Parallel Programming Tools in Python

Guillermo Cornejo-Suárez *†, Esteban Meneses*†
* Escuela de Computación, Instituto Tecnológico de Costa Rica
† Colaboratorio Nacional de Computación Avanzada, Centro Nacional de Alta Tecnología
Email: {gcornejo, emeneses}@cenat.ac.cr

I. Introduction

The Python programming language is widely used in scientific computing. Five rankings on the popularity of programming languages with different methodologies [1], [2], [3], [4] and [5] list Python in position 1, 4, 2, 3 and 3, respectively. Those rankings considered the number of results returned by web searches, Github projects, questions tagged in StackOverflow, open job positions, among other variables. As stated in [6], in 2015 approximately 10% of all computer jobs at Texas Advanced Computing Center were Python applications. Also, there exists specific modules for a wide diversity of scientific disciplines. Because of this strong code base and ease of use, Python is commonly the first language learned by engineering, basic and applied science students.

In the current context, multi-core and many-core processors represent industry's answer to Moore's Law technical halt. New code must be parallel to take advantage of modern architectures, but at the moment, there is no obvious answer on how to offer parallelism in the Python programming language through idiomatic abstractions. Python's community has generated a plethora of projects providing parallelism trough modules, function calls, job submission, annotations, just-in-time compilers, and run time systems.

Considering its popularity, growth potential, and absence of *de facto* standard for parallel computing, providing parallelism in the Python programming language is a promising research field. An important step to propose effective solutions is understanding which strategies have already been deployed and its results. In this work, we present an extensive survey of projects that aim to provide parallelism in Python.

Other similar works focus on comparing usability and performances for some parallel tools, for example: [7] compares Python together with Cython, Numpy, Numba and Global Arrays against Fortran code solving a CDF problem; [8] qualifies usability and performance of parallel extensions for high level languages; [9] presents an end-to-end argument for Python as a suitable language in high performance environments. To the best of our knowledge, this is the first work to survey and classify Python parallel tools in an extensive manner.

II. Methodology

In order to understand the different approaches delivered by the Python community and which combination of characteristics are common, a systematic review of published projects was carried out. It included indexed publications: IEEExplore, ACM Digital Library, SciencieDirect, SpringerLink, as well as Google Scholar and commercial Google. "Parallel python" was used as search key phrase. Results were sorted by date in descending order from 2016 until 1993. After analyzing close to fifty works, we established the following inclusion criteria:

1) The project provides some kind of parallelism to the Python programming language.
2) The project aims general programming rather than domain-specific.

Because interpreted languages commonly have lower performance than compiled languages, many projects intent to improve python performance by using ahead-of- and just-in-time compiling [10], [11], [12], or tuning distributed platforms for launching Python [6], [13]. Clearly these strategies are interesting for the high performance computing community. Nevertheless, this paper focuses on the problem of how to write parallel code with Python, and the previously discussed strategies were excluded.

A total of 35 projects were included, and classified according to the following criterion:

- **Execution strategy:** Interpreted, binary binding, compiled. This refers to how code written by programmer is finally executed.
- **Parallel paradigm:** Data, Task, Message Passing, MapReduce. How parallelism is exposed to the programmer, not necessarily how it is implemented.
- **Vector Data oriented:** If the project fundamentally offers parallel vector structures.
- **Language support:** Full, or Subset. Which version of python is supported (Python 2 or Python 3) and if the whole language is supported or only a subset.
- **Code modifications:** Function calls, Job submission, Annotations, None. How a programmer must modify a given serial program to run in parallel using the tool.
- **Target platform:** Symmetric multiprocessing (SMP), Clusters, GPU. The particular architecture for which the tool was designed.

III. Results and Analysis

Table I presents the result of classifying each project. First column presents project name and reference. The second column classifies each project according to its execution strategy. 60% of projects use the interpreter to provide parallelism, commonly this implies the project itself is also written in Python. This decision simplifies and accelerates development, but makes harder to overcome natural limitations of Python,

TABLE I
CLASSIFICATION OF TOOLS THAT PROVIDE PARALLELISM IN THE PYTHON PROGRAMMING LANGUAGE.

Project	Execution strategy	Parallel paradigm	Vector data oriented	Language support	Code modifications	Parallel platform	Latest release
Bohrium [14]	Interpreted	Data	Yes	Full, Python 2	None	SMP, GPU, Clusters	0.3, Apr-2016
PyStream [15]	Compiled	Data	Yes	Subset, Python 2	None	GPU	0.1, Jul-2011
Dask.array [16]	Interpreted	Data	Yes	Full, Python 3	FunCall	SMP, Clusters	0.13.0, Jan-2017
PupyMPI [17]	Interpreted	MsgPsg	No	Full, Python 2	FunCall	SMP, Clusters	0.9.5, May-2011
Papy [18]	Interpreted	Task	No	Full, Python 2	JobSub	SMP, Clusters	1.0.8, Nov-2014
GAiN [19]	Binary binding	Data	Yes	Full, Python 2	FunCall	Clusters	1.0, 2009
Global Arrays [20]	Binary binding	Data	Yes	Full, Python 2	FunCall	Clusters	5.5, Aug-2016.
mpi4Py [21]	Binary binding	MsgPsg	No	Full, Python 2-3	FunCall	SMP, Clusters	2.0.0, Oct-2015
Pythran [22]	Compiled	Data	Yes	Subset, Python 3	Annotations	SMP	0.7.6.1, Jul-2016
ASP [23]	Binary binding	Data, Task	No	Full, Python 2	JobSub	SMP, GPU	0.1.3.1, Oct-2013
Dispel4py [24]	Interpreted	Data, Task	No	Full, Python 2-3	JobSub	SMP, Clusters	1.2, Jun-2015
PMI [25]	Interpreted	Data	No	Full, Python 2-3	FunCall	SMP, Clusters	1.0, Dec-2009
Jit4OpenCL [26]	Compiled	Data	Yes	Full, Python 2	Annotations	SMP, GPU	1.0, 2010
MRS [27]	Interpreted	MapRed	No	Full, Python 2-3	FunCall	Clusters	0.9, Nov-2012
Pydron [28]	Interpreted	Task	No	Subset	Annotations	Clusters	–
CoArray [29]	Interpreted	Data	Yes	Full, Python 2	FunCall	Clusters	2004
PyCuda, PyOpenCL [30]	Binary binding	Data	Yes	Full, Python 2-3	FunCall	SMP, GPU	2016.2, Oct-2016
SCOOP [31]	Interpreted	Task	No	Full, Python 2-3	JobSub	SMP, Clusters	0.7.1.1, Ago-2015
DistArray [32]	Interpreted	Data	Yes	Full, Python 2-3	JobSub	SMP, Clusters	0.6, Oct-2015
Dispy [33]	Interpreted	Data, MapRed	No	Full, Python 2-3	JobSub	SMP, Clusters	4.6.17, Sep-2016
IpyParallel [34]	Interpreted	Data, Task	No	Full, Python 2-3	JobSub	SMP, Clusters	5.3.0, Oct-2016
PyRo [35]	Interpreted	MsgPsg	No	Full, Python 2-3	Annotations, FunCall	Clusters	4.50, Nov-2016
Parallel python [36]	Interpreted	Task	No	Full, Python 2-3	JobSub	SMP, Clusters	1.6.5, Jul-2016
JUG [37]	Interpreted	Task	No	Full, Python 2-3	Annotations, FunCall	SMP, Clusters	1.3.0, Nov-2016
Multiprocessing [38]	Interpreted	Task, Data	No	Full, Python 2-3	FunCall	SMP, Clusters	3.6, Jul-2016
Copperhead [39]	Binary binding	Data	Yes	Subset, Python 2	Annotations	GPU	2013
Celery [40]	Interpreted	Task	No	Full, Python 2-3	Annotations, FunCall	SMP, Clusters	4.0.0, Nov-2016
Disco [41]	Interpreted	MapRed	No	Full, Python 2	Annotations, FunCall	SMP, Clusters	0.5.4, Oct-2014
Spark [42]	Binary binding	Task	No	Full, Python 2-3	FunCall	Clusters	2.0.2, Nov-2016
Theano [43]	Binary binding	Data	Yes	Full, Python 2-3	FunCall	SMP, GPU, Clusters	0.8.2, Apr-2016
Numba [44]	Compiled	Data	Yes	Full, Python 2-3	Annotations	SMP, GPU	0.29.0, Oct-2016
Joblib [45]	Interpreted	Task	No	Full, Python 2-3	JobSub, Annotations	SMP	0.10.3, Oct-2016
Hadoopy [46]	Binary binding	MapRed	No	Full, Python 2	JobSub	Clusters	0.5.0, Jun-2012
PyMW [47]	Interpreted	Task	No	Full, Python 2	FunCall	Clusters	0.4, Jun-2010
Pyfora [48]	Compiled	Data	No	Subset, Python 2	None	Clusters, SMP	0.5.8, Set-2016

like the *global interpreter lock* (GIL). Following interpretation, 26% of the projects use binary binding to execute code written by programmer. Projects like NumPy use this strategy to improve performance. This gives the possibility to use well-established parallel computing technology developed for other languages and releasing the GIL, but interfaces must be carefully designed to provide enough functionality in an idiomatic way. Finally, 14% of the projects attempt to compile Python code to machine language. This requires type inference or just-in-time compiling, in general, those projects try to provide parallelism in a transparent way.

Third column classifies projects according to their parallel paradigm. Data is the most common choice, 50% of the projects offer data parallelism. The second most common choice is Task parallelism, with a 31% of the projects. The fact that data and task parallelism add up to 81% of the projects possibly respond to the nature of most common parallel jobs. In scientific computing, data sets are efficiently represented as multidimensional vectors, so it makes sense to provide data parallelism. Other applications require multiple functions executing concurrently, for example: web servers, multimedia display, gaming; so, it is useful to provide an abstraction for tasks. Message Passing and MapReduce are also present, with 10% and 9% of the projects. They are not a very popular choice, possibly because Message Passing forces programmers to think about low level abstractions like processes, and this breaks idiomatic Python. Similarly, MapReduce forces a programming pattern that does not fit many parallel problems.

Fourth column states if the project fundamentally offers parallel vector structures and holds an strong correlation with third column. 37% of the projects provide vectors as principal structures, all of them expose data parallelism, however, the other way around is not true. Fifth column shows project Language Support. 86% of projects supports the full language. This column holds a relation with second column, projects that attempt compiling Python code to machine language commonly narrow Python support and because the majority of projects are interpreted, and written in Python itself, supporting the full language is easy. Also, this column states if Python 3 is supported, supporting both is very common, even so, 40% of projects only support Python 2, and plans to upgrade to Python 3 are mixed, strongly depending on project continuity.

Sixth column talks about Code modifications. 43% of the projects use Function call and 27% use Job submission, this holds a relation with parallel paradigm in column three. Function calls are suitable for implementing universal operations over a collection of data and job submission makes explicit the creation of a new task. When attempting compiling code to machine language or using parallel library as a backend (CUDA, for example) Annotations (21%) or not change at all (9%) are preferred, this also holds a relation with column three, the majority of those projects provide data parallelism.

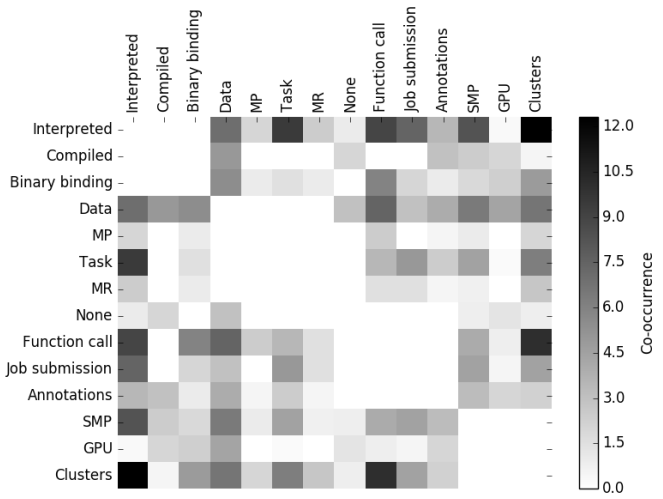

Fig. 1. Co-occurence matrix for characteristics in table I. Scale indicates the Number of projects that share the same characteristic

Seventh column shows target Parallel platform. This classification is not mutually exclusive, many projects run on two or three platforms, percentages are then pondered accordingly. 51% of the projects support cluster computing, while 36% support SMP. Python's standard library provides Multiprocessing and Threading modules, an attempt to create idiomatic abstractions for parallel programming. Those modules simplify creation of new instances of interpreters and hardware threads. Because of that, many projects exploit shared memory and clusters over the Python interpreter. For example, some projects create multiple instances of the interpreter in different computers, creating a cluster of interpreters, others exploit threads in shared memory platforms. Again, this holds a relation with column two, using the interpreter to provide parallelism is a popular choice. GPUs represent 13% of the projects. Exploiting GPUs from Python is more difficult, they have a separate memory from principal memory and are programmed with hardware-dependent libraries. In general, projects targeting GPUs use CUDA or similar libraries as backend, so they are classified as Compiled or Binary binding.

Finally, eighth column presents latest release date: version number, month and year. 48% of projects released a new version the last year and 60% in the past two years, also, 46% of projects have a *stable* release, that means, a version number bigger than 1.0. This suggests there is an important effort of the community to address the problem.

These relations could be represented with a co-occurrence matrix as shown in Figure 1. Vector data oriented and Language support were excluded from the representation to avoid visual cluttering. One interesting observation is that even when Interpretation, Data parallelism, and Function call are the preferred design decisions, they do not co-occur together frequently. Interpreted and Clusters exhibit an strong co-occurrence. As explained above, Multiprocessing module, from Python's standard library, simplifies creating multiple instances of the Python interpreter, in consequence, many projects written in Python will target clusters. In the same line of Interpreted, Task and Function call show relevant co-occurrence, suggesting that when implemented above the Python interpreter, offering Task parallelism and exposing parallelism trough Function calls is common, but the observed co-occurrence of both characteristics is not high, meaning Function calls are not used to implement Task parallelism. Function call and Clusters are another co-occurrence, if you create many instances of the Python interpreter, it is simple to interact with them trough function calls, possibly manipulating a collection or invoking a remote procedure.

Also, it is interesting to analyze the absence of some co-occurrences. Compiled only occurs for Data parallelism through Annotations or not change to code, and only targets SMP and GPUs; all other fields are empty. That suggests those projects are trying to take advantage of architectural features like vectorization to process data in parallel by mapping operations to high optimized BLAS functions or common kernels.

IV. Conclusions

An extensive survey of projects that provide parallelism for the Python programming language was carried out. Projects were classified according to how code is executed, how parallelism is exposed, if vectors are the principal data structure, language support, how code must be modified to fit the projects, which platform does the project targets and latest dated of release.

The most common characteristics are: using Python interpreter to run the code or using external code (binary binding), offering data parallelism or task parallelism, if data parallelism is offered in general the main data structure will be vectors. Majority of projects support full language specification, but compiled projects tent to exclude some part. Parallelism is exposed as function calls or explicit job submission. Targeting SMP and clusters platform is common, the latter more than the former.

References

[1] N. Diakopoulos and S. Cass, "Interactive: The top programming languages 2017," http://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017, 2017, accessed: 08/07/2017.

[2] T. S. BV, "Tiobe index for may 2017," https://www.tiobe.com/tiobe-index/, 2017, accesed: 05/23/2017.

[3] P. Carbonnelle, "Pypl popularity of programming language," http://pypl.github.io/PYPL.html, 2017, accesed: 05/23/2017.

[4] S. O'Grady, "The redmonk programming language rankings: January 2017," http://redmonk.com/sogrady/2017/03/17/language-rankings-1-17/, 2017, accesed: 05/23/2017.

[5] J. Patel, "The 9 most in-demand programming languages of 2017," http://www.codingdojo.com/blog/9-most-in-demand-programming-languages-of-2017/, 2017, accesed: 05/23/2017.

[6] T. Evans, A. Gómez-Iglesias, and C. Proctor, "Pytacc: Hpc python at the texas advanced computing center," in *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing*, ser. PyHPC '15. New York, NY, USA: ACM, 2015, pp. 4:1–4:7. [Online]. Available: http://doi.acm.org/10.1145/2835857.2835861

[7] A. Basermann, M. Röhrig-Zöllner, and J. Illmer, "Performance and productivity of parallel python programming: A study with a cfd test case," in *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing*, ser. PyHPC '15. New York, NY, USA: ACM, 2015, pp. 2:1–2:10. [Online]. Available: http://doi.acm.org/10.1145/2835857.2835859

[8] L. Humphrey, B. Guilfoos, H. Smith, A. Warnock, J. Unpingco, B. Elton, and A. Chalker, "Evaluating parallel extensions to high level languages using the hpc challenge benchmarks," in *2009 DoD High Performance Computing Modernization Program Users Group Conference*, June 2009, pp. 410–415.

[9] M. Mortensen and H. P. Langtangen, "High performance python for direct numerical simulations of turbulent flows," *Computer Physics Communications*, vol. 203, pp. 53 – 65, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465516300200

[10] J. Akeret, L. Gamper, A. Amara, and A. Refregier, "Hope: A python just-in-time compiler for astrophysical computations," *Astronomy and Computing*, vol. 10, pp. 1 – 8, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2213133714000687

[11] R. Garg and J. N. Amaral, "Compiling python to a hybrid execution environment," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 19–30. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735695

[12] B. Ren and G. Agrawal, "Compiling dynamic data structures in python to enable the use of multi-core and many-core libraries," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011, pp. 68–77.

[13] Yu Feng and Nick Hand, "Launching Python Applications on Peta-scale Massively Parallel Systems," in *Proceedings of the 15th Python in Science Conference*, Sebastian Benthall and Scott Rostrup, Eds., 2016, pp. 137 – 143.

[14] T. Blum, M. R. B. Kristensen, and B. Vinter, "Transparent gpu execution of numpy applications," in *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, ser. IPDPSW '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 1002–1010. [Online]. Available: http://dx.doi.org/10.1109/IPDPSW.2014.114

[15] N. Bray, "Pystream: Compiling python onto the gpu," in *Proceedings of the 10th Python in Science Conference*, 2011, pp. 79–82.

[16] M. Rocklin, "Dask: Parallel computation with blocked algorithms and task scheduling," in *Proceedings of the 14th Python in Sciencie Conference (SciPy 2015)*, 2015, pp. 130–136.

[17] R. Bromer, F. Hantho, and B. Vinter, "pupympi - mpi implemented in pure python," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 130–139. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042476.2042492

[18] M. Cieslik and C. Mura, "Papy: Parallel and distributed data-processing pipelines in python," in *Proceedings of the 8 th Python in Science Conference (SciPy 2009)*, 2009, pp. 41–49=8.

[19] J. Daily and R. R. Lewis, "Using the global arrays toolkit to reimplement numpy for distributed computation," in *Proceedings of the 10 th Python in Science Conference (SciPy 2011)*, 2011, pp. 23–29.

[20] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory programming toolkit," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 203–231, May 2006. [Online]. Available: http://dx.doi.org/10.1177/1094342006064503

[21] L. D. Dalcin, R. R. Paz, P. A. Kler, and A. Cosimo, "Parallel distributed computing using python," *Advances in Water Resources*, vol. 34, no. 9, pp. 1124 – 1139, 2011, new Computational Methods and Software Tools. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0309170811000777

[22] S. Guelton, J. Falcou, and P. Brunet, "Exploring the vectorization of python constructs using pythran and boost simd," in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '14. New York, NY, USA: ACM, 2014, pp. 79–86. [Online]. Available: http://doi.acm.org/10.1145/2568058.2568060

[23] S. Kamil, D. Coetzee, and A. Fox, "Bringing parallel performance to python with domain-specific selective embedded just-in-time specialization," in *Proceedings of the 10th Python in Sciencie Conference (SciPy 2011)*, 2011, pp. 83–89.

[24] A. Krause, R. Filgueira, and M. Atkinson, "Dispel4py: A python framework for data-intensive escience," in *Proceedings of the 5th Workshop on Python for High-Performance and Scientific Computing*, ser. PyHPC '15. New York, NY, USA: ACM, 2015, pp. 6:1–6:10. [Online]. Available: http://doi.acm.org/10.1145/2835857.2835863

[25] O. Lenz, "Pmi - parallel method invocation," in *Proceedings of the 8th Python in Science Conference (SciPy 2009)*, 2009, pp. 48–51.

[26] X. Li, R. Garg, and J. N. Amaral, "A new compilation path: From python/numpy to opencl," in *Workshop on Python for High Performance and Scientific Computing (PyHPC), 2011*, 2011.

[27] A. McNabb, J. Lund, and K. Seppi, "Mrs: Mapreduce for scientific computing in python," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, Nov 2012, pp. 600–608.

[28] S. C. Muller, G. Alonso, and A. Csillaghy, "Scaling astroinformatics: Python + automatic parallelization," *Computer*, vol. 47, no. 9, pp. 41–47, Sep. 2014. [Online]. Available: http://dx.doi.org/10.1109/MC.2014.262

[29] C. E. Rasmussen, M. J. Sottile, J. Nieplocha, R. W. Numrich, and E. Jones, *Co-array Python: A Parallel Extension to the Python Language*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 632–637. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27866-5_83

[30] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih, "Pycuda and pyopencl: A scripting-based approach to {GPU} run-time code generation," *Parallel Computing*, vol. 38, no. 3, pp. 157 – 174, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111001281

[31] Y. Hold-Geoffroy, O. Gagnon, and M. Parizeau, "Once you scoop, no need to fork," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, ser. XSEDE '14. New York, NY, USA: ACM, 2014, pp. 60:1–60:8. [Online]. Available: http://doi.acm.org/10.1145/2616498.2616565

[32] IPython Development Team and Enthought, Inc.., "Distarray 0.6," http://distarray.readthedocs.io/en/v0.6.0/, 2015, accessed: 09/10/2016.

[33] G. Pemmasani, "dispy: Distributed and parallel computing with/for python," http://dispy.sourceforge.net/, 2014, accessed: 09/09/2016.

[34] The IPython Development Team, "Using ipython for parallel computing," https://ipyparallel.readthedocs.io/en/latest/, 2015, accessed: 11/15/2016.

[35] I. de Jong, "Pyro - python remote objects," http://pythonhosted.org/Pyro4/, 2016, accessed: 09/10/2016.

[36] V. Vanovschi, "Parallel python," http://www.parallelpython.com/, 2016, accessed: 08/28/2016.

[37] L. P. Coelho, "Jug: A task-based parallelization framework," https://jug.readthedocs.io/en/latest/, 2016, accessed: 09/15/2016.

[38] P. S. Foundation, "multiprocessing — process-based parallelism," https://docs.python.org/3.6/library/multiprocessing.html, 2016, accessed: 08/29/2016.

[39] B. Catanzaro, "Copperhead: Data parallel python," https://devblogs.nvidia.com/parallelforall/copperhead-data-parallel-python/, 2013, accessed: 09/08/2016.

[40] A. S. . Contributors, "Celery - distributed task queue," http://docs.celeryproject.org/en/latest/index.html, 2015, accessed: 09/10/2016.

[41] The Disco Project, "Disco: Massive data - minimal code," http://discoproject.org/, 2014, accessed: 09/12/2016.

[42] Spark development team, "Spark overview," https://spark.apache.org/docs/0.9.0/index.html, accessed: 09/13/2016.

[43] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, may 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[44] Continuum Analytics, "Numba," http://numba.pydata.org/, 2012, accessed: 08/28/2016.

[45] G. Varoquaux, "Joblib: running python functions as pipeline jobs," https://pythonhosted.org/joblib/, 2009, accessed: 09/17/2016.

[46] B. White, "Hadoopy: Python wrapper for hadoop using cython," http://www.hadoopy.com/en/latest/index.html, 2012, accessed: 09/17/2016.

[47] Eric Heien, "Pymw," http://http://pymw.sourceforge.net/, 2010, accessed: 10/18/2016.

[48] Ufora, Inc., "Pyfora," http://docs.pyfora.com/en/stable/, 2016, accesed: 11/22/2016.

ACKNOWLEDGMENT

G.C.S. was supported by "Asistente Especial de posgrado" scholarship of Instituto Tecnológico de Costa Rica.