2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing Workshops

A Minimalistic Dataflow Programming Library for Python

Tiago A. O. Alves, Brunno F. Goldstein and
Felipe M. G. França
Programa de Engenharia de Sistemas e Computação, COPPE
Universidade Federal do Rio de Janeiro
Rio de Janeiro, Brasil
Email:{tiagoaoa,bfgoldstein,felipe}@cos.ufrj.br

Leandro A. J. Marzulo
Dep. de Informática e Ciência da Computação
Instituto de Matemática e Estatística - IME
Universidade do Estado do Rio de Janeiro - UERJ
Rio de Janeiro, Brasil
Email: {leandro}@ime.uerj.br

***Abstract*—Current work on parallel programming models are trending towards the dataflow paradigm. Previous works on that topic have shown that dataflow programming is indeed a natural way to exploit parallelism in programs. However, there is still a gap in terms of ease of programming between high level languages adopted by the scientific community and the languages and tools available for dataflow programming. In this paper we present Sucuri: a minimalistic Python library that provides dataflow programming with reasonably simple syntax. To parallelize applications using our library, the programmer needs only to identify functions of his code that are good candidates for parallelization and instantiate a dataflow graph where each node is associated with one of such functions, and the edges between nodes describe data dependencies between functions. We then proceed to implement two benchmarks that represent important parallel programming patterns using our library and execute them on a cluster of multicores. Experimental results are promising, proving that our library can be an interesting first option for parallelization.**

I. Introduction

Recent work on parallel programming models and languages [1], [2], [3], [4], [5], [6], [7] brought dataflow programming, in different levels of abstractions and granularities, as the solution for the difficulties programmers have always faced while parallelizing their code. Indeed, it is clear that in some cases the programming effort (either if measured in time or lines of code) is considerably reduced when one applies dataflow programming to parallelize their applications, as opposed to traditional tools as OpenMP or Pthreads. Also, as shown in [1], [8], dataflow execution can present performance comparable to those traditional tools. Such results corroborate dataflow as a de facto technique for parallel programming, as it offers both ease of programming and good performance.

Typically, dataflow programming is done by instantiating blocks of codes and connecting them in a graph according to their dependencies. This relieves the programmer from most of the complexity of parallelization, since the synchronization is handled by the dataflow runtime or library. Nevertheless, there still is a gap between current dataflow projects and the very high level languages vastly adopted by the scientific community, such as Python [9], R etc. Although one might argue that such languages were not designed for high performance computing and are inherently not ideal for parallel programming, their immense popularity makes it worthwhile investigating ways of accelerating their execution.

In this context, dataflow programming comes across as a suitable option for closing the gab between parallel programming and very high level languages. The reasoning for this is that dataflow programming derives from the same concept as those languages, i.e. exposing to the user only what is strictly necessary (in this case the dependencies) and keeping the complex details hidden *under the hood*. Besides, dataflow can be specially interesting when applied to object oriented (OO) languages such as Python, for it can put to use OO principles for the description of the dataflow graph (think of the program tasks as objects whose codes and graph-edges are attributes).

Following this idea, we present Sucuri, a minimalistic library for the Python language that provides dataflow programming and execution at a very high level. The idea is that the programmer will implement the functions of his program and then attribute them to *node objects*, which are in turn connected according to their functions' dependencies. After this nodes and their respective dependencies are instantiated, the program can be executed either in a single multicore machine or in a cluster of them, and the functions will be triggered according to the dataflow rule. This means that if two functions have no dependencies between them, they will potentially execute in parallel, depending on the availability of idle cores on the allocated machines.

Experimental results show that Sucuri achieved good speedups for the Longest Common Subsequence problem and almost reach a state of art implementation using MPI for Numerical Integration problem proving to be a promising library for parallelizing programs using a high level dataflow programming model.

The rest of this paper is organized as follows: Section II provides an overview of Sucuri and discusses its architecture and how to used to implemente applications; III presents and discusses experiments and results; Some related works are presented in IV. Section V presents some discussion about our library and future works.

II. Sucuri

Sucuri is a library written in Python language that aims at allowing dataflow programming in a high level. As it can be seen in Figure 1 its structure is divided in three main components: `Graph`, `Scheduler` and `Worker`. When used

978-1-4799-7014-8/14 $31.00 © 2014 IEEE
DOI 10.1109/SBAC-PADW.2014.20

in a cluster of computers, each of this components is replicated in each machine of the cluster.

The `Graph` holds a set of nodes, each containing:

- A list of operands already received and waiting to be matched
- The function to be executed when the node receives all operands it needs
- A list of destinations that will be used to send the node output to the ones that depend on it
- Specific attributes, such as an unique id that can be used to divide work between nodes.

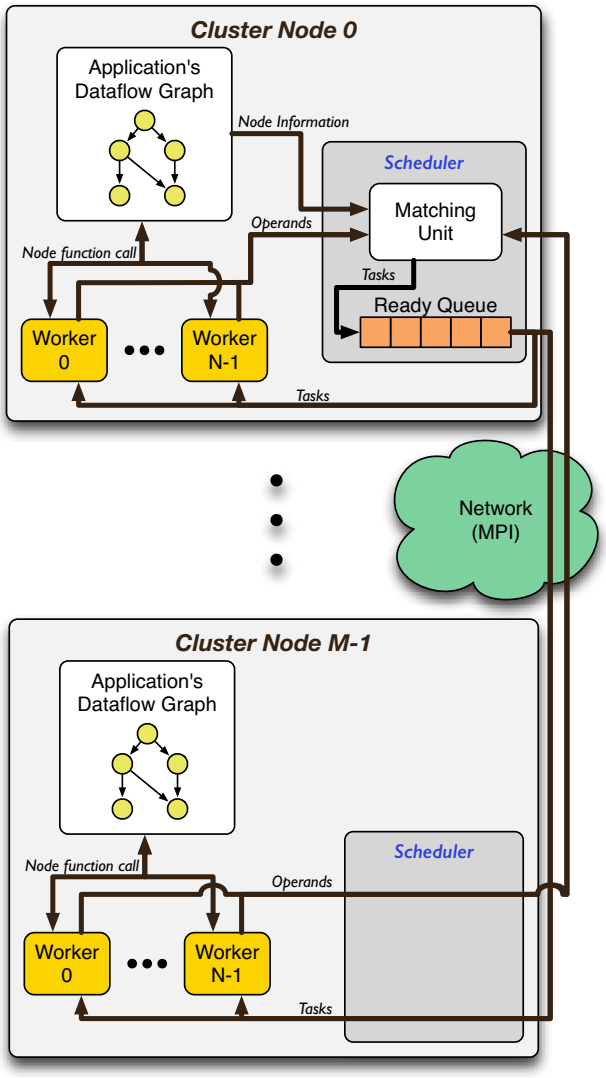


Fig. 1. The Sucuri Architecture. The same structure is replicated in each node but only the Scheduler from Cluster Node 0 contains the Matching Unit and the Ready Queue. It is responsible for receiving operands from local Workers and from the others Schedulers and also to generate tasks and put them into the Ready Queue.

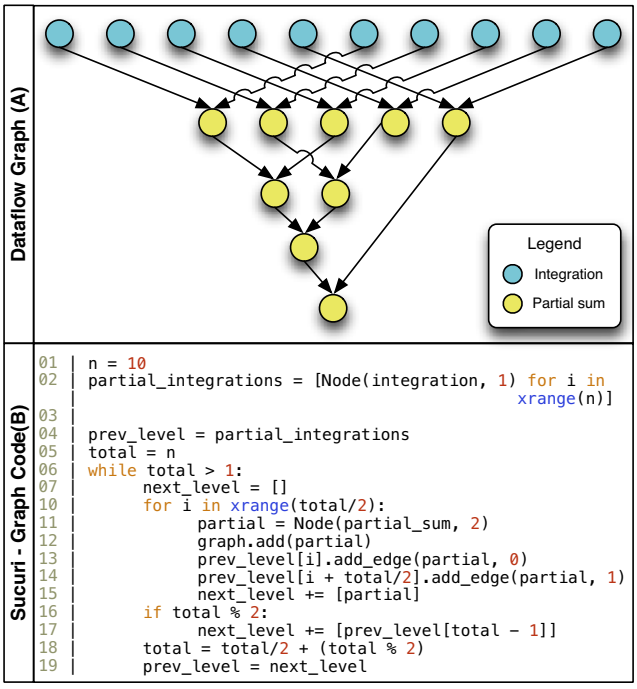
The main `Scheduler`, placed at Cluster Node 0, is composed of a Matching Unit, a Ready Queue and a Waiting Queue. It is responsible for delivering all operands to the operand list of its destination Node in the Graph. If a match happens, a task is created and put in the Ready Queue. When the `Scheduler` receives an operand containing a `REQUEST_TASK` tag from a `Worker`, a task will be removed from the Ready Queue and then delivered to the requesting Worker. The `Scheduler` placed in the other cluster nodes are more simple and only forwards tasks from the main `Scheduler` to their local workers and operands from their workers to the main `Scheduler`. The graph is replicated in all nodes of the Cluster but only the graph in node 0 can receive operands from the main `Scheduler`. The last component is responsible for consuming the tasks created by the `Scheduler`. The Worker requests a task by sending an operand to his local `Scheduler` with a tag `REQUEST_TASK`. By doing that, the local `Scheduler` can manage to get a new task at the `Ready Queue` to be processed.

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 run 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 between the main components cited above is done via shared memory and between Schedulers from different nodes is done via interface.

Each node of a Sucuri graph is associated with a function that can be implemented for any purpose. The programmer needs only to implement his custom functions and pass them to the node instantiation when creating the dataflow graph. After instantiating the nodes, the programmer can then proceed to connect them using the `add_edge()` method, which basically creates a new dependency in the dataflow graph. When the scheduler dispatches a task for execution in a certain worker, that worker will call the `run()` method of the node corresponding to that task. In most cases, this `run()` method will just act as a wrapper that calls the function associated with the node during the construction of the graph and send the values returned by the function call to the main scheduler.

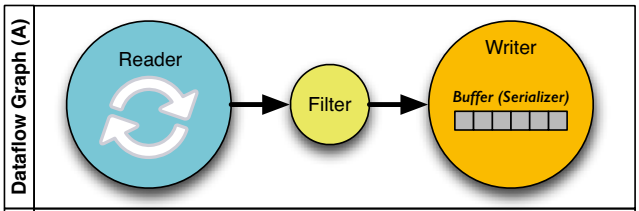
Figure 2 (panel A) shows a graph representing a hierarchical reduction and Sucuri's code for this operation is described in panel B. First, all `integration` nodes are created in line 2 for the first level of the graph. Then, for each iteration at line 10, `partial_sum` nodes are created in the order of half of total nodes from previous level. At line 13 and 14, each `partial_sum` node created is now connected to two nodes from previous level.

Now a pipeline pattern is presented in Figure 3. Panel A shows a graph representation of this pattern and panel B the Sucuri's code for this operation. To implement an application with pipeline pattern (sometimes also called streaming programming) the programmer can instantiate special nodes designed for that purpose. The special node `Source` receives an *iterable* python object (for instance, a list or a file descriptor) at instantiation. During program execution, the `run()` method of the *Source* node will be fired only once, since this node works as a root, i.e., has no input operands from the graph and serves to initiate the computation. The execution of such method, however, will typically last until the contents of the iterable object are exhausted. By default the `Source` node will loop over the iterable object and pass the values retrieved in a call




```
01 | n = 10
02 | partial_integrations = [Node(integration, 1) for i in
   |                                                 xrange(n)]
03 |
04 | prev_level = partial_integrations
05 | total = n
06 | while total > 1:
07 |       next_level = []
10 |       for i in xrange(total/2):
11 |             partial = Node(partial_sum, 2)
12 |             graph.add(partial)
13 |             prev_level[i].add_edge(partial, 0)
14 |             prev_level[i + total/2].add_edge(partial, 1)
15 |             next_level += [partial]
16 |       if total % 2:
17 |             next_level += [prev_level[total - 1]]
18 |       total = total/2 + (total % 2)
19 |       prev_level = next_level
```

Fig. 2. Hierarchical Reduction with Sucuri. Panel A shows the dataflow graph of the application, panel B describes the graph using Sucuri.




```
01 | from Sucuri import *
02 | import sys
03 | nprocs = int(sys.argv[1])
04 |
05 | graph = DFGraph()
06 | sched = Scheduler(graph, nprocs)
07 | fp = open(sys.argv[2], "r")
10 |
11 | reader = Source(fp)
12 | filter = Node(filter_line, 1)
13 | writer = Serializer(write_line, 1)
14 |
15 | reader.pin(0)
16 | writer.pin(1)
17 | graph.add(reader)
18 | graph.add(filter)
19 | graph.add(writer)
20 |
21 | reader.add_edge(filter, 0)
22 | filter.add_edge(writer, 0)
23 | sched.start()
```

Fig. 3. Pipelining with Sucuri. Panel A shows the dataflow graph of the application, panel B describes the graph using Sucuri.

of the function associated with the node. When the associated function returns, the `run()` method sends the returned values as operands to the nodes that depend on that `Source` and goes on to the next iteration of the loop. The `Source` will not become idle after producing an operand, since it produces operands at every loop iteration. Therefore it is important that these operands do not contain the `REQUEST_TASK` tag. Consequently, after finishing the loop (for instance, at the end of the file being read) the node must send a null operand with the `REQUEST_TASK` piggybacked just to signalized that it became idle and needs new tasks.

The data produced by the `Source` node is consumed by a regular node that performs a filter function on it. After applying the filter function to the data, that node forwards the processed data to the `Serializer` node, that is responsible for writing the data to a file. It is possible for the data produced by the `Source` node to be processed out of order by the second node, since the multiple tasks may be scheduled to different workers. Therefore, it is necessary to reorder the data before writing it to the file, in the `Serializer` node. For that purpose, the data produced by the `Source` node must be encapsulated in a `TaggedValue` object, which contains a `tag` attribute, indicating its position it the ordered data set. The middle node will also send the filtered data inside of a `TaggedValue` object, with the same tag of the chunk of data it received. The `Serializer` node then, upon receiving data from the filter node, will store it in a buffer sorted according to the tag. If the tag of the last piece of data received corresponds to the next data to be written to the file, the `Serializer` node proceeds to pop data from the sorted buffer and write to the file until there is a gap in the ordering, i.e. the chunk of data that is the next to be written 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. In the case of the example, we pin the nodes that performs IO operations on disk to the workers that have direct access to that disk.

Figure 3 (panel B) shows how Sucuri is used to describe the pipeline graph. The following steps are worth noting:

- The initialization of the program and the graph and also the assignment of input parameters are described in line 1 to line 7.
- Node `Reader` is initialized in line 11, receiving the file pointer (fp) as its iterable operand.
- Nodes `Filter` and `Writter` are instantiated in lines 12 and 13, each one with their specific function(`filter_line` and `write_line`).
- The `Reader` node is of type `Source` where all the data will be consumed and encapsulated in a `TaggedValue` object, while the `Writter` is a `Serializer` type node where the data will be store in a buffer sorted by tags to be written in the right sequence..
- At lines 15 and 16, `Reader` and `Writer` nodes are pinned to worker 0 and 1, respectively, since those workers have direct access to the disk.
- All nodes created are add to the graph in lines 17, 18 and 19.
- Dependencies edges are created in lines 21 and 22.

III. Experiments and Results

As discussed in Section II, dataflow application described in Sucuri can be executed in clusters of computers without the

need to change its code. To evaluate the performance of this solution a set of two applications that adopt representative parallel patterns were implemented in Sucuri: LCS and Numerical Integration.

Our experimental environment consists of a cluster with 32 nodes each having an Intel® Core™ i5-3330 CPU (3.00GHz), 8GB of DDR3-1333MHz memory and runs Linux 3.8 kernel. Interconnection is a regular 100 Mb/s Ethernet and one of the nodes runs a NFS server to store executables and input/output files.

Our first experiment, the LCS algorithm, calculates the length of a Longest Common Subsequence, or Levenshtein distance, between two sequences of symbols in dataflow, as described in [8]. The LCS algorithm is important for several fields, such as biology, medicine and linguistics. In linguistics, for example, LCS provides information about similarities between words written in two idioms, since the Levenshtein distance is the minimum number of characters insertions, deletions or substitutions needed to turn one word into the other. In Genetics, LCS is used to find similarities between to protein or DNA molecules, represented as chains of nucleotides.

LCS traditional implementation [10] uses dynamic programming to fill up a score matrix, where elements depend on their upper, left and upper-left neighbors. The LCS length will be the bottom right element of the score matrix. To find out one of the longest common subsequences a traceback is need. In this experiment, we did not implement the traceback.

Figure 4 shows the implementation details of LCS. Panel A shows the dataflow graph of the application, made of a matrix of nodes that will calculate the score matrix in a distributed way. Each node will calculate a block of the score matrix, called Local Score Matrix (LSM). Notice that each node will receive the last line of the LSM form its upper neighbor and will insert this data on its own LSM. It will also receive the last column of the LSM from its left neighbor and insert its data on the first column of his own LSM. The dependence with the upper-left neighbor is transitive and, therefore, those edges are eliminated.

Figure 4 (panel B) shows how Sucuri is used to describe the graph. The following steps are worth noting:

- At line 1 we import the Sucuri module.
- At line 2 we instantiate a new (empty) graph.
- At line 3 we create a Scheduler, that receives the graph and the number of desired workers.
- Form lines 6 to 8, we add all nodes to the graph.
- From lines 10 to 15 we create all necessary communication edges.
- From lines 17 to 19 we create the printer node, add it to the graph and connect it with the bottom right node of the LCS graph.
- At line 20 we start the execution of our graph, commanded by the scheduler.

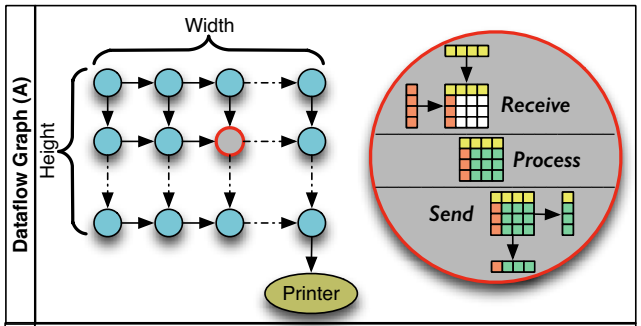
Figure 4 (panel C) shows the source code for one LCS node. The following steps are worth noting:

- From lines 22 to 26 we calculate which parts of the sequences will be used by each node. This is done using nodes coordinates (i, j) and the block size (passed by ordinary command line parameters, not shown in the figure).
- Line 28 allocates the LSM.
- From lines 30 to 32 the node receives the last LSM line from his upper neighbor and insert it at his own LSM.
- From lines 34 to 36 the node receives the last LSM column from his left neighbor and insert it at the first line of his LSM.
- line 45 returns both the line and column fro his LSM. Operand sending is managed by the schedulers.

Figure 5 presents the performance results of LCS on the Cluster. The x axis show the number of workers (cores) used, while the y axis shows the obtained speedups. Each line provide the results for a pair of sequences of the same size (15k, 30k, 45k, 60k and 75k). A block size of 150×150 was used. Notice that for 128 workers, we managed to run LCS about 40 times faster than the sequential Python version. Also, for smaller matrices we could observe that it stops scaling, since the number of blacks will be small. For example, when we compare two 15k sequences with blocks of 150×150, it results on a total of $10,000$ blocks (100×100); for 75k sequences we have a total of $250,000$ blocks (500×500 blocks). The maximum theoretical speedup (MxSpd) of LCS is given by the number of blocks divided by the size of the critical path, which is the number of diagonals of the graph (or $Width + Height - 1$), since all nodes have the same computational cost. For the 15k matrix, the maximum speedup is 50.25, while for the 75k matrix, $MxSpd = 250.25$.

Notice that the results do not show the speedups for sequence sizes in all scenarios. LCS can be executed in one cluster node for sequences with at most 15k elements. To calculate the speedups for other scenarios, we had to use an estimation of the baseline time, since we could not actually run it (not enough memory). To estimate the baseline time, we ran LCS for 1k to 15k, with an increase of 1k per scenario and observed a linear behavior. We then just divided total time of the 15k $\times$ 15k scenario by the number of computed elements (225M) and multiplied this time by the number of elements in each scenario we needed to predict. We find this to be a fair method, since the actual times could not be better then linear.

The second experiment is a simple numerical integration. The algorithm uses a fork-join pattern to divide the whole job among all available workers. We measured the performance of a Sucuri and a C with MPI implementation with an input size of 10^6 using up to 32 nodes of the Cluster and 4 Workers in each node. Figure 6 shows the speedup obtained for our scenario when compared to a sequential version in C. Results show that with a simple dataflow implementation of numerical integration, the Sucuri's library reach MPI implementation for a few number of Workers. When the number of Worker increases, Sucuri speedup get a little bit distant from MPI curve. Since Sucuri uses a centralized `Scheduler`, this behavior was expected with the increasing number of nodes/workers. This distance between the two speedup curves



Sucuri - Graph Code(B)

```
01  from Sucuri import *
02  lcsGraph = DFGraph()
03  sched = Scheduler(lcsGraph, nprocs)
04  G = [[Node2D(LCS,inputs(i,j),i,j) for j in
                xrange(Width)] for i in xrange(Height)]
05
06  for i in xrange(Height):
07    for j in xrange(Width):
08      lcsGraph.add(G[i][j])
09
10  for i in xrange(Height):
11    for j in xrange(Width):
12      if i > 0:
13        G[i-1][j].add_edge(G[i][j],0,0)
14      if j > 0:
15        G[i][j-1].add_edge(G[i][j],int(i>0),1)
16
17  printer = Node(printLCS, 1)
18  lcsGraph.add(printer)
19  G[Height-1][Width-1].add_edge(printer,0)
20  sched.start()
```

LCS function(C)

```
21  def LCS(self,oper):
22    i = self.i
23    j = self.j
24    startA, endA, startB, endB = getRanges(i,j)
25    lsizeA = endA - startA
26    lsizeB = endB - startB
27
28    LSM = [[0 for x in xrange(lsizeA+1)] for y in
                                      xrange(lsizeB+1)]
29    port=0
30    if i>0:
31      for x in xrange(len(oper[0])):
32        LSM[0][x]=oper[0][x]
33      port=port+1
34    if j>0:
35      for x in xrange(len(oper[port])):
36        LSM[x][0] = oper[port][x]
37
38    for ii in xrange(1, lsizeB+1):
39      for jj in xrange(1, lsizeA+1):
40        if seqB[startB+ii-1] == seqA[startA+jj-1]:
41          LSM[ii][jj] = LSM[ii-1][jj-1] + 1
42        else:
43          LSM[ii][jj] = max(LSM[ii][jj-1], LSM[ii-1][jj])
44
45    return (LSM[lsizeB],[m[lsizeA] for m in LSM])
```

Fig. 4. LCS Application. Panel A shows the dataflow graph of the application, panel B describes the graph using Sucuri and panel C shows the python function that computes a portion of the LCS score matrix.

can be minimized with an implementation of a hierarchical distributed scheduler.

It is important to remember that the programmer does not need to know if his target environment is a cluster of multicores or a simple multicore machine. This abstraction layer is provided by Sucuri's library as said before. Moreover, although this application uses math functions that are wrappers of C functions (as in many python libraries), comparing a python implementation with another written in pure C is quite unfair, since python is an interpreted language. We find the results to be quite promising, since programmers can use Sucuri to get a much higher level of abstraction and still obtain performance comparable to MPI for C.

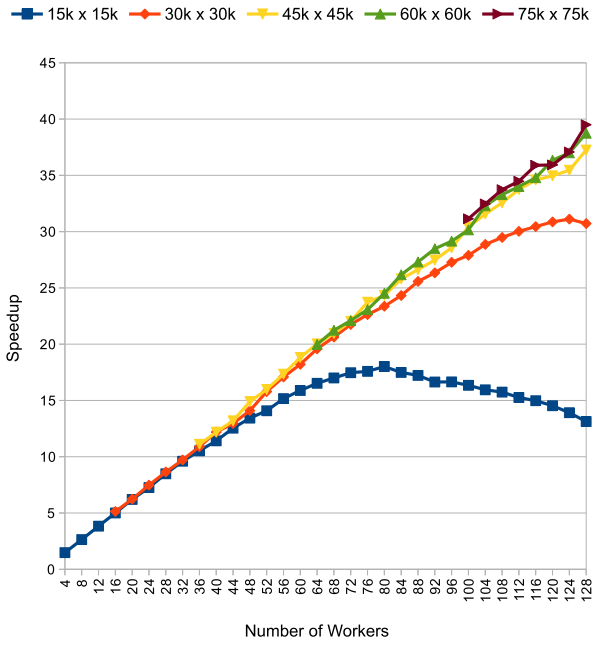


Fig. 5. Results for the LCS Application. The x axis show the number of workers (cores) used, while the y axis shows the obtained speedups. Each line provide the results for a pair of sequences of the same size (15k, 30k, 45k, 60k and 75k).

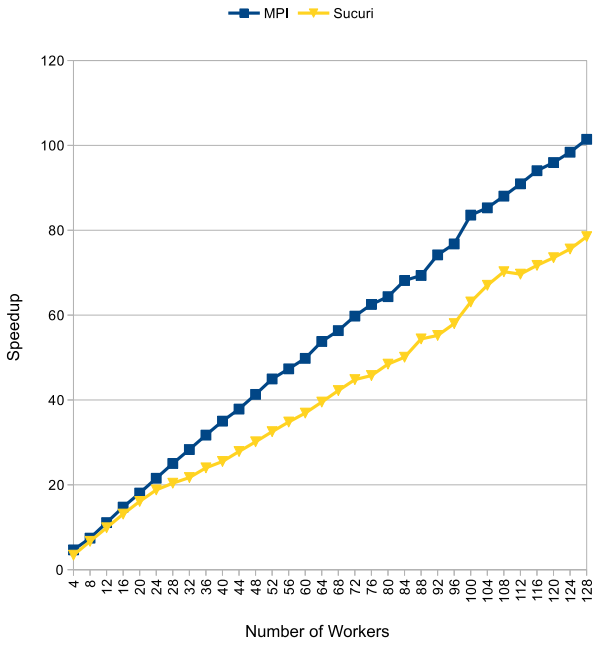


Fig. 6. Results for the Numerical Integration Application. The x axis show the number of workers (cores) used, while the y axis shows the obtained speedups. Each line provide the results for and input of size 10^6, one for Sucuri implementation and the other for a MPI implementation.

IV. Related Work

Auto-Pipe [11] 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 X-language, the language employed in Auto-Pipe, provides a simple syntax where the user connects the blocks corresponding to the pipeline stages and can specify which of the components of the heterogeneous system will execute which state. StreamIt [12] is another programming language intended for streaming programming, it allows the implementation of large stream based applications. Although Sucuri is more generic, allowing the implementation of different patterns, not only pipelines, these functionalities are also easily obtained with Sucuri.

Intel Threading Building Block (TBB) [13] 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, which may be too difficult for users in fields not directly related to computer science/engineering, and the fact that it only works in multicore machines, not clusters.

Ruffus [14] is a python library that provides support for the execution of parallel jobs on large pieces of data in pipelined applications. The communication between pipeline stages in Ruffus is done via I/O operations in files specified in the configuration of the application. Due to the focus on pipelines only, the syntax in Ruffus is more succint than Sucuri's. However, Ruffus is not designed to work in clusters.

V. Discussion and Future Work

In this work we have presented Sucuri, a minimalistic library for the Python language that provides dataflow programming and execution at a very high level. Our library proved to be a promising way to parallelize programs using a dataflow model in order to achieve high degrees of parallelism. With a simple syntax in Python language, the programmer can construct the dependence graph by creating nodes with functions associated and then connect all of them based on their dependencies using the `add_edge()` method. Our library is very versatile, allowing the program to be transparently executed on clusters of multicores.

To improve performance, we will implement a backend in C language that will provide a hierarchical distributed scheduler to substitute our centralized scheduler implemented in Python. This implementation will provide the scalability necessary to enable Sucuri to run in bigger clusters with minimal performance penalty.

The Python runtime offers the possibility of implementing modules in C language and compile them as shared libraries to be called in Python code. This creates a great opportunity for Sucuri, since it allows programmers to implement the kernels of their applications in C and connect these kernels in a dataflow graph using Sucuri, thus achieving both good performance and ease of programming. Therefore, it is important to do further experimentation with benchmarks implemented in this fashion in order to assess the maximum performance that can be achieved using Sucuri.

Since the `Node` class can be extended for specific purposes, a library containing all kinds of nodes can be implemented, making Sucuri even easier to use. The same can be done with the creation of templates of the `Graph` class for the most common parallel patterns. Programmers will then be able to create their applications by connecting subgraphs with regular nodes.

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] 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.

[3] 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.

[4] 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

[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] 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, available online 18 April 2014.

[8] T. A. Alves, L. A. J. Marzulo, and F. M. G. Franca, "Unleashing parallelism in longest common subsequence using dataflow," in *4th Workshop on Applications for Multi-Core Architectures*, 2013.

[9] G. Rossum, "Python reference manual," Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 1995.

[10] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *J. ACM*, vol. 21, no. 1, pp. 168–173, Jan. 1974. [Online]. Available: http://doi.acm.org/10.1145/321796.321811

[11] 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.

[12] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=647478.727935

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

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