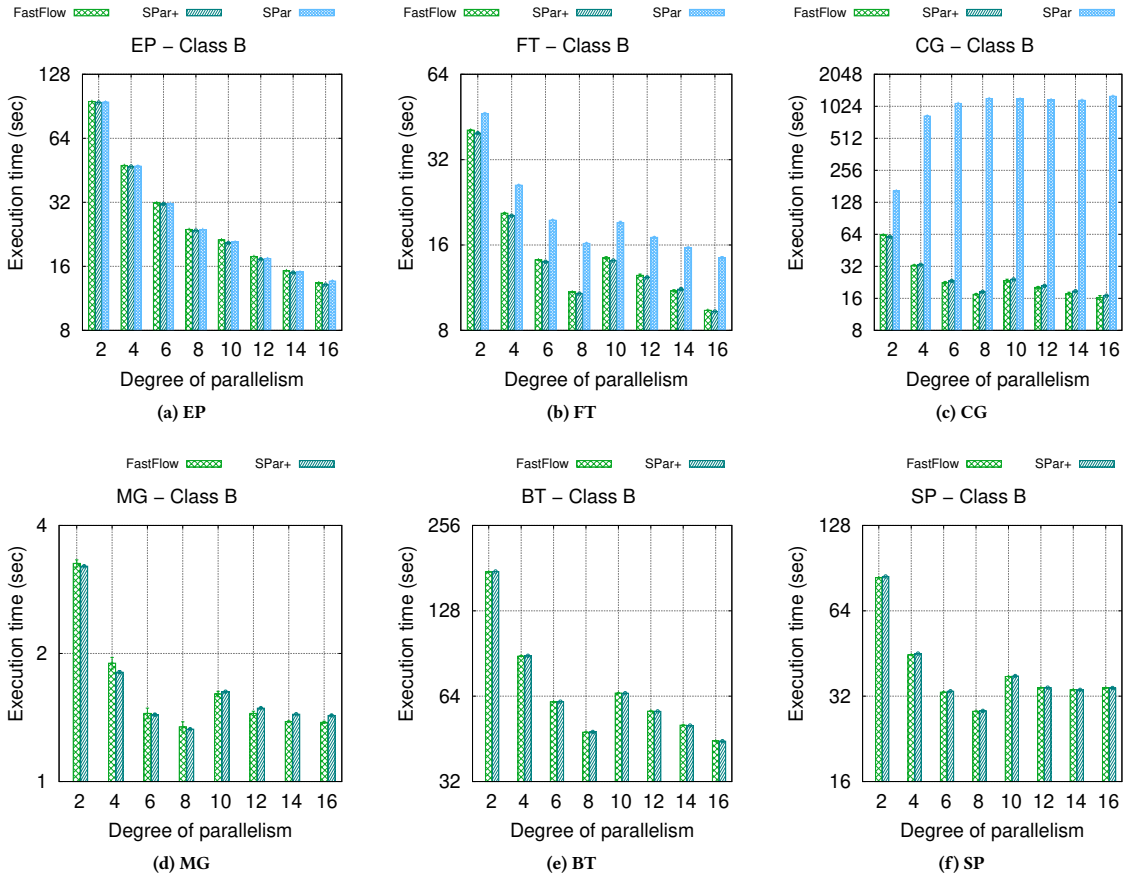

Figure 3: NPB with handwritten parallelizations [12] vs. automatic parallelizations using SPar and SPar+.

revealed that SPar+ can achieve similar performance to handwritten FastFlow parallelizations. The major differences between SPar+ and FastFlow are in CG and MG. In CG, FastFlow uses dynamic scheduling, whose optimal configuration was obtained through experimental tests. SPar+ by default applies static scheduling. In MG, the difference is that SPar+ implements one less MapReduce pattern than FastFlow. The reason is that currently SPar+ only supports summation reduce operations while MG implements a reduction of type `max`. It is worth noting that in some graphs the execution time increases in the transition from degree of parallelism 8 to 10 due to the hyper-threading technology, which introduces workload balancing issues.

In this section, our goal was to evaluate SPar using data-parallel applications. Since by default SPar generates parallel code to the FastFlow runtime, we expected SPar+ to execute similar to handwritten FastFlow parallel programs. The experiments have revealed that SPar+ achieves similar results at most 3.8% lower than handwritten parallelizations. In the best case scenario, SPar+ achieves up to 2% better performance with respect to FastFlow, and up to 74.9x better performance than SPar. SPar+ was slightly better than handwritten parallelism in some situations because we used C++ mechanisms such as operator overloading to abstract reductions.

3.2 Stream and Data Parallelism Composition

In this section, we assess the performance of different parallelism paradigms. Considering the nature of the applications, it is known that the performance may not improve as we include an extra level of parallelism. Some applications have internal unbalanced work and fine grained workloads. The goal is investigating the behavior of parallelism composition and how to accommodate different parallel patterns, which may bring new insights to this research domain.

The experiments were executed in a machine equipped with 24 GB of RAM and two processors Intel(R) Xeon(R) CPUE5-2620 v3 @ 2.40GHz with 6 cores each and support to hyper-threading, adding up to 24 threads. The operational system was Ubuntu 18.04 with kernel 4.15.0-112-generic. We used GCC 7.5 with the -O3 flag enabled. The execution was repeated 5 times and the graphs show the average results. The basis of the graph (x and y axis) are different configurations for data and stream degrees of parallelism. The execution time is represented through intensity colors, where red are higher execution times while blue are lower execution times.

Figure 4 illustrates the results obtained with the Mandelbrot Set application. The first behavior we observed is that stream parallelism scales better than data parallelism. In fact, the best execution time was obtained with degree of stream parallelism 20 and degree

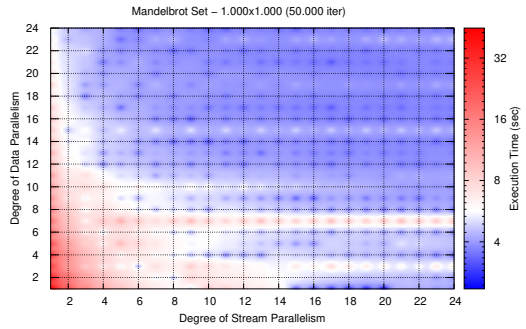

Figure 4: Performance results using SPar+ in Mandelbrot Set.

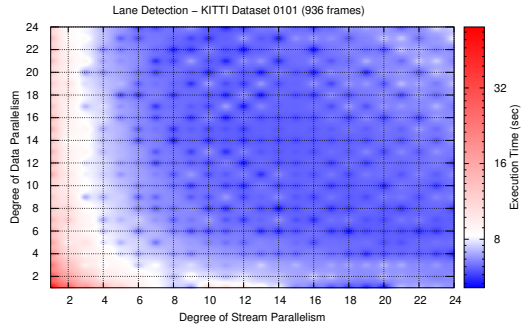

Figure 5: Performance results using SPar+ in Lane Detection.

of data parallelism 1. Furthermore, it is worth highlighting that data parallelism limits the total execution time of the application, since varying the stream degree has no impact at all, and execution time is the same. In the Mandelbrot Set application we observe unbalanced workload. However, this is an interesting result because it suggest that a second nested level of parallelism can balance workloads derived from originally unbalanced applications.

Figure 5 illustrates the results we obtained with the Lane Detection application. Again we observe that stream parallelism scales better than data parallelism. However, now the best execution time was obtained combining different degrees of parallelism. The best results was using degree of stream and data parallelism 14 and 8, respectively. Differently from the previous application, the results are not limited by data parallelism. In the contrary, there are many points along the grid that obtained much lower execution time than their neighbors.

In this section, our goal was to analyze the behavior when composing parallelism from different paradigms. The experiments have revealed that there are opportunities to improve parallelism efficiency by exploiting this approach. The most important characteristic we are interested in is to improve resource usage. In Mandelbrot Set the best load balancing was obtained with higher degrees of stream parallelism while in Lane Detection the best load balancing was obtained combining different degrees of stream and data parallelism.

There are open research questions towards combining different levels of parallelism. However, the literature still lacks solutions that exploit compositions between different parallelism paradigms. For example, using stream parallelism combined with data parallelism to exploit both highly scalable and distributed cloud environment (stream parallelism) and internally implement fine-grained parallelizations for multi-cores (data parallelism). Our analysis is limited to machine resources available. There are other much powerful multi-core machines delivered by the industry with hundreds of processing cores. We expect that our approach of combining data and stream parallelism will present more benefits in these larger multi-core machines due to the massive parallelism available.

4 RELATED WORK

In the literature, many works are moving towards increasing the level of parallelism by offering high-level abstractions with almost no performance cost. Table 1 summarizes such related works. We focus on works targeting stream processing and multi-core architectures. Others [21] proposed a DSL named StreamIt, which introduces a new language and compiler. Similar to SPar, StreamIt offers a high-level interface for expressing stream parallelism and generates automatic parallel code using source-to-source transformations. However, StreamIt requires to learn a new syntax and language based on Java while SPar uses C++11 attributes, which are fully recognized and represented in the standard language AST (abstract syntax tree). GrPPI [4] (Generic Reusable Parallel Pattern Interface) instead offers a parallel programming abstraction with generic parallel patterns. For that, the programmer only instantiates parallel patterns once, and chooses at compilation-time for which runtime GrPPI should generate parallel code. In GrPPI, the programmer is responsible for identifying the best pattern refactoring the code to it manually. In contrast, SPar automatically decides and generates a parallel pattern.

Table 1: Comparison between related works.

Work	API	Programming Language	Runtime	Supported Architectures
StreamIt [21]	External domain specific language	Java	Custom	multi-cores and clusters
GrPPI [4]	template library	C++	FastFlow, TBB, OpenMP and C++ Parallel STL	multi-cores
OpenStream [19]	pragma compilation directives	C/C++	POSIX Threads	multi-cores
OmpSs [5]	pragma compilation directives	C/C++	Custom	multi-cores, clusters and accelerators
WindFlow [14]	Parallel library	C++	FastFlow	multi-cores and accelerators
PiCo [16]	C++ domain specific language	C++	FastFlow	multi-cores
SPar [7]	C++ domain specific language	C++	FastFlow, TBB	multi-cores, clusters and accelerators

OpenMP is the *de facto* standard for data parallelism in C++ and multi-core architecture. Some works notice the difficulties for developing stream processing applications using OpenMP and proposed extensions. OpenStream [19] extended OpenMP by offering additional support to task parallelism and Pipelines. This tool is based on pragma compilation directives used by the programmer to annotate dependencies between tasks and provide information about the data flow. OmpSs [5] is another high-level language that extended OpenMP. The authors propose a new syntax to annotate parallel code also based on pragma directives. Besides, OmpSs support heterogeneous programming (GPUs and FPGAs). In recent

versions OpenMP also supports a *task-based model* taking inspiration from OmpSs programming model. Developers are equipped with pragmas for creating tasks and linking them with dependencies. Instead of using pragmas, SPar leverages C++ attributes that are available for any compiler that recognizes C++11 and newer. Attributes are fully represented in the C++ grammar.

WindFlow [14] introduces a specific-domain template library for leveraging data stream parallelism in multi-core and heterogeneous architectures. The library is based on the stream domain and offers domain-specific operators. However, the API requires the programmer to learn domain-specific approaches to implement the most efficient data flow. PiCo [16] is a DSL that tries to simplify parallelism with respect to other Big Data solutions. However, although PiCo proposes a high-level abstraction over FastFlow (similar to SPar), the syntax used by PiCo is still very similar to FastFlow. Differently, SPar clearly distinguishes between low-level parallelism optimizations and high-level abstractions. SPar's main goal is to support application programmers achieving higher levels of productivity and performance using high-level parallel abstractions.

5 CONCLUSION

In this paper, we investigated the feasibility of extending the expressiveness and flexibility in a high-level parallelism abstraction to support data and stream parallelism. For that, we used SPar as the use case, adding two new attributes to SPar's language and implementing a new algorithm for SPar's compiler. Now, SPar is able to generate parallel code more efficiently because it can select parallel patterns between stream patterns (Pipeline and Farm), data patterns (Map and MapReduce), and arbitrary composition of them. In the experiments, we evaluated the new SPar version using representative applications extracted from stream and data parallelism domains. Results evaluating only the data parallelism domain have revealed that SPar's new compiler algorithm can improve performance by up to 74.9x compared to old SPar compiler. Moreover, the performance of automatic parallel code generation is similar to handwritten parallelization, varying between 3.8% slower and 2% faster than FastFlow. Also, results evaluating stream and data parallelism composition have revealed that there is opportunity for exploiting fine-grained data parallelism inside stream parallelism stages for improving resource usage and raising scalability.

As future work, we plan to evaluate the new SPar version using other complex stream processing applications and larger multi-core machines. We expect to conduct more experiments to better understand the advantages or disadvantages of combining stream and data parallelism. Furthermore, we intend to investigate the necessity for including more parallel patterns, besides Pipeline, Farm, Map and MapReduce. Also, supporting automatic parallel code for combining different architectures such as clusters of multi-cores, or multi-cores with GPUs.

ACKNOWLEDGMENTS

We would like to acknowledge the support of LAD-PUCRS, GMAP research group and PUCRS university. This research is partially funded by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, FAPERGS 05/2019-PQG project ParAS (Nº 19/2551-0001895-9), FAPERGS 10/2020-ARD project SPar4.0 (Nº 21/2551-0000725-7), Universal MCTIC/CNPq Nº 28/2018 project SParCloud (Nº 437693/2018-0), and MCTIC/CNPq call 25/2020 (Nº 130484/2021-0).

REFERENCES

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. *Fastflow: High-Level and Efficient Streaming on Multicore.* John Wiley & Sons, Ltd, Chapter 13, 261–280. https://doi.org/10.1002/9781119332015.ch13

[2] M. Cole. 1989. Algorithmic Skeletons: Structured Management of Parallel Computation.

[3] M. Danelutto, D. D. Sensi, G. Mencagli, and M. Torquati. 2019. Autonomic management experiences in structured parallel programming. In *2019 International Conference on High Performance Computing Simulation (HPCS).* 336–343.

[4] David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. 2017. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience* 29, 24 (2017), e4175. e4175 cpe.4175.

[5] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *PPL* 21, 2 (2011), 173–193.

[6] Dalvan Griebler. 2016. *Domain-Specific Language & Support Tool for High-Level Stream Parallelism.* Ph.D. Dissertation. Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil.

[7] Dalvan Griebler. 2016. *Domain-Specific Language & Support Tool for High-Level Stream Parallelism.* Ph.D. Dissertation. Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil. http://tede2.pucrs.br/tede2/handle/tede/6776

[8] Dalvan Griebler, Marco Danelutto, Massimo Torquati, and Luiz Gustavo Fernandes. 2017. SPar: A DSL for High-Level and Productive Stream Parallelism. *Parallel Processing Letters* 27, 01 (March 2017), 1740005.

[9] Dalvan Griebler, Renato B. Hoffmann, Marco Danelutto, and Luiz Gustavo Fernandes. 2017. Higher-Level Parallelism Abstractions for Video Applications with SPar. In *Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing (ParCo'17).* IOS Press, Bologna, Italy, 698–707.

[10] Dalvan Griebler, Junior Loff, Gabriele Mencagli, Marco Danelutto, and Luiz Gustavo Fernandes. 2018. Efficient NAS Benchmark Kernels with C++ Parallel Programming. In *26th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP) (PDP'18).* IEEE, Cambridge, UK, 733–740.

[11] ISO/IEC 14882:2017 2017. *ISO/IEC 14882:2017 - Programming languages – C++.* Standard. International Organization for Standardization, Geneva, Switzerland.

[12] Júnior Löff, Dalvan Griebler, Gabriele Mencagli, Gabriell Araujo, Massimo Torquati, Marco Danelutto, and Luiz Gustavo Fernandes. 2021. The NAS parallel benchmarks for evaluating C++ parallel programming frameworks on shared-memory architectures. *Future Generation Computer Systems* (2021).

[13] Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming* (first ed.). Addison-Wesley Professional.

[14] Gabriele Mencagli, Massimo Torquati, Dalvan Griebler, Marco Danelutto, and Luiz Gustavo L. Fernandes. 2019. Raising the Parallel Abstraction Level for Streaming Analytics Applications. *IEEE Access* 7 (2019), 131944 – 131961.

[15] Microsoft. 2016. Parallel Patterns Library (PPL). https://docs.microsoft.com/en-us/cpp/parallel/concrt/parallel-patterns-library-ppl

[16] Claudia Misale, Maurizio Drocco, Guy Tremblay, Alberto R. Martinelli, and Marco Aldinucci. 2018. PiCo: High-performance data analytics pipelines in modern C++. *Future Generation Computer Systems* 87 (2018), 392 – 403.

[17] OpenMP ARB. 2018. OpenMP Application Program Interface Version 5.0. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[18] Ricardo Pieper, Dalvan Griebler, and Luiz G. Fernandes. 2019. Structured Stream Parallelism for Rust. In *23st Brazilian Symposium on Programming Languages (SBLP) (SBLP'19).* ACM, Salvador, Brazil, 8.

[19] Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and Data-Flow Compilation of OpenMP Streaming Programs. *ACM Trans. Archit. Code Optim.* 9, 4, Article 53 (Jan. 2013), 25 pages. https://doi.org/10.1145/2400682.2400712

[20] S. Prema, Rupesh Nasre, R. Jehadeesan, and B. K. Panigrahi. 2019. A study on popular auto-parallelization frameworks. *CCPE* 31, 17 (2019).

[21] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 179–196.

[22] Michael Voss, Rafael Asenjo, and James Reinders. 2019. *Pro TBB: C++ Parallel Programming with Threading Building Blocks* (1st ed.). Apress, USA.