Task Scheduling in Sucuri Dataflow Library

Rafael J. N. Silva*, Brunno Goldstein†, Leandro Santiago†,
Alexandre C. Sena*, Leandro A. J. Marzulo*, Tiago A. O. Alves*, Felipe M. G. França†
*Instituto de Matemática e Estatística
Universidade do Estado do Rio de Janeiro (UERJ), Rio de Janeiro, Brazil
Email: {rsilva,asena,leandro,tiago}@ime.uerj.br
†Programa de Engenharia de Sistemas e Computação - COPPE
Universidade Federal do Rio de Janeiro (UFRJ), Rio de Janeiro, Brazil
Email: {lsantiago, bfgoldstein, felipe}@cos.ufrj.br

***Abstract*—Sucuri is a minimalistic Python library that provides dataflow programming through a reasonably simple syntax. It allows transparent execution on computer clusters and natural exploitation of parallelism. In Sucuri, programmers instantiate a dataflow graph, where each node is assigned to a function and edges represent data dependencies between nodes. The original implementation of Sucuri adopts a centralized scheduler, which incurs high communication overheads, specially in clusters with a large number of machines. In this paper we modify Sucuri so that each machine in a cluster will have its own scheduler. Before execution, the dataflow graph is partitioned, so that nodes can be distributed among the machines of the cluster. In runtime, idle workers will grab tasks from a ready queue in their local scheduler. Experimental results confirm that the solution can reduce communication overheads, improving performance in larger clusters.**

I. Introduction

Dataflow computing became an attractive alternative to mitigate the burden of developers in dealing with difficulties imposed by the parallel programming. In this model applications are represented as dataflow graphs, where nodes are the program's instructions (or tasks) and edges describe data dependencies between such instructions. An instruction is triggered to run when all its input data is available, according to their dependencies. Independent nodes are allowed to execute concurrently, meaning that parallelism naturally arises by just following the dataflow model. There are several projects that employ dataflow to exploit parallelism, in the form of runtime environments and libraries ([1], [2], [3], [4], [5], [6], [7], [8]) that allow the dataflow paradigm be supported in Von Neumann machines or in the form of full dataflow machines ([9], [10]) or FPGA accelerators [11].

Dataflow programming solutions for Von Neumann machines usually require that users define nodes and their connections, leaving all the complexity of parallelization, such as synchronization, control and scheduling, to be handled by the runtime environment. However, there is a lack of dataflow tools to be used with the most popular high-level languages employed by scientific community, such as Python.

Sucuri [12] is a minimalistic dataflow library for Python, that provides a simple and straightforward syntax. Intuitively, programmers must describe their application in the form of a dataflow graph, by just creating *node objects* associated to functions and connecting the nodes according to the data dependencies. The graph is passed to the Sucuri scheduler that takes care of the execution in a transparent way, either in multicores or clusters.

Current implementation of Sucuri has a centralized dynamic scheduler that employs a first-come first-served approach to distribute tasks among workers in the system. When the library is used in larger clusters, communication overhead can considerably degrade the overall performance, making Sucuri unsuited for such environments. To solve this problem, this paper proposes a decentralized scheduler for Sucuri, meaning that each Machine will have its own scheduler and workers can request tasks locally, without the need of using the network. Before execution, the dataflow graph is partitioned, using an algorithm based on list scheduling, so that nodes can be distributed among the machines of the cluster.

Experimental results show that our solution offered a better distribution of work, minimizing communication costs and reducing bottlenecks. We evaluated applications with different parallelism patterns: fork-join, wavefront and pipeline. All classes of applications obtained good speedups suggesting that Sucuri can also be a promising parallel programming tool for larger clusters.

The rest of this paper is organized as follows: Section II explains the Sucuri library; Section III presents our proposal for decentralizing the dynamic scheduler and our graph partition approach; Section IV presents the experiments and discusses the results; Section V discusses some related work; and Section VI concludes and indicate possible future contributions.

II. The Sucuri Library

Sucuri [12] is a high level parallel programming library for Python language that aims at enabling dataflow execution on general purpose processors. To create an dataflow application in Sucuri, user need to: (*i*) instantiate `Node` objects associated with functions defined by the programmer; (*ii*) connect them, according to their dependencies, using the `add_edge` method; (*iii*) add the nodes to a `DFGraph` object; (*iv*) instantiate a `Scheduler` object that will receive that graph; (*v*) run the scheduler, by calling the `start` method.

Figure 1 provides an example on how to program in Sucuri. Figure 1(a) shows the sequential Python code of an appli-

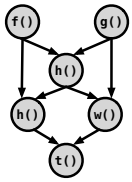
978-1-5090-4844-1/16 $31.00 © 2016 IEEE
DOI 10.1109/SBAC-PADW.2016.15

cation, while Figure 1(b) shows the corresponding dataflow graph. Finally, Figure 1(c) shows the Sucuri code, where:

- Line 1 imports the Sucuri Library;
- Line 2 instantiates an empty dataflow graph;
- Line 3 instantiates a scheduler that will create a certain number of workers (variable `nWorkers`) and run the graph;
- Lines 4 to 9 create the nodes of the graph and relates each of them to a function. The second parameter in the node creation is the number of inputs needed by the node;
- Lines 10 to 15 add the nodes to the graph;
- Lines 16 to 23 connect the nodes, using the `add_edge` method, where the first parameter is the destination node and second parameter is the input port in the destination node;
- Line 24 starts the scheduler (runs the application).

```
01 | a = f()
02 | b = g()
03 | c = h(a,b)
04 | d = h(a,c)
05 | e = w(c,b)
06 | x = t(d,e)
```

(a) Sequential Code



(b) Dataflow Graph

```
01 | from Sucuri import *
02 | graph = DFGraph()
03 | sched = Scheduler(graph, nWorkers)
04 | a = Node(f, 0)
05 | b = Node(g, 0)
06 | c = Node(h, 2)
07 | d = Node(h, 2)
08 | e = Node(w, 2)
09 | x = Node(t, 2)
10 | graph.add(a)
11 | graph.add(b)
12 | graph.add(c)
13 | graph.add(d)
14 | graph.add(e)
15 | graph.add(x)
16 | a.add_edge(c,0)
17 | b.add_edge(c,1)
18 | a.add_edge(d,0)
19 | c.add_edge(d,1)
20 | c.add_edge(e,0)
21 | b.add_edge(e,1)
22 | d.add_edge(x,0)
23 | e.add_edge(x,1)
24 | sched.start()
```

(c) Sucuri Code

Fig. 1. Programming in Sucuri.

Figure 2 details the Sucuri architecture proposed in this work. To avoid including very similar figures (for the original and modified Sucuri), we will use this Figures to explain how the original implementation works. Sucuri is composed of three main components: *Workers*, *Scheduler* and the dataflow graph of the application. *Workers* are responsible for processing functions of each node in the graph. Also, they receive and deliver input and output operands to the *Scheduler*. The *Scheduler* receives the resulting operands that come from *Workers* and nodes that depends on them to execute. If a match occurs, a new task is created and placed at the end of the *Ready Queue* to be consumed by an idle *Worker*.

The library provides an abstraction layer for parallel computing allowing the same code to run either on multicores or a cluster of multicores. In a cluster of multicores, Sucuri uses *MPI* to exchange messages between machines. Therefore, users are required to describe all dependencies in the dataflow graph. When executing in multicores, users could rely on side effects, i.e., use shared memory for implicit exchange of data between nodes. However this is not recommended, since the idea is to have the same code for clusters and multicores.

The major problem of the original Sucuri implementation is that it adopts a centralized *Scheduler*, meaning that all components are replicated on each machine of the cluster but only the *master* machine holds a fully functional *Scheduler*. Schedulers on *slave* machines are responsible only for forwarding messages (with tasks and operands) between remote workers and the *master*. In Figure 2 all machines have their own fully functional scheduler, as described further in this Section.

III. Decentralized Scheduler for Sucuri

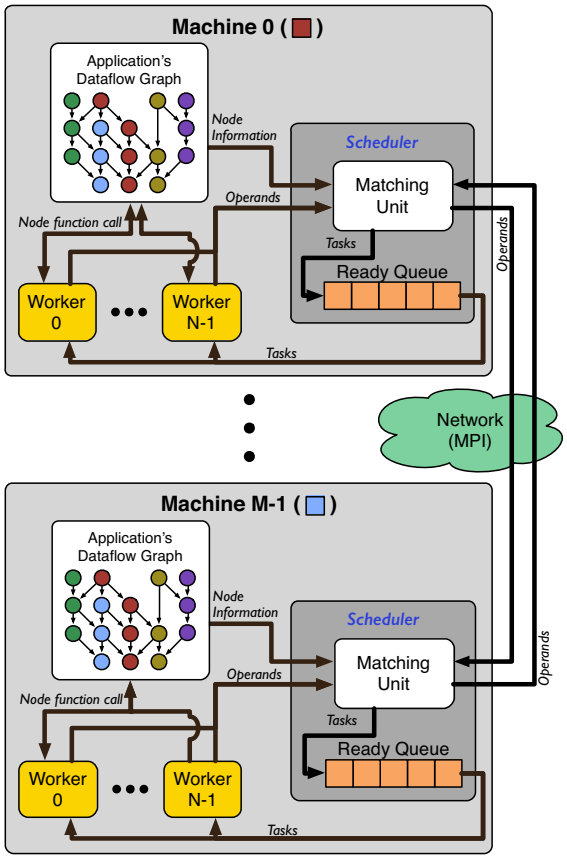
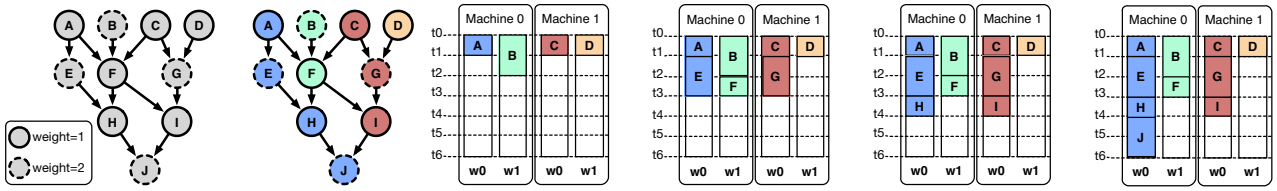


Fig. 2. Sucuri Architecture (with decentralized scheduler).

Sucuri has shown excellent results on a large set of parallel applications [12]. However, due to the overhead imposed by a centralized scheduler, scalability issues were observed when executing on larger clusters [13]. In order to solve this problem, we modified Sucuri to include a fully functional



(a) Application graph (b) Allocation (c) (d) (e) (f)

Fig. 3. List Scheduling example.

Scheduler in each machine and implemented a graph partition algorithm to distribute nodes among machines. Each machine will now hold the entire dataflow graph and the information regarding the allocation of nodes on cluster machines, so that the scheduler knows where to send messages, depending on the destination node. Figure 2 shows the Sucuri architecture, where each machine has its own *Scheduler*. Notice that nodes in the dataflow graph and the machines are colored according to the partition.

The original Sucuri allows programmers to determine which worker could run some of the nodes in the graph. This is specially useful for placing nodes that exchange too much data in the same machine or when working with files that are local to a certain machine. This is done with the `pin` method of the node. Calling `a.pin(4)` means that node `a` can only run on worker number 4 (each worker has a global id). The *pin* method can be used to perform a manual partition of the graph, but it can be too restrictive, since it selects a worker instead of a machine. For that purpose we included the method `manual_rank` that allows programmers to associate each node to a different machine.

Since Sucuri was designed to provide transparent execution in computer clusters, there should be an automatic graph partitioning mechanism. Therefore, we have implemented an algorithm based on List Scheduling (LS), which is considered the dominant heuristic technique encountered in scheduling algorithms [14], [15], [16], [17]. The list scheduling version implemented uses a naive priority scheme that has 3 levels (worker, machine and remote). Graph nodes (tasks) are scheduled taking into account where their *incident* nodes are. Given two nodes i and n, i is called an *incident* node of n, if there is at least one edge going from i to n ($i \rightarrow n$). The algorithm start allocation source nodes in a circular way over the available cores of each machine. Then, nodes are placed according to the following the rules:

1) Allocate in the same worker of the *incident* node with greater computational cost (weight);
2) Allocate in the same machine of the *incident* node with greater computational cost;
3) If all *incident* nodes have the same computational cost, allocate in a circular way over workers of *incident* nodes;
4) If none of the above, allocate in a remote machine.

Using levels is a simple way of taking communication costs into account and removing the need to measure the network latencies and bandwidth of the entire system. However, we understand that using the real costs might improve the allocation efficiency and provide better speedups. Evaluating those aspects is part of future work.

To illustrate the List Scheduling algorithm implemented in Sucuri, Figure 3 provides an example where tasks have different weights (computation costs). Figure 3(a) shows the application's dataflow graph and Figure 3(b) shows a color-coded graph representing the node allocation (result of the algorithm). Figures 3(c), 3(d), 3(e) and 3(f) show the evolution of the algorithm. It is clear that the algorithm implemented prioritizes *incident* nodes to be allocated in the same worker or machine before going to a remote machine. By doing that, the communication cost is reduced and performance is increased.

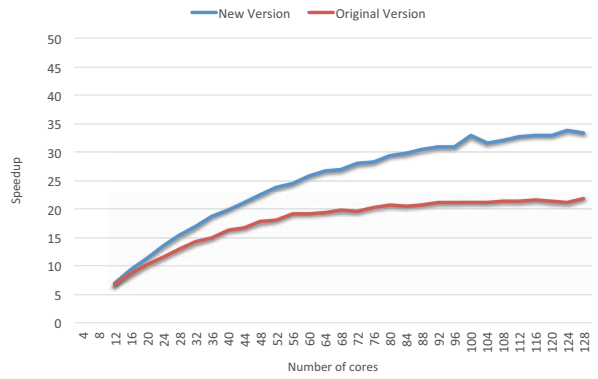
The LS algorithm is called automatically by Sucuri, which will consider that all nodes have *weight = 1*. Programmers can define the weight of each node when instantiating them. Running LS inside Sucuri may increase execution time unnecessarily, since the same application may be executed multiple times. However, this approach is useful for testing. Therefore, programmers can disable LS in Sucuri and pass an allocation file as parameter. The allocation file can be generated by Sucuri on the first execution.

It is important to notice that, since the dataflow graph is being partitioned, there is no need to replicate the full graph in all machines. Each machine could have only the sub-graph assigned to it and information about which machine holds the neighbor nodes, so that messages can be properly sent between machines. However, since our intention is to implement a work-stealing [18] mechanism in the future, machines would have to receive information on the stolen tasks. Therefore it would be easier to just keep the entire graph in each machine. Moreover, the graphs used in our experiments are relatively small and replicating them will not consume much memory.

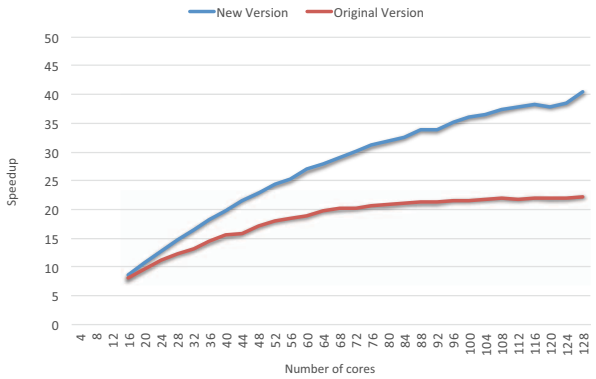
It is also important mentioning that local schedulers adopt a pool of tasks, as the original centralized scheduler. This means the local scheduler can perform load balancing inside a machine in an on-demand basis. This type of local scheduler could be a bottleneck for machines with a large number of cores. However, this is subject of future work.

IV. Experiments and Results

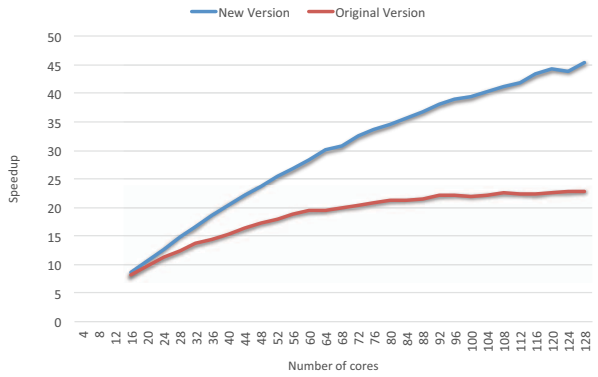
To evaluate the proposed technique, a set of four applications with different parallelism patterns was implemented in Sucuri: Longest Common Subsequence (LCS), matrix multiplication and two versions of numerical integration, one using a fork-join pattern and the other using a pipeline pattern.



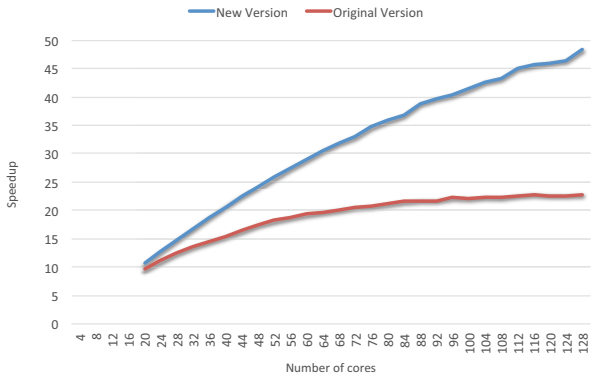
(a) 30k Sequences



(b) 45k Sequences



(c) 60k Sequences



(d) 75k Sequences

Fig. 4. Speedup for LCS.

All experiments were performed in a cluster of 32 machines. Each machine has an Intel^{TM}Core i5-3330 CPU processor (3.00GHz) with 4 cores and 8 GB of DDR3-1333MHz RAM memory, running Linux 3.8 kernel. Machines are connected by a regular network 100 Mb/s Ethernet and the NFS server is ran by a dedicated machine to store the shared data, such as input/output files.

It is important to note that each node of the resulting dataflow graph has *weight = 1* for all experiments. Moreover, LS runtime is not being computed, since the partition of the graph can be done once and used for several executions of an application.

The first experiment, the LCS algorithm, calculates the length of the Longest Common Subsequence between two input strings. Normally, LCS is implemented using dynamic program to fill up a score matrix, where each element depend on its upper, left and up-left neighbors. At the end of running, the right bottom element will contain the LCS length. LCS pattern, known as wavefront, has a important applicability in several fields, such as biology, medicine and linguistics, to find similarity between proteins, DNA molecules or words in different idioms.

Figure 4 shows the speedup of both LCS implementations (Original Sucuri and Sucuri with decentralized scheduler) to different number of cores (one worker per core). In all scenarios, the decentralized scheduler provided a better speedup than original version. It is important to mention that in the centralized version all messages are sent through machine 0 which increased memory usage, allowing the evaluation of sequences of up to $75k$. With the decentralized approach we were able to experiment with larger sequences.

Note that we had to estimate the sequential execution time for scenarios where sequences sizes that did not fit the memory of one machine, in order to calculate the speedups of both Sucuri version. A machine can evaluate sequences of up to 15kb. In [12], this estimation was performed by linear approximation, but the best approximation for that scenarios is, in fact, to use a 2 degree polynomial approximation given by equation $f(x) = 0,3398x^2 - 0,0295x + 0.0563$, where x represents the size of the input in kb and $f(x)$ is the runtime. Figure 5 shows the estimated runtime for sequential execution.

It is important to mention that the LCS graph is larger and more complex than the graph of the other 3 applications. In the worse case scenario, LS took about six seconds to run, which corresponds to 18% of the execution time of the application, meaning that running LC inside Sucuri might be easier but

not recommended for all applications.

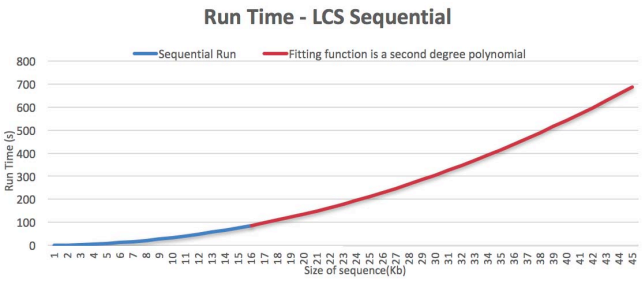


Fig. 5. Estimated runtime for sequential LCS.

The second experiment is a matrix multiplication following the fork-join pattern. To reduce network latency in the network switch, just 10 machines were used. The speedup is presented in Figure 6. Results show that static scheduling has doubled the performance when compared to the original version of Sucuri. This application consumes a lot of network bandwidth and need to be evaluated in a system with better network connection (gigabit or better).

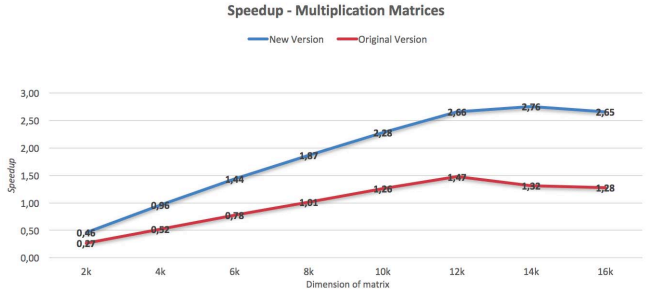


Fig. 6. Speedup for matrix multiplication.

The last two experiments consist of numerical integration implemented as fork-join and pipeline patterns. In the fork-join implementation the whole work is divided between all available workers. Figure 7 shows the results for fork-join implementation. Each node only calculates the integration of a range of points passed as inputs. This means that messages are very short and do not stress out the centralized scheduler, leaving little room for improvement by our solution.

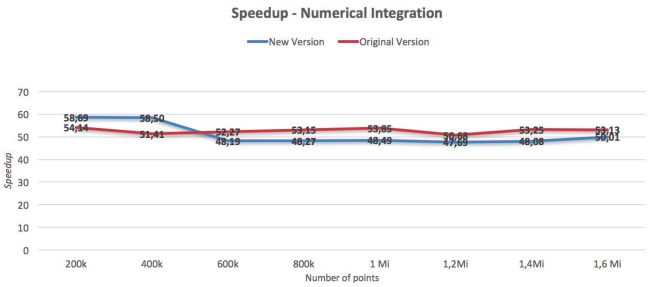


Fig. 7. Speedup for numerical integration using fork-join pattern.

In the pipeline implementation of numerical integration, three stages were used: reading from the input file, execution of numeric integration and writing to the output file. It is important to notice that although this graph has only 3 nodes, the numeric integration node is called multiple times throughout program execution, generating several tasks. Since there are no dependencies between numerical integration nodes, this application has a high level of concurrency. However, if the graph is partitioned, speedups will be limited by the number of cores in one machine because all tasks generated for the numerical integration node will be confined in one machine. Results are presents in Figure 8 and the speedups of our solution were limited, as predicted. Dynamic scheduling support, such as work-stealing, could solve this problem by allowing dynamic tasks to be run by remote idle workers.

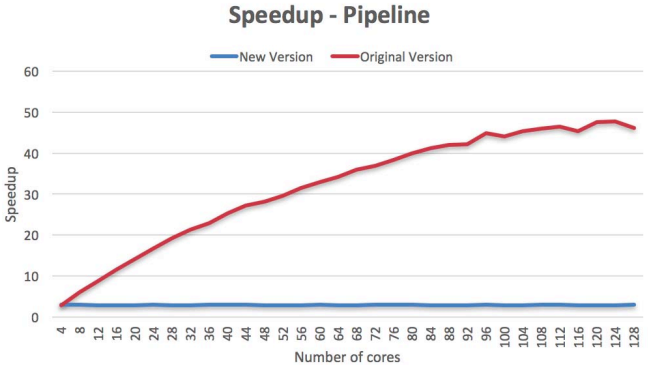


Fig. 8. Speedup for numerical integration using pipeline pattern.

V. Related Work

Intel Threading Building Block (TBB) [19] is a C++ library that provides facilities to write parallel programs on a dataflow perspective. Basically, the programmer describes flow graphs using the tbb::flow namespace, where the nodes are instantiate and connected through the make_edge function. The shortcomings of TBB are the difficulties to program in C++ and the nonexistence of dataflow templates to express easily the main parallelism patterns, since many users are not related to computer science/engineering fields. Moreover, it does not work in clusters, only in multicore machines.

Ruffus [20] is a python library for pipelined applications able to execute parallel jobs with large pieces of data. In configuration of an application, the programmer must describe I/O operations in files to the communication between the pipeline stages. That library is also not designed to work in clusters.

Auto-Pipe [21] project is composed by a toolchain and a language (X language) that provides a simple syntax to describe block connections in pipelines applications in heterogeneous architectures, such as, multicores architectures, FPGAs, GPUs etc. In addition, the language allow to map the pipeline stages with the components of the heterogeneous system.

VI. Conclusions and Future Works

In this paper we present a decentralized scheduler and graph partition solution to improve the scalability in the Sucuri, a dataflow library for Python that provides transparent execution on computer clusters. The solution adds a Sucuri scheduler in per machine, each using a ready-queue with firt-come first-served policy to serve idle workers. The dataflow graph of the application is partitioned before execution, using an algorithm based on List Scheduling.

To evaluate the proposed solution, we applied the method in four applications with different parallelism patterns: LCS (wavefront), matrix multiplication (fork-join) and numerical integration (fork-join and pipeline). As results, fork-join and wave-front applications achieved better speedups than the original implementation of Sucuri, mainly due to a reduction in communication costs. Nevertheless, the pipeline pattern had no speedups, because the graph was small and was confined to one machine of the cluster. However, since pipeline stages might generate multiple tasks, a dynamic scheduling mechanism, such as work stealing, might improve performance.

Following the idea of proposed approach, a direct optimization to improve the Sucuri is to implement a support to the dynamic scheduling scheme. Work Stealing is an appealing technique to improve performance in several applications. This technique is paramount to load balancing and to obtain the best efficiency in high performance computing. Moreover, we are extending Sucuri to allow execution on heterogeneous systems, composed of multicore processors, Xeon Phi coprocessors, GPUs and FPGA accelerators. This will require a comprehensive scheduling solution that considers computation an communication capabilities of each device. Our research group is currently working in those questions.

Acknowledgments

The authors would like to thank FAPERJ (grant E-26/203.537/2015), CNPq and CAPES for the financial support to this work.

References

[1] 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.

[2] 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. [Online]. Available: http://doi.acm.org/10.1145/2155620.2155628

[3] 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.

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

[6] 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.

[7] 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.

[8] 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.

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

[10] R. Giorgi, R. M., F. Bodin *et al.*, "TERAFLUX: Harnessing dataflow in next generation teradevices," *Microprocessors and Microsystems*, pp. –, 2014, available online 18 April 2014.

[11] O. Pell, O. Mencer, K. H. Tsoi, and W. Luk, *Maximum Performance Computing with Dataflow Engines*. New York, NY: Springer New York, 2013, pp. 747–774.

[12] T. A. O. Alves, B. F. Goldstein, F. M. G. França, and L. A. J. Marzulo, "A Minimalistic Dataflow Programming Library for Python," in *Proceedings of IEEE 26th International Symposium on Computer Architecture and High Performance Computing Workshops*, Oct 2014, pp. 96–101.

[13] B. F. Goldstein, L. A. J. Marzulo, T. A. O. Alves, and F. M. G. Franç., "Exploiting Parallelism in Linear Algebra Kernels through Dataflow Execution," in *Proceedings of IEEE 27th International Symposium on Computer Architecture and High Performance Computing Workshops*, Oct 2015, pp. 103–108.

[14] K. M. C. T. I. Adam and J. R. Dickson, "A comparison of list schedules for parallel processing systems," in *Commun. ACM, vol. 17, no.12*, 1974, pp. 685–690.

[15] O. Sinnen, *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2007.

[16] M. W. H. Topculoglu, S. Hariri, "Performance-effective and low- complexity task scheduling for heterogeneous computing," in *IEEE Trans. Parallel Distrib. Systems 13 (3)*, March 2002, pp. 260–274.

[17] M. Lombardi, M. Milano, M. RUGGIERO *et al.*, "Stochastic allocation and scheduling for conditional task graphs in multi-processor systems-on-chip," in *Journal of Scheduling doi: 10.1007/s10951-010-0184-y.*, 2010, pp. 315–345.

[18] B. R. D. P. C. G. ARORA, N. S., "Thread Scheduling for Multiprogrammed Multiprocessors," in *Theory of Computing Systems, v. 34, n. 2*, ISSN: 1432-4350. doi: 10.1007/ s002240011004, jan 2001, p. 115144.

[19] J. Reinders, "Intel threading building blocks : outfitting C++ for multi-core processor parallelims," OReilly, 2007.

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

[21] 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*, April 2006.