Graph Templates for Dataflow Programming

Alexandre C. Sena*, Eduardo S. Vaz‡,
Felipe M. G. França†, Leandro A. J. Marzulo* and Tiago A. O. Alves*
*Dep. de Informática e Ciência da Computação
Instituto de Matemática e Estatística, Universidade do Estado do Rio de Janeiro
Rio de Janeiro, Brasil. Email: {asena,leandro,tiagoaoa}@ime.uerj.br
†Programa de Engenharia de Sistemas e Computação, COPPE, Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brasil. Email: felipe@cos.ufrj.br
‡FlatLabs, Rio de Janeiro, Brasil. Email: eduardovaz@flatlabs.com.br

***Abstract*—Current works on parallel programming models are trending towards the dataflow paradigm, which naturally exploits parallelism in programs. The Sucuri Python Library provides basic features for creation and execution of dataflow graphs in parallel environments. However, there is still a gap between dataflow programming and traditional parallel programming. In this paper we aim at narrowing that gap by introducing a set of templates for Sucuri that represent some of the most important parallel programming patterns. Through these templates programmers can implement applications that use patterns such as fork/join, pipeline and wavefront just by instantiating and connecting sub-graph objects. Evaluation showed that the use of templates makes programming easier, while allowing a significant reduction in lines of code, compared to manually creating the dataflow graph.**

I. Introduction

Recent work on programming models and languages [1]–[11] show that the Dataflow model is an interesting alternative to ease parallelism exploitation, while achieving higher performance. In dataflow, tasks start running as soon as their input operands are ready, instead of following the program order. Therefore, parallelism naturally arises from the application description.

Typically, dataflow programming is done by instantiating blocks of codes and connecting them in a graph, according to data dependencies. This relieves the programmer from worrying about technicalities, such as thread/process creation, scheduling and synchronization, which are handled by the runtime system or library.

Sucuri [3] is a minimalistic dataflow library for the Python language. In Sucuri, the programmer instantiates *node objects*, which represent tasks that should run concurrently. Each node will have a function as an attribute, describing the work to be performed once the node execution is triggered, according to the dataflow rule. Then, the programmer must connect the nodes according to their data dependencies to form a dataflow graph representing the whole application. The graph is handed to a scheduler that will dispatch the execution of ready nodes to workers (processes) in a single multicore or a cluster of them. This means that if two functions have no dependencies between them, they may execute in parallel, depending on the availability of idle cores on the allocated machines.

Although one might argue that Python was not designed for high performance computing and is inherently not ideal for parallel programming, its immense popularity among the scientific community makes it worthwhile investigating ways of accelerating its execution. Nevertheless, there are several Python libraries for scientific computing that present C-like performance. In fact, most of those libraries are written in C and are just invoked by Python.

A successful parallel programming model should balance opacity and visibility of the underlying architecture, hiding complicated details and exposing features which exploit the full computational power of the underlying device. Moreover, it also determines how parallel tasks communicate and which types of synchronization among them are available [12].

Most of the parallel programming models provide some traditional parallel programming patterns vastly used by programmers, such as, pipeline, fork/join, reduction, among others. Patterns can be defined as valuable algorithmic structures that are commonly seen in efficient programs and in most parallel programming models [13].

Most dataflow machines or execution environments are not compatible with imperative languages [14]. Therefore, functional programming is often used for dataflow programming. However, the majority of parallel programmers (e.g. researchers) are not used to functional languages, instead they are used to coding through parallel programming patterns.

In this paper we present a set of templates for Sucuri that represent some of the most important parallel programming patterns. Through these templates programmers can implement applications that use patterns such as fork/join, pipeline and wavefront just by instantiating and connecting sub-graph objects. We believe that this set of templates can narrow the gap between dataflow programming and traditional parallel programming. Sucuri was modified to allow the creation of sub-graphs that can be interconnected to describe the whole application. This improves composability of the Sucuri programming model and allows the creation of new patterns by inheriting from the `DFsubGraph` class. Evaluation of case studies showed that overhead is negligible when using these templates, compared to manual dataflow graph creation, since templates only add the overhead of creating an object and calling a method that will run the same could that would be used in a manual build. Moreover, it was much simpler for the programmer and also there was reduction in lines of code.

The rest of this paper is organized as follows: Section II provides an overview of Sucuri and discusses its architec-

978-1-4673-8621-0/15 $31.00 © 2015 IEEE
DOI 10.1109/SBAC-PADW.2015.20

ture; Section III presents and discusses our proposed parallel programming patterns; Some related works are presented in Section IV. Section V presents some discussion about our library and future works.

II. SUCURI

Differently from the sequential Von Neumann execution model, the Dataflow execution model [15], where tasks start running as soon as their input operands are ready, presents a natural way of exploiting parallelism. In this model, programs are described as a dataflow graph, where each node represents a task (or instruction) and the edges that connect these nodes represent a direct data dependency between them. Therefore, a machine (or execution model) that observes the dataflow rule will be able to perform parallel execution of different tasks if there is no path between them in the graph and if there are enough computational resources available to execute them at the same time.

Sucuri [3] is a Python Library that provides dataflow programming in a high level and allows transparent execution on computer clusters. Sucuri is based on four main components: `DFGraph`, `Node`, `Scheduler` and `Worker`. The `DFGraph` is a container object used to hold the program to be executed, which is represented by a dataflow graph. A `Node` is a dataflow task and it is related to a function that can be implemented for any purpose. The Sucuri programmer implements custom functions that are passed as parameters at the instantiation of `Node` objects. After instantiating the nodes, the programmer can proceed to connecting them using the `add_edge()` method, which basically creates a new dependency in the dataflow graph. The `Scheduler` is responsible for routing messages between nodes and checking if a `Node` has received all its input operands. Ready nodes will be inserted in a ready queue for further execution. Each `Worker` is a process that will continuously fetch ready nodes. For each ready `Node`, the worker will call the function that was attributed to it and send the return values back to the `Scheduler`, which may trigger the execution of other nodes.

The library provides an abstraction layer that allows the same code to run in a multicore or in a cluster of multicores. If the programmer desires to be executed in a cluster it is only necessary to set the flag `mpi_enabled` as true and the program will run over multiple nodes with multiple threads in each node of the cluster. All communication intra-node is done via shared memory. When used in a cluster of computers, each of this components is replicated in each machine of the cluster. However, the current version of Sucuri implements a centralized `Scheduler`, meaning that the main `Scheduler`, placed at Cluster Node 0, will be responsible for all the scheduling, while remote schedulers will only forward messages to the main one.

Sucuri programming will be discussed in more detail in Section III, where parallel programming patterns for Sucuri are also introduced.

III. PARALLEL PATTERNS FOR DATAFLOW IN SUCURI

A programming model can be seen as an abstraction of the underlying system architecture. Moreover, it is not related to a specific machine, but to all machines with architecture that comply to the abstraction. A parallel programming model should carefully balance what the programmer should know about the underlying parallel machine, showing what is necessary to achieve the full potential of the architecture and hiding complex details that will not improve the performance [12]. More specifically, it should determine how to represent parallel tasks and how those parallel tasks communicate.

The most common parallel programming models or APIs for HPC are based on shared memory and message passing. The former is based on the assumption that all data is allocated in a common space that is accessible from every processor and the latter assumes that each processor has its own private data [16], [17]. The main implementations for those programming models are OpenMP and Pthreads for shared memory and MPI for message passing. Besides, new heterogeneous parallel programming models were produced for the new processor architectures that are composed of multi-core CPUs and many-core GPUs [17].

Most of the parallel programming models available have some traditional parallel programming patterns that are used to make coding easier, such as, pipeline, fork/join, reduction, among others. Patterns can be defined as valuable algorithmic structures that are commonly seen in efficient programs and in most parallel programming models [13]. Most of the parallel programmers are used to coding using such patterns.

Dataflow machines or execution environments are usually not compatible with imperative languages [14]. The problem is that imperative languages rely on global state (side effects), which create implicit data dependencies that would not appear in a dataflow graph. Therefore, functional programming is often used for dataflow programming. As the majority of programmers, specially the researchers, are not used to functional languages, some dataflow environment have adopted other ways of programming. For instance, Sucuri utilizes a graph model to facilitate parallel programming [3].

Despite the graph model being very simple for expressing parallelism, scientists are used to coding through structured patterns (described above). Therefore, the addition of templates that represent the most common parallel patterns in the dataflow graph structure would clearly help the scientists build parallel programs for dataflow execution models, such as Sucuri. Actually, through these templates, any parallel programmer would be able to program using Sucuri and harness the parallelism available in computer clusters and multi/many-core machines.

In this Section we discuss the changes required in Sucuri to support parallel programming patterns and present the implementation details of three patterns: Fork/Join, Wavefront and Linear Pipeline.

A. Sub-graphs in Sucuri

Patterns in Sucuri are special graphs that should be easier to assemble. Moreover, there should be a way of composing larger graphs formed by different patterns that interconnect. Therefore, in this work we introduce the `DFSubGraph` abstract class that inherits from the `DFGraph` class. It implements the following features:

- It allows the addition of sub-graphs, instead of just nodes. Sub-graphs will be flattened into the main graph, enabling composability.
- It keeps a list of input and output nodes that will be extremely useful if one wants to connect two sub-graphs.
- It implements the `add_edge()` method that allows the connections of a sub-graph to another sub-graph or node. When a sub-graph is the source of a connection, all its output nodes will be sources. When a sub-graph is the destination of a connection, all its input nodes will be destinations. At the original Sucuri, the `add_edge()` method only existed in nodes.
- Its constructor calls the `buildSubGraph()` method, not implemented by the `DFSubGraph` class. Patterns should inherit form the `DFSubGraph` class and implement the `buildSubGraph()` method to construct the graph automaticaly.

When using templates, the programmer needs the main graph to allow the addition of sub-graphs. This main graph should not call the `buildSubGraph()` in its constructor, since it is not building a pattern. Therefore, a `ContainerGraph` class was created for this purpose and must be used instead of `DFGraph`.

B. Fork/Join

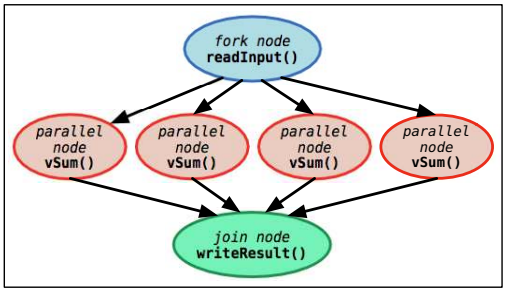
The Fork/Join template was implemented in Sucuri through the `ForkJoinGraph` class that inherits from `DFSubGraph` (presented in Section III-A). The `ForkJoinGraph` constructor receives: (*i*) three functions (fork, parallel and join); (*ii*) an integer, representing the number of tasks to be created for the parallel section; and (*iii*) and the number of inputs of the graph (optional, with 0 as default). It generates a *fork node*, a set of *parallel nodes* and a *join node*, as observed in Figure 1(a). The fork/join syntax can be observed in Figure 1(c) (line 7).

The `ForkJoinGraph` supports connections with external nodes, as explained in Section III-A. The *fork* and *join* nodes are, respectively, the only input and output nodes.

Figure 1 shows how a Fork/Join Graph can be implemented in Sucuri. Figure 1(a) depicts the visual representation of the graph, while figures 1(c) and 1(b) present the Sucuri code with and without templates, respectively. It is important to notice:

1) The import of the Sucuri template library (line 2 in Figure 1(c)). This is needed when using any of the templates.
2) The use of a `ContainerGraph` instead of a `DFGraph` (line 3 in Figure 1(b) vs. line 4 in Figure 1(c)).
3) The entire fork/join graph is built just by creating an object (lines 4-12 in Figure 1(b) vs. line 5 in Figure 1(c)).
4) The fork/join graph needs to be added to the container graph (line 6 in Figure 1(c)) .

The Sucuri fork/join template was used in a case study to implement a numerical integration and compared to a version that produces the same dataflow graph without templates. The application skeleton is the same shown in Figure 1. The execution time of the template version is the same of the manual version, which suggests that there are no significant overheads for template creation.



(a) Fork/Join dataflow graph

```
01 from sucuri import *
02 nprocs = 4
03 graph = DFGraph()
04 forkNode = MultiOutputNode(readInput,0)
05 graph.add(forkNode)
06 joinNode = Node(writeResult, nprocs)
07 graph.add(joinNode)
08 for i in xrange(nprocs):
09         pNode = Node(vSum,1)
10         graph.add(pNode)
11         forkNode.add_edge(pNode,0,i)
12         pNode.add_edge(joinNode,i)
13 sched = Scheduler(graph, nprocs, mpi_enabled = False)
14 sched.start()
```

(b) Sucuri Fork/Join code without template

```
01 from sucuri import *
02 from sucuri.templates import *
03 nprocs = 4
04 graph = ContainerGraph()
05 fj = ForkJoinGraph(inputRead,vSum,nprocs,writeResult)
06 graph.add(fj)
07 sched = Scheduler(graph, nprocs, mpi_enabled = False)
08 sched.start()
```

(c) Sucuri Fork/Join code with template

Fig. 1. Fork/Join in Sucuri

C. Wavefront

Wavefront [18] is a commonly used technique to address synchronization issues on applications that process data in arrays where there is a dependence pattern between elements. A classic example is the longest-common subsequence (LCS) algorithm, basis of some DNA analysis algorithms, where an element can be computed when its upper and left neighbors are finished. This pattern was implemented in Sucuri through the `ClassicWavefrontGraph` class that inherits from `DFSubGraph`. The `ClassicWavefrontGraph` constructor receives: (*i*) one function; (*ii*) a tuple with 2 elements, representing the wavefront matrix dimensions (width and height); and (*iii*) and the number of inputs of the graph (optional, with 0 as default). The wavefront syntax can be observed in Figure 2(c) (line 8).

The `ClassicWavefrontGraph` supports connections with external nodes, as explained in Section III-A. The upper-

left node $(0, 0)$ is the only input node and the lower-right node $(h - 1, w - 1)$ is the only output node.

Figure 2 shows how LCS can be implemented in Sucuri using a Wavefront Graph. Figure 2(a) depicts the visual representation of the graph, while figures 2(c) and 2(b) present the Sucuri code with and without templates, respectively. It is important to notice:

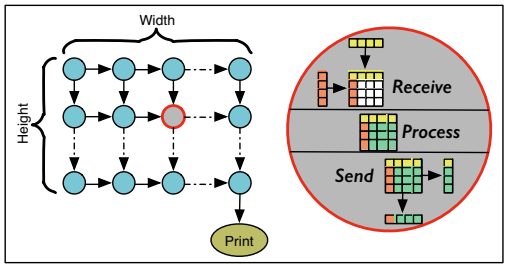
1) The import of the Sucuri template library (line 2 in Figure 2(c)). This is needed when using any of the templates.
2) The use of a `ContainerGraph` instead of a `DFGraph` (line 11 in Figure 2(b) vs. line 5 in Figure 2(c)).
3) The entire wavefront graph is built just by creating an object (lines 12-26 in Figure 2(b) vs. line 6 in Figure 2(c)).
4) The wavefront graph needs to be added to the container graph (line 7 in Figure 2(c)).
5) In a classic wavefront pattern the upper-left node has no inputs (unless it is connected with an external node), the other nodes in the first row and column have only one input and the rest of the nodes have two inputs. Therefore, implementing a wavefront graph without a template requires a function to determine the number of inputs, based on the node coordinates (lines 3-8 in Figure 2(b)).
6) The wavefront graph is connected to a printer node in a much easier way in the template version (line 30 in Figure 2(b) vs. line 11 in Figure 2(c)).
7) In this application, each node processes a portion of a score-matrix that will be used to determine the size of a longest common subsequence. This sort of matrix partition is the key to select task granularities that can avoid overheads and provide good scalability (computing one element per node would incur in high overhead).

The Sucuri wavefront template was used in a case study to implement the LCS algorithm and compared to a version that produces the same dataflow graph without templates. The application skeleton is the same shown in Figure 2. The execution time of the template version is the same of the manual version, which suggests that there are no significant overheads for template creation.

D. Linear Pipeline

The Pipeline pattern is very useful for hiding I/O latency and for processing streams of data, such as videos and database queries. Therefore, in this pattern we assume that there should be a `Source` node streaming information into the pipeline. The `Source` node receives an *iterable* python object (for instance, a list or a file descriptor) at instantiation. Its execution will be fired only once and will typically last until the contents of the iterable object are exhausted. By default the `Source` node will loop over the iterable object sending the values retrieved in each iteration to the nodes that depend on that `Source`.

Our pipeline template assumes asynchronous stages and a final ordered stage, which could be used for writing results. The data produced by the `Source` traverses the pipeline and



(a) Wavefront dataflow graph

```
01 from sucuri import *
02 #wavefront nodes have different number of inputs
03 def ninputs(i,j):
04     if i==0 and j==0:
05         return 0
06     if i==0 or j ==0:
07         return 1
08     return 2
09 sA, sB, sizeA, sizeB, width, height = readSeq()
10 nprocs = 4
11 graph = DFGraph()
12 #create wavefront nodes
13 G = [[MultiOutputNode(functionWrapper({"tid":(i,j),
       "ntasks":(width,height)}, LCS),ninputs(i,j))
       for j in xrange(width)] for i in xrange(height)]
14 #add nodes to graph
15 for i in xrange(height):
16     for j in xrange(width):
17         graph.add(G[i][j])
18 #connect nodes
19 for i in xrange(height):
20     for j in xrange(width):
21         if i > 0:
22               #create edge from  upper neighbor
23               G[i-1][j].add_edge(G[i][j],0,0)
24         if j > 0:
25               #create edge from left neighor
26               G[i][j-1].add_edge(G[i][j],int(i>0),1)
27 printer = Node(printLCS, 1)
28 graph.add(printer)
29 #connect last node of the wavefront to the printer
30 G[height-1][width-1].add_edge(printer,0)
31 sched = Scheduler(graph, nprocs, mpi_enabled = False)
32 sched.start()
```

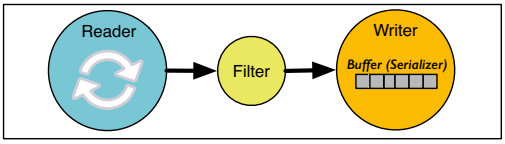
(b) Sucuri wavefront code without template

```
01 from sucuri import
02 from sucuri.templates import *
03 sA, sB, sizeA, sizeB, width, height = readSeq()
04 nprocs = 4
05 graph = ContainerGraph()
06 G = ClassicWavefrontGraph(LCS,(width,height))
07 graph.add(G)
08 printer = Node(printLCS, 1)
09 graph.add(printer)
10 #connect last node of the wavefront to the printer
11 G.add_edge(printer,0)
12 sched = Scheduler(graph, nprocs, mpi_enabled = False)
13 sched.start()
```

(c) Sucuri Wavefront code with template

Fig. 2. Wavefront in Sucuri - Basic implementation of an LCS algorithm

may arrive out-of-order in the last node due to load imbalance. Therefore, the last node must be a `Serializer`, a special node that is responsible for processing data in the order that it was generated by the `Source`. The data produced by the `Source` node is encapsulated in a `TaggedValue` object, which contains a `tag` attribute, indicating its position in the ordered data set. All nodes in the pipeline will forward the original `tag`. The `Serializer` node will store received data in a buffer sorted according to the tag. If the tag of the last received piece of data corresponds to the next data to processed, the `Serializer` node proceeds to pop data from the sorted buffer and processes it until there is a gap in the ordering, i.e. the chunk of data that is the next to be processed has not arrived yet. If the data received by the `Serializer` is out of order, the node just stores it in the sorted buffer and waits for more data. The `pin()` method is used to pin the node to a certain worker, which will make it only be executed by that worker.



(a) Linear Pipeline dataflow graph

```
01 from sucuri import
02 nprocs = 4
03 fp = open("input.txt","r")
04 graph = DFGraph()
05 s=Source(fp)
06 f=FilterTagged(filterFunc,1)
07 w=Serializer(pValue,1)
08 w.pin(0)
09 graph.add(s)
10 graph.add(f)
11 graph.add(w)
12 s.add_edge(f,0)
13 f.add_edge(w,0)
14 sched = Scheduler(graph, nprocs, mpi_enabled = False)
15 sched.start()
```

(b) Sucuri Linear Pipeline code without template

```
01 from sucuri import *
02 from sucuri.templates import
03 nprocs = 4
04 fp = open("input.txt","r")
05 graph = ContainerGraph()
06 pipe = LinearPipelineGraph((filterFunc, pValue), fp)
07 graph.add(pipe)
08 sched = Scheduler(graph, nprocs, mpi_enabled = False)
09 sched.start()
```

(c) Sucuri Linear Pipeline code with template

Fig. 3. Linear Pipeline in Sucuri

A `LinearPipelineGraph` class that inherits from the `DFSubGraph` class (presented in Section III-A) was implemented in Sucuri to describe the pipeline pattern. Its constructor receives a list of functions to be executed by each stage and an optional iterable object (`dataSource`) that will be passed to a `Source` node. If `dataSource` is not passed as parameter, the programmer should connect an external `Source` node (or another pipeline) to that pipeline. The pipeline syntax can be observed in Figure 3(c) (line 6).

The `LinearPipelineGraph` supports connections with external nodes, as explained in Section III-A. If the pipeline has a `dataSource` it will have no input nodes, otherwise its first stage will be used as input node. The last stage is the only output node.

Figure 3 shows how a pipeline can be implemented in Sucuri. Figure 3(a) depicts the visual representation of the graph, while figures 3(c) and 3(b) present the Sucuri code with and without templates, respectively. It is important to notice:

1) The import of the Sucuri template library (line 2 in Figure 3(c)). This is needed when using any of the templates.
2) The use of a `ContainerGraph` instead of a `DFGraph` (line 4 in Figure 3(b) vs. line 5 in Figure 3(c)).
3) The entire pipeline graph is built just by creating an object (lines 5-13 in Figure 3(b) vs. line 6 in Figure 3(c)).
4) The pipeline graph needs to be added to the container graph (line 7 in Figure 3(c)).

The Sucuri pipeline template was used in a case study to implement a Text Filter application and compared to a version that produces the same dataflow graph without templates. The application skeleton is the same shown in Figure 3 and there was no overhead for using a template.

IV. Related Work

Swift is a scripting language that can execute applications on distributed environments. It provides a simple set of language constructs to specify how applications are glued together and executed in parallel at large scale [10], [11]. Swift employs a Dataflow execution model and is based on functional languages. In other words, Swift is a language for composing Workflows, while the graph structure of Sucuri is more suitable for programming parallel applications.

Ruffus [19] is a python library for the execution of parallel jobs on large pieces of data in pipelined applications. The communication between pipeline stages is done through I/O operations in files specified in the configuration of the application. Ruffus is specific for pipeline applications and is not designed to work in clusters.

Auto-Pipe [20] is a project comprised by a toolchain and a language designed for the implementation of pipelines in heterogeneous architectures, i.e. architectures with multicores, FPGAs, GPUs etc. The language employed in Auto-Pipe (X-language) provides a simple syntax to connect the blocks corresponding to the pipeline stages and specify which of the components of the heterogeneous system will execute which state.

StreamIt [21] is another programming language intended for streaming programming, it allows the implementation of large stream based applications. The graph structure of Sucuri is more generic, allowing the implementation of different patterns, including pipelines. Moreover, the main parallel patterns are available through templates to be more accessible to the scientists and researchers.

Intel Threading Building Block (TBB) [22] is a C++ library that introduces features very similar to the ones found in

Sucuri. The programmer may describe flow graphs using the `tbb::flow` namespace. Similar to Sucuri, the programmer will instantiate the nodes and then connect them, using the `make_edge` function. The shortcomings of TBB are the complex C++ syntax and the lack of dataflow templates (there are TBB templates for traditional parallel programming), which may be too difficult for users in fields not directly related to computer science/engineering, and mainly the fact that it only works in multicore machines, not clusters.

V. Conclusions and Future Work

The dataflow execution model, employed in Sucuri, presents a natural way of exploiting parallelism. Sucuri adopts a graph model to facilitate the programming, instead of an functional language that is commonly used in dataflow.

Although using the graph model to express parallelism is very simple, most scientists are used to employing structured parallel patterns that are available in most parallel languages such as fork/join, pipeline, wavefront, among others. Thus, this work presented templates for three important parallel patterns, allowing scientists to adopt Sucuri to program a parallel application. The benefits of the templates are twofold: facilitate the utilization of the parallel patterns implemented and show to the scientists the simplicity of the graph model encouraging them to implement more complex structures.

The parallel templates implemented in this work presented a simple way of allowing scientists to program using Sucuri. As a future work, new templates, such as parallel for, non-linear pipelines, and other wavefront patterns will be implemented, as well as a set of reduction functions to be used with the fork/join and parallel for patterns. Moreover templates will be enhanced to accept sub-graphs, improving composability. We also want to provide conditional execution and predication, like *Steer* and *Select* instructions at WaveScalar [14]. Moreover, we plan on extending Sucuri to allow dynamic graph creation, where new nodes could be dynamically instantiated by functions that run on other nodes.

Since the `Node` class can be extended for specific purposes, we also intend to implement a library containing all kinds of nodes, making Sucuri even easier to use.

Acknowledgments

The authors would like to thank CAPES, CNPq and FAPERJ for the financial support given to the authors of this work.

References

[1] T. A. Alves, L. A. Marzulo, F. M. Franca, and V. S. Costa, "Trebuchet: exploring TLP with dataflow virtualisation," *International Journal of High Performance Systems Architecture*, vol. 3, no. 2/3, p. 137, 2011.

[2] L. A. Marzulo, T. A. Alves, F. M. França, and V. S. Costa, "Couillard: Parallel programming via coarse-grained data-flow compilation," *Parallel Computing*, vol. 40, no. 10, pp. 661 – 680, 2014.

[3] T. Alves, B. Goldstein, F. Franca, and L. Marzulo, "A minimalistic dataflow programming library for python," in *Computer Architecture and High Performance Computing Workshop (SBAC-PADW), 2014 International Symposium on*, Oct 2014, pp. 96–101.

[4] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, pp. 173–193, 2011-03-01 2011.

[5] S. Balakrishnan and G. Sohi, "Program Demultiplexing: Data-flow based Speculative Parallelization of Methods in Sequential Programs," in *33rd International Symposium on Computer Architecture (ISCA'06)*. Washington, DC, USA: IEEE, 2006, pp. 302–313.

[6] G. Gupta and G. S. Sohi, "Dataflow execution of sequential imperative programs on multicore architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: ACM, 2011, pp. 59–70.

[7] "Tbb flowgraph," http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm, accessed on August 8, 2014.

[8] G. Bosilca, A. Bouteiller, A. Danalis, T. Hrault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing." *Parallel Computing*, vol. 38, no. 1-2, pp. 37–51, 2012.

[9] R. Giorgi, R. M., F. Bodin, A. Cohen, P. Evripidou, P. Faraboschi, B. Fechner, G. R., A. Garbade, R. Gayatri, S. Girbal, D. Goodman, B. Khan, S. Koliaï, J. Landwehr, N. Minh, F. Li, M. Lujàn, A. Mendelson, L. Morin, N. Navarro, T. Patejko, A. Pop, P. Trancoso, T. Ungerer, I. Watson, S. Weis, S. Zuckerman, and M. Valero, "TERAFLUX: Harnessing dataflow in next generation teradevices," *Microprocessors and Microsystems*, pp. –, 2014.

[10] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, no. 9, pp. 633 – 652, 2011, emerging Programming Paradigms for Large-Scale Scientific Computing.

[11] J. Wozniak, T. Armstrong, M. Wilde, D. Katz, E. Lusk, and I. Foster, "Swift/t: Large-scale application composition via distributed-memory dataflow processing," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, May 2013, pp. 95–102.

[12] V. Dimakopoulos, "Parallel programming models," in *Smart Multicore Embedded Systems*, M. Torquati, K. Bertels, S. Karlsson, and F. Pacull, Eds. Springer New York, 2014, pp. 3–20.

[13] M. D. McCool, A. D. Robison, and J. Reinders, *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[14] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, "WaveScalar," in *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*. IEEE Comput. Soc, 2003, pp. 291–302.

[15] J. B. Dennis and D. P. Misunas, "A preliminary architecture for a basic data-flow processor," *SIGARCH Comput. Archit. News*, vol. 3, no. 4, pp. 126–132, 1974.

[16] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, Eds., *Sourcebook of Parallel Computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[17] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 8, pp. 1369–1386, Aug 2012.

[18] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling, and K. Tan, "Generating parallel programs from the wavefront design pattern," in *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, ser. IPDPS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 165–.

[19] L. Goodstadt, "Ruffus: A lightweight python library for computational pipelines," *Bioinformatics*, 2010.

[20] M. A. Franklin, E. J. Tyson, J. Buckley, P. r. Crowley, and Maschmeyer, "Auto-Pipe and the X Language : A Pipeline Design Tool and Description Language Tool and Description Language ,in Proc. of Intl Parallel and Distributed Auto-Pipe and the X Language : A Pipe line Design Tool and Description Language," no. April, 2006.

[21] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 179–196.

[22] J. Reinders, *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.