An Embedded C++ Domain-Specific Language for Stream Parallelism

Dalvan Griebler [a,b], Marco Danelutto [b], Massimo Torquati [b], Luiz Gustavo Fernandes [a]

[a] *Faculty of Informatics, Computer Science Graduate Program, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre, Brazil*

[b] *Computer Science Department, University of Pisa, Italy*

Abstract. This paper proposes a new C++ embedded Domain-Specific Language (DSL) for expressing stream parallelism by using standard C++11 attributes annotations. The main goal is to introduce high-level parallel abstractions for developing stream based parallel programs as well as reducing sequential source code rewriting. We demonstrated that by using a small set of attributes it is possible to produce different parallel versions depending on the way the source code is annotated. The performances of the parallel code produced are comparable with those obtained by manual parallelization.

Keywords. Parallel Programming, Domain-Specific Language, Stream Parallelism, Multi-Core, Parallel Patterns, C++11 Attributes.

Introduction

Stream processing is used in many domains for solving problems such as image, data, and network processing. Almost all of these algorithms have the same pattern of behavior: they process continuous input and produce continuous output [1]. Moreover, stream-based applications require high throughput and low latency for real-time data analysis, quality of image processing, and efficient network traffic.

To achieve high-performance in streaming applications on multi-core architectures, developers have to deal with low-level parallel programming for taking advantage of the platform features. This can be a complex and time consuming task for software engineers that are not familiar with fine tuning and optimization of parallel code.

Our primary design goal is to allow C++ developers to express stream parallelism by using the standard syntax grammar of the host language. Secondly, we aim to enable minimal sequential code rewriting thus reducing the efforts needed to program the parallel application. Additionally, we aim to provide enough flexibility for annotating the C++ sequential code in different ways. The objective is to quickly obtain several different parallel versions to perform fast parallel code prototyping.

This research builds on DSL-POPP [2,3]. It is a Domain-Specific Language (DSL) designed for pattern-oriented parallel programming aimed at providing suitable building blocks (that are filled with sequential C code), and simplifying the implementation of Master/Slave and Pipeline patterns on shared memory systems. Its interface relies on the availability of a set of user mumble and intrusive annotations, parsed and processed by a dedicated compiler. This paper presents a new DSL to address limitations imposed by

the DSL-POPP's interface when expressing stream parallelism, parallel code generation, and different domain design goals.

Due to sustained contact with the REPARA project[1] team during the development of our current research, we have integrated the REPARA principles with those developed in DSL-POPP. We preserve the DSL-POPP principles of reducing significant sequential code rewriting by annotating the code. We have adopted the REPARA annotation mechanisms that allow us to be fully compliant with the standard C++11 language features (the host language syntax is embedded in the DSL) and reuse some techniques (like the GCC plugins to register custom attributes) to avoid rebuilding existing compiler tools. While DSL-POPP annotations are designed for novice parallel programming developers, REPARA's annotations are intermediate attribute annotations for parallelizing sequential code which are not directly exposed to end users [4], instead they are automatically inserted by appropriate tools. The main contributions of this paper are as follows:

- A domain-specific language for expressing stateless stream parallelism.
- An annotation-based programing interface that preserves the semantics of DSL-POPP's principles and adopts REPARA's C++ attribute style, avoiding significant code rewriting in sequential programs.

The paper is organized as follows. Section 1 highlights the contributions in comparison with the related work. Sec. 2 presents the proposed attributes for stream parallelism. Sec. 3 demonstrates the design implementation to introduce the proposed C++ attributes and parallel code transformation. Sec. 4 describes how to use our DSL in two applications as well as performance experiments. Finally, Sec. 5 discuss our paper contributions and future works.

1. Related Work

Standard parallel programming interfaces for High Performance Computing (HPC) such as OpenMP [5] and Cilk [6] were not designed for stream parallelism. Also, they are not provided by C++11 standard host language syntax, requiring the user to learn a new syntax. TBB [7] and FastFlow [8] are frameworks for structured parallel programming using algorithmic skeletons. They both preserve the host language syntax but require many changes in the source code to express parallelism. Finally, StreamIt[2] is a language specifically designed for stream parallelism [9]. To use it, C++ developers have to rewrite the source code, learn a new syntax and programming language. In contrast to the previously mentioned frameworks, our DSL only requires developers to find the stream parallelism regions and annotate them with the appropriate attributes. In addition, it also avoids rewriting the source code (like in FastFlow and TBB) and learning of a non-standard C++ syntax (like in Cilk and OpenMP).

Even though OpenMP and Cilk are designed to exploit loop-based parallelism, recent research ([10] and [11]) have tried to use the concept of stream parallelism in their interfaces. Authors of [10] proposed to add input and output pragma primitives for OpenMP tasks, which is like describing the data-flow graph in a sequential code and has now been adopted by the OpenMP-4.0. On the other hand, researchers proposed to in-

[1] http://parrot.arcos.inf.uc3m.es/wordpress/
[2] http://groups.csail.mit.edu/cag/streamit

clude pipeline parallelism using Cilk based implementation [11]. In their concept, "while loops" are used to implement pipelines assisted by language extension keywords. Even if both initiatives target stream parallelism, the drawback of these solutions remain source code rewriting, low-level parallel programming, and non-standard C++ programming. Building on these approaches is against our design goals. However, all related work offers potential environments to implement the parallel code for our domain-specific language. We have chosen FastFlow because it provides suitable building blocks for pipeline and farm skeletons, and it is efficient enough on multi-core architectures due to its lock-free implementation [12,13].

2. Attributes for Stream Parallelism

This section introduces the domain-specific language keywords to meet the design goals. We do not change the standard C++ attributes' syntax [14] for annotating codes. However, we determine how the attributes should be declared, ensuring the correct parallel code transformation. Table 1 describes the proposed keywords. All attributes are part of the stream parallelism namespace (named as `spar`), and they are placed inside of double brackets (`[[attr-list]]`). The first one on the list must be the attribute identifier (ID), which is one of the two first attributes from Table 1.

Attributes	Description
`ToStream()`	it annotates a loop or block for implementing a stream parallelism region
`Stage()`	it annotates a potential block stages that computes the stream inside of `ToStream`
`Input()`	used to indicate input streams for the ID attributes
`Output()`	used to indicate output streams for the ID attributes
`Replicate()`	it is an auxiliary attribute to indicate a stage replication

Table 1. The generalized C++ attributes for the domain-specific language.

As described in the introduction, almost all streaming applications have the same pattern of behavior when processing input and output streams. Therefore, we provide two keywords (`Input()` and `Output()`), which are optional arguments for the ID attributes. Basically, they are used to indicate the input and output stream that can be any variable's data type used as an argument. Consequently, a stream declaration in our context is viewed as singular or a list of the same or different data types. Also, we provide the auxiliary replicate attribute (`Replicate()`), which is associated with a stage block (`Stage()`). When necessary, this attribute will allow programmers to build non-linear computations for scaling the performance, where its argument is a constant value delimiting the number of workers for the stage.

```
1 auto b;
2 [[ToStream(Input(b))]] loop(...){
3   b = read(b);
4   [[Stage(Input(b),Output(b))]]
5   { b = proc(b); }
6   [[Stage(Input(b))]]
7   { b = write(b); }
8 }
```

Listing 1 Bounded/Unbounded streams in loops.

```
1 [[ToStream(Input(u))]]{
2   u = read(u);
3   [[Stage(Input(u),Output(u)),Replicate(N)]]
4   { u = proc(u); }
5   [[Stage(Input(u))]]
6   { u = write(u); }
7 }
```

Listing 2 Unbounded streams with replication.

A stream flow can be bound or unbound and can come from internal or external sources. The possibility of annotating loops is advantageous because the block declara-

tion can be reused and the end of the stream can be determined based on the loop statement. However, the end of the stream may never be known or difficult to handle when coming from external sources. In this case, it is possible to annotate the parallel region as a code block, where the end of the stream has to be programed inside the `ToStream` by using a break. Listing 1 and 2 illustrate the usage of all attributes for two scenarios.

Using the `ToStream()` is like saying: "*execute this block of code for each instance of the input*". Once it is declared, at least one stage block has to be annotated inside of its declaration. Grammar 1 describes the semantic rules for annotating the code. We do not allow any intermediary code between stage blocks. This limitation is needed to ensure correct code transformation and maintain the source code semantics. `ToStream()` structure declaration nesting is not allowed in the current implementation.

```
⟨tostream⟩ ::= '[[' 'ToStream' '('['','] [⟨input⟩] [','] [⟨output⟩] [',']')' ']]' [⟨loop-stmt⟩] '{' {⟨cmds⟩}* {⟨stage⟩}+ {⟨cmds⟩}* '}'
    ⟨stage⟩ ::= '[[' 'Stage' '('['','] [⟨input⟩] [','] [⟨output⟩] [',']')' [ ',' ⟨replicate⟩ ] ']]' [⟨loop-stmt⟩] '{' {⟨cmds⟩}* '}'
    ⟨input⟩ ::= 'Input' '(' ⟨args⟩ ')'
    ⟨output⟩ ::= 'Output' '(' ⟨args⟩ ')'
    ⟨replicate⟩ ::= 'Replicate' '(' ⟨consts⟩ ')'
```

Grammar 1: DSL's grammar using EBNF notation.

Once the stream parallel region is known, programmers have to identify the stream consuming activities and their input and output dependency for annotating one or more `Stage()` blocks. This activity can analyze network packets, apply a filter over images/videos, discover relevant information in a data set, among other kinds of computations. Depending on the application, it is possible to have a sequence of activities computing over the same stream and data dependent on the result of the previous activity. This is the same as the pipeline parallel pattern, which is composed of stages. Note that we are not able to manage statefull pipeline parallelism implementation. Therefore, when adding the replicate attribute to the stage block, our DSL assumes that the computations inside the stage can compute independently. It is up to the user to deal with the statefull pipeline activities, since this can not always be avoided [15].

3. Implementation

This section describes the design implementation of the proposed attributes using the GCC plugins technique and some parallel code transformation rules.

3.1. Introducing C++ Attributes with GCC plugin technique

C++ attributes originated from GNU C attributes (`__attribute__((<name>))`). Since C++11 up to the most recent version, a new way to annotate was included in the standard C++ language (`[[attr-list]]`) [14,16]. The syntax of the attributes was improved as well as the interface to support C++ features. A great advantage over the pragma-based annotation is the possibility of declaring almost everywhere in a program. However, each attribute implementation will determine where it can be annotated (*e.g.*, types, classes, code blocks, etc.). Also, the compiler is able to fully recognize the new standard mechanism, even if an attribute is not implemented by the language. This means that when adding customized attributes in the source code, the compiler attaches them in the Abstract Syntax Tree (AST) and simply ignores them in the code generation.

There are different techniques for taking advantage of this annotation mechanism such as adapting the source code of the GCC compiler, using an external parser, and creating a plugin. We propose using GCC plugins[3] because it avoids changing the internal structure of the compiler and gives us access to AST for parsing the arguments and source code tree. Therefore, it is possible to intercept the compiler during the program compilation for performing source-to-source parallel code generation, based on the the transformation rules that will be present in the next section.

3.2. Parallel Code Transformation

Our DSL provides high-level abstractions that are simple for the user, but have a complex system design for efficient code transformation. With only five keywords, we aim to provide several ways for annotating code to introduce stream parallelism without significant source code changes, which will be demonstrated in the experiment section (Section 4). To address this flexibility in code transformation, the system integrates algorithm skeleton principles such as the interacting and nesting mode [17]. These are mapped by the annotation of the computation activities (identify by using the `ToStream` and Stage ID attributes), spatial constrains (identify by using input and output attributes), temporal constrains (defined by the order of the declarations and their spatial constrains), and interaction (based on the users' stream dependency specification and using lock-free queues of FastFlow[4] runtime to communicate). Finally, we implement persistent nesting when the replicate attribute is declared in a stage activity.

We use a functional notation for representing our transformation rules, dividing in two phases: first we built our skeleton tree and after the FastFlow ones. *S* is a sequential code block and *R* is a replication function used to replicate *S* for a number of times. In FastFlow, we used their functional skeleton notation that includes Farm and Pipeline patterns [13]. Therefore, *S* can be a `ToStream` or `Stage` block type, where our syntax allows only add `Replicate` on the stages. It means that *R* only calls *S* for a stage type.

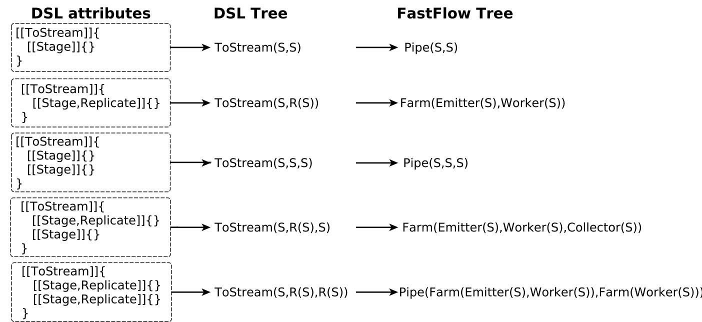


Figure 1. Transformation rules.

Figure 1 presents the main transformation rules. The data graph dependency is build when analysing the input and output so that the communication is implement. Note that in our stream parallelism region, the first block always will be no more than *S*, and

[3]https://gcc.gnu.org/onlinedocs/gccint/Plugins.html
[4]http://mc-fastflow.sourceforge.net/

everything that is left out the stage is part of the `ToStream` S. Consequently, it is also responsible for producing and coordinate the stream.

Figure 2 sketches the pseudo parallel code transformation to give an overview of the transformation sequence. The source code example is implemented in a simple pipeline, whereas in our interface the first stage is always the `ToStream` annotation, and the subsequent `Stage` annotations follow. Therefore, our example is a pipeline with three stages. To transform this source into a parallel FastFlow code, first we analyze all data dependencies declared as input and output attributes. Because AST access is possible, we can recognize the data types to create a single stream "struct" (label 1), which is the communication channel between stages. Secondly, we transformed the annotated blocks in stages by using the FastFlow building block interface (labels 2, 3, and 4). Lastly, we built the pipeline to run accordingly (label 5).

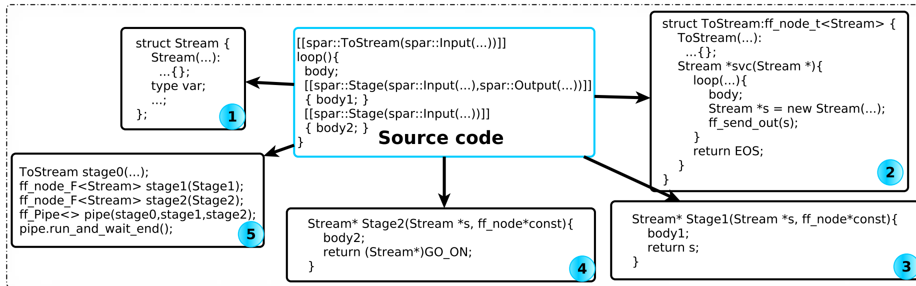


Figure 2. Example of the peseudo parallel code transformation using FastFlow runtime.

Declaring the `Replicate` for a `Stage` attribute, means that this stage will replicate as many times as specified using the FastFlow farm skeleton. Since the computation is structured, the plugin only needs to change the code for the last step (label 5). Although we did not illustrate all of the code in detail, this example also shows why it is important for us to use FastFlow for our runtime instead of other libraries in the code transformation. In addition, it avoids having to rebuild the synchronization and communication that FastFlow has already implemented on top of POSIX Threads.

4. Experiments

This section presents the DSL usage on an image processing and a numerical application. Whenever possible, we compare the performance of the transformed code with an equivalent code annotated with OpenMP pragmas, which is the reference standard for parallel programming on multi-core platforms. We reported the best execution time obtained for each tests. The performance value reported is the average value obtained over 40 runs. We used a dual-socket NUMA Intel multi-core Xeon E5-2695 Ivy Bridge micro-architecture running at 2.40GHz featuring 24 cores (12+12) each with 2-way Hyperthreading. Each core had 32KB L1 and 256KB L2 private, and 30MB L3 shared. The operating system was Linux 2.6.32 x86_64 shipped with CentOS 6.5. We compiled our tests using GNU GCC 4.9.2 with the –O3 flag. Since we are in the process of completing the implementation of the tools, results of the parallel code generation are inserted manually based on the rules of Section 3.2.

4.1. Sobel Application

This program applies the Sobel filter over bitmap images contained in a directory and writes the filtered images in a separate directory. Looking at Listing 3 and 4, the stream computation starts with the "while loop", where it is getting the file's information from the directory. Inside the loop, each file is preprocessed to get its name and extension. If it is a bitmap file, image's information is stored (e.g., image size, height, and width), as well as its buffers. Only after this has been completed, the program reads the image bytes from the disk to apply the filter. When this phase is concluded, the program writes the image to the disk. Finally (outside the loop), the program prints how many images were filter and how many files were ignored (non bitmap files).

```
1  using namespace spar;
2  //global declaration
3  int main(int argc, char *argv[]){
4    //open directory ...
5    DIR *dptr = opendir(...);
6    struct dirent *dfptr;
7    [[ToStream(Input(dfptr,dptr,argv),Output(
         tot_not,tot_img))]] while((dfptr =
         readdir(dptr)) != NULL){
8      //preprocessing
9      if (file_extension == "bmp"){
10       //Reads the image ...
11       tot_img++;
12       image = read(filename,height,width);
13       [[Stage(Input(image,height,width),Output(
        new_image)),Replicate(workers)]]{
14         //Applies the Sobel
15         new_image=sobel(image,height,width);
16       }
17       [[Stage(Input(new_image,height,width))]]{
18         //Writes the image ...
19         write(new_image,height,width);
20       }//end stage
21     }else{
22       tot_not++;
23     }
24   }//end of stream computing
25   //end processing
26   return 0;
27 }
```

Listing 3 Sobe version $S \to R(S) \to S$.

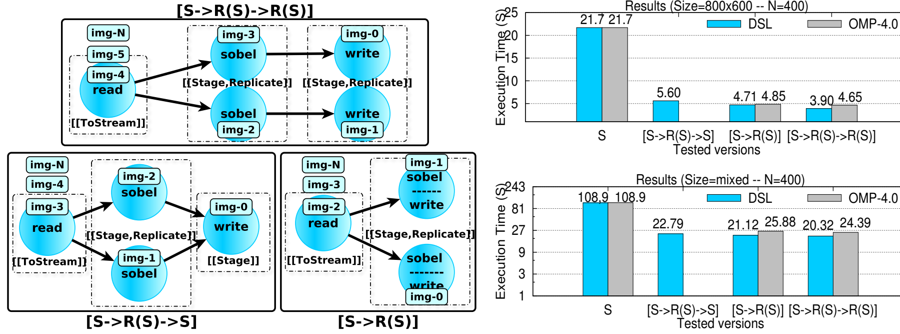
```
1  using namespace spar;
2  //global declaration
3  int main(int argc, char *argv[]){
4    //open directory ...
5    DIR *dptr = opendir(...);
6    struct dirent *dfptr;
7    [[ToStream(Input(dfptr,dptr,argv),Output(
         tot_not,tot_img))]] while((dfptr =
         readdir(dptr)) != NULL){
8      //preprocessing
9      if (file_extension == "bmp"){
10       //Reads the image ...
11       tot_img++;
12       image = read(filename,height,width);
13       [[Stage(Input(image,height,width)),
       Replicate(workers)]]{
14         //Applies the Sobel
15         new_image=sobel(image,height,width);
16         //Writes the image ...
17         write(new_image,height,width);
18       }//end stage
19     }else{
20       tot_not++;
21     }
22   }//end of stream computing
23   //end processing
24   return 0;
25 }
```

Listing 4 Sobel version $S \to R(S)$.

Using the proposed attributes, different ways for annotating this application are possible. We can observe that there are three potential stages (read, filter, and write). Therefore, the first strategy is to annotate the stage as presented in Listing 3, introducing the ToStream block in order to read images from the disk. After this, we can replicate the filtering phase because it is a costly computation and each image can be computed in parallel. The last stage just writes the images on the disk. Note that it is very important that input and output dependencies are properly declared[5]. This annotated code will be transformed as sketched in the worker model of Figure 3(a) (label $[S \to R(S) \to S]$).

It is up to the developer to choose the best code annotation schema for scaling performance. However, the expressiveness of our model allows developers to perform minimal changes on the sequential code with different annotation scheme and degree of parallelism. For example, in order to merge the two stages, it is sufficient to comment out lines 16 and 17 of Listing 3. Consequently, we have the code of Listing 4, producing the worker model sketched in Figure 3(a) (label $[S \to R(S)]$). Moreover, in respect to the first version of this application (Listing 3), we can produce a different worker model labelled as $S \to R(S) \to R(S)]$ (Figure 3(a)) by only adding the replicate attribute in the last stage.

[5] This is because our system assumes that the user's specification is correct. Wrong specifications may produce non-sequential code equivalence.



(a) The worker's model implementation. (b) Transformation's performance.

Figure 3. Implementation of the sobel application.

OpenMP-4.0 provides task parallelism pragma annotations for implementing computations that are not "for loop-based". However, non-linear pipelines such as the solution of the worker model of Figure 3(a) (label $[S \rightarrow R(S) \rightarrow S]$) can not be implemented (unless the original code is properly changed). This is because once a parallel region is annotated, all threads will go through the entire code block. Recently, OpenMP-4.0 added the new "depend" clauses, where the user can specify dependencies of the task group. This allows us to implement a comparable implementation of the $[S \rightarrow R(S) \rightarrow S]$ version. Yet, the worker model behavior is different. While in our solution each stage is always processing its input stream, in OpenMP, the end of each task region is a barrier for the group of tasks and all threads go through each phase. Furthermore, in both OpenMP implementations ($S \rightarrow R(S) \rightarrow R(S)$ and $S \rightarrow R(S)$) it was necessary to annotate two critical sections, where the program sums the total of the images read and ignored. On the other hand, using our solution this may be avoided.

We executed two experiments for this application. The graph on top of Figure 3(b) uses a balanced workload that has 400 images with the same size and 40 files that are not bitmap extensions. On the other hand, the graph at the bottom uses an unbalanced workload. As can be observed, our solution achieves the best executions times.

4.2. Prime Numbers Application

This application is the naïve algorithm for finding primer numbers. Basically, our implementation receives a number as input and checks by simply dividing it, and adding up every prime that is found. Listing 5 and 6 demonstrate the two ways for introducing stream parallelism for this application by using our DSL. In Listing 5 we annotate as stage the part of number checking using the auxiliary replicate attribute. A critical point in this program is the sum of the prime number already found (line 16). For this version we let the `ToStream` block perform this operation because it is sequentially executed. Another key point is to declare the output dependency so that when a number is checked it will be sent back.

In the second version we simply annotated the sum operation as a normal stage (Listing 6 on line 16). Consequently, we can accelerate our program because the stream flowed naturally and the `ToStream` does not pay with extra control of the multiple in-

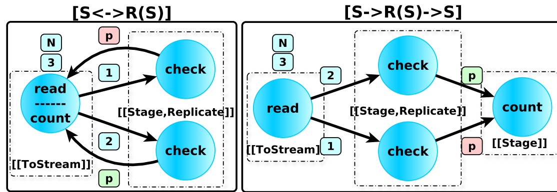
puts. This communication difference can be visualized in Figure 4(a). In OpenMP version, the only way for implementing this application was using the “parallel for” annotation, and specifying the following clauses: reduction(+:total), schedule(auto), shared(n), and num_threads(workers).

```
1  using namespace spar;
2  //global declarations ...
3  int prime_number(int n){
4    int total = 0;
5    [[ToStream(Input(total,n),Output(total)
       )]]
6    for (int i = 2; i <= n; i++ ){
7      int prime = 1;
8      [[Stage(Input(i,prime),Output(prime))
        ,Replicate(workers)]]
9      for (int j = 2; j < i; j++ ){
10       if ( i % j == 0 ){
11         prime = 0;
12         break;
13       }
14     }
15     total = total + prime;
16   }
17   return total;
18 }
```

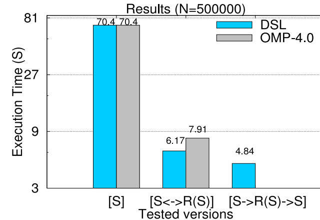
Listing 5 Prime numbers $S \leftrightarrow R(S)$.

```
1  using namespace spar;
2  //global declarations ...
3  int prime_number(int n){
4    int total = 0;
5    [[ToStream(Input(total,n),Output(total))]]
6    for (int i = 2; i <= n; i++ ){
7      int prime = 1;
8      [[Stage(Input(i,prime),Output(prime)),Replicate(
        workers)]]
9      for (int j = 2; j < i; j++ ){
10       if ( i % j == 0 ){
11         prime = 0;
12         break;
13       }
14     }
15     [[Stage(Input(total,prime),Output(total))]] { total
        = total + prime; }
16   }
17   return total;
18 }
```

Listing 6 Primer numbers $S \rightarrow R(S) \rightarrow S$.



(a) The worker's model implementation.



(b) Transformation's performance.

Figure 4. Implementation of the primer numbers application.

The results of the experiments are sketched in the graph of Figure 4(b). Our workload was to find the prime numbers in [1:500000]. As can be observed, we achieved better execution times than OpenMP in the $[S \leftrightarrow R(S)]$ version and even better results in the $[S \rightarrow R(S) \rightarrow S]$ transformation.

5. Conclusions

In this paper we presented a new embedded DSL for expressing simple stream-based parallelism. We used C++11 attributes as annotation mechanism and proposed GCC plugins technique for their implementation. We achieved all desired goals, demonstrating through simple examples that our standard annotation-based interface is flexible enough by only providing five attributes.

In the experiment section, the proposed DSL demonstrated good flexibility by supporting different ways for annotating programs to obtain different stream parallel implementations of the same application. The experiments shown efficient parallel code transformations by using FastFlow as runtime framework. The obtained performance are comparable with those obtained by parallelizing the code (when possible) using OpenMP. The DSL interface proposed increased code productivity (measured by SLOCCount[6])

[6]http://www.dwheeler.com/sloccount/

when compared directly with the FastFlow code: by 23.4% for the sobel application and by 27.5% for the prime number application. As a future work, we plan to implement automatic source-to-source transformation and to perform more experiments to evaluate our DSL with other stream-based applications.

Acknowledgements. This work has been partially supported by FAPERGS, CAPES, FACIN (Faculty of Informatics of PUCRS), and EU projects REPARA (No. 609666) and RePhrase (No. 644235).

References

[1] Henrique C. M. Andrade, Bugra Gedik, and Deepak S. Turaga. *Fundamentals of Stream Processing*. Cambridge University Press, New York, USA, 2014.

[2] Dalvan Griebler and Luiz G. Fernandes. Towards a Domain-Specific Language for Patterns-Oriented Parallel Programming. In *Programming Languages - 17th Brazilian Symposium - SBLP*, volume 8129 of *LNCS*, pages 105–119, Brasilia, Brazil, October 2013. Springer.

[3] Dalvan Griebler, Daniel Adornes, and Luiz G. Fernandes. Performance and Usability Evaluation of a Pattern-Oriented Parallel Programming Interface for Multi-Core Architectures. In *International Conference on Software Engineering & Knowledge Engineering*, pages 25–30, Canada, July 2014. SEKE.

[4] REPARA Project. D6.2: Dynamic Runtimes for Heterogeneous Platforms. Technical report, University of Pisa, Pisa, Italy, November 2014.

[5] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. MIT Press, London, UK, 2007.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Symposium on Principles and Practice of Parallel Programming*, volume 30 of *PPOPP '95*, pages 207–216, USA, August 1995. ACM.

[7] James Reinders. *Intel Threading Building Blocks*. O'Reilly, Sebastopol, CA, USA, 2007.

[8] Marco Aldinucci, Sonia Campa, Peter Kilpatrick, and Massimo Torquati. Structured Data Access Annotations for Massively Parallel Computations. In *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *LNCS*, pages 381–390, Greece, August 2012. Springer.

[9] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, Grenoble, France, April 2002. Springer-Verlag.

[10] Antoniu Pop and Albert Cohen. A Stream-Computing Extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 5–14, Heraklion, Greece, January 2011. ACM.

[11] I-Ting Angelina Lee, Charles E. Leiserson, Tao B. Schardl, Jim Sukha, and Zhunping Zhang. On-the-fly Pipeline Parallelism. In *ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 140–151, Portland, Oregon, USA, June 2013. ACM.

[12] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, Massimiliano Meneghin, and Massimo Torquati. An Efficient Unbounded Lock-Free Queue for Multi-core Systems. In *Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 662–673, Greece, August 2012. Springer.

[13] Marco Aldinucci, Sonia Campa, Fabio Tordini, Massimo Torquati, and Peter Kilpatrick. An Abstract Annotation Model for Skeletons. In *Formal Methods for Components and Objects*, volume 7542 of *Lecture Notes in Computer Science*, pages 257–276, Turin, Italy, October 2011. Springer Berlin Heidelberg.

[14] Jens Maurer and Michael Wong. Towards Support for Attributes in C++ (Revision 6). Technical report, The C++ Standards Committee, September 2008.

[15] William Thies and Saman Amarasinghe. An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 365–376, Austria, September 2010. ACM.

[16] ISO/IEC. Information Technology - Programming Languages - C++. Technical report, International Standard, Geneva, Switzerland, August 2011.

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