Seamless Scaling of Applications Across Programming Models

Reto Krummenacher
Department of Mathematics and Computer Science,
University of Basel
Basel, Switzerland

Quentin Guilloteau
Univ. Lyon, Inria, CNRS, ENSL, UCBL, LIP
Lyon, France

Jonas H. Müller Korndörfer
Astronomical Institute, University of Bern
Bern, Switzerland

Florina M. Ciorba
Department of Mathematics and Computer Science,
University of Basel
Basel, Switzerland

ABSTRACT

We present a comparative study of the productivity and performance of four programming languages: Python, Julia, C++, and DaphneDSL, for the Connected Components graph algorithm from the GAP benchmark suite. Using various code productivity metrics, we evaluated the effort of scaling applications from a local parallel version to a distributed implementation. Experiments carried out on the Vega EuroHPC system reveal that, with moderate coding effort, Julia offers the best performance, while DaphneDSL enables seamless distributed execution with no code changes, albeit at a small performance cost.

1 INTRODUCTION

Many programming languages require substantial effort to express parallelism. An exception is DAPHNE [4], where its domain-specific language (DaphneDSL) requires no code changes to express parallelism, locally or between computing nodes. Hence, it seamlessly scales and is highly productive for writing parallel programs.

In this work, we evaluate the productivity and performance of the well-known programming languages Python, Julia, and C++ against DaphneDSL. We consider coding in a programming language to be productive when only a few changes are required in an implementation to express local and distributed parallelism.

This poster builds on own previous work [6] with additional productivity metrics: (1) the source lines of code [2, 11], (2) tokens of code [2, 5, 12], (3) Halstead's effort [2, 5, 12], and (4) cyclomatic complexity number [2, 5, 12].

We implemented the Connected Components (**CC**) graph algorithm of the GAP benchmark suite [3] in Python, Julia, C++, and DaphneDSL. We evaluated the above implementations through experiments on the Vega EuroHPC system [8] with the Wikipedia-20070206 [15] sparse matrix as input.

2 METHODOLOGY

2.1 Algorithm implementation

For each language, we implemented CC in two forms: a sequential and local parallel form (denoted *SePa)* and a distributed parallel form that uses MPI (denoted *Dist*). The exception is DaphneDSL, where the same code can be executed both locally and distributed without code changes due to the underlying DAPHNE infrastructure [4]. All implementation details can be found in our recent work [6].

2.2 Quantifying coding productivity

We use four metrics [1] to measure coding productivity. The source lines of code (**SLOC**), representing the number of lines of code excluding comments and blank lines. The tokens of code (**TOC**) is defined as the sum of the total number of operators N_1 and operands N_2 [1]:

$$TOC = N_1 + N_2. \tag{1}$$

N_1 includes punctuation, arithmetic, comparisons, member access, control flow, and types. N_2 are variables, objects, object methods, and classes. Both SLOC and TOC are a measure of program length.

Coding effort can be measured with the Halstead effort metric (**EFF**) [7], which requires the total number of operators N_1 and operands N_2 and the number of distinct operators n_1 and operands (n_2) [13]:

$$EFF = D \times V, \quad \text{width} \tag{2}$$

$$D = (n_1/2) \times (N_2/n_2), \quad \text{and} \tag{3}$$

$$V = (N_1 + N_2) \times log_2(n_1 + n_2) \tag{4}$$

where D is a measure of the difficulty of a program and V of its volume.

To understand the complexity of a program, we use the Cyclomatic Complexity Number (**CCN**), which captures the number of paths through a directed program control graph G with n nodes, e directed edges, and p connected components [10]:

$$CCN = v(G) = e - n + 2 \times p. \tag{5}$$

All metrics are automatically obtained with the *multimetric* [14] tool. We extended the tool to enable support for DaphneDSL and adapted the original implementation to collect SLOC instead of the classical LOC metric. For each language (Pyhon, Julia, C++, DaphneDSL), code implementation (*SePa*, *Dist*), and metric (SLOC, TOC, EFF, CCN), we report metric values normalized to the respective *SePa* value.

2.3 Experimental evaluation

We conducted three sets of experiments to evaluate strong scaling performance.

(A) Strong scaling performance of exploiting **node-level parallelism** using the *SePa* form on a single node.
(B) Strong scaling performance of exploiting **node-level parallelism** using the *Dist* form on a single node.
(C) Strong scaling performance of exploiting **cross-node parallelism** using the *Dist* form and various MPI process counts on multiple nodes.

Table 1: Design of factorial experiments to evaluate the performance of Python, Julia, C++, and DaphneDSL.

Factor	Value	Properties
Set of Strong Scaling Experiments	(A) Exploiting node-level parallelism on a single node	Language: Python, Julia, C++, DaphneDSL; Number of computing nodes: 1; Number of Threads: 1, 2, 4, 8, 16, 32;
	(B) Exploiting node-level parallelism with MPI on a single node	Language: Python, Julia, C++, DaphneDSL; Number of computing nodes: 1; Number of MPI ranks per node: 1, 2, 4, 8, 16; Number of threads per rank: 1;
	(C) Exploiting cross-node parallelism with MPI on multiple nodes	Language: Python, Julia, C++, DaphneDSL; Number of computing nodes: 1, 2, 3, 4, 5; Python, Julia, C++: Number of MPI ranks per node: 16; Number of threads per rank: 1; DaphneDSL: Number of MPI ranks per node: 1; Number of threads per rank: 16;
Benchmark Suite	GAP suite of graph algorithms	Connected Components algorithm
Input	Sparse matrix representing a graph	wikipedia-20070206: [15] Size (rows×cols): 3’566’907×3’566’907; Density (%): 3.54 $\times 10^{-4}$
Computing system	Vega EuroHPC [8] CPU standard partition node	2× AMD Rome 7H12 (64 cores, 4 NUMA domains, 2.6 GHz, 280 W) 256 GB RAM, 1× HDR100 single port mezzanine, 1× 1.92 TB M.2 SSD
Experiment	Repetitions	5
Metric	Execution time	End-to-end, data reading, and compute time (seconds)

Table 2: Normalized and *absolute* coding productivity metrics for the sequential and local parallel form (*SePa*) and distributed parallel form (*Dist*) of the connected components algorithm: source lines of code (SLOC), tokens of code (TOC), Halstead's effort (EFF), and cyclomatic complexity number (CCN). The colors in the table cells indicate productivity from high, to less high, to medium, and to low.

Language (abbrv.)	External dependencies	SLOC		TOC		EFF		CCN	
		SePa	*Dist*	*SePa*	*Dist*	*SePa*	*Dist*	*SePa*	*Dist*
Python (`py`)	Numpy, Scipy	1.0 *23*	4.3 *99*	1.0 *178*	5.6 *988*	1.0 *12’884*	23.2 *298’492*	1.0 *3*	4.7 *14*
Julia (`jl`)	MatrixMarket	1.0 *67*	1.5 *99*	1.0 *358*	2.2 *779*	1.0 *55’852*	3.8 *210’888*	1.0 *13*	1.0 *13*
C++ (`cpp`)	Eigen	1.0 *60*	2.5 *151*	1.0 *518*	2.7 *1’392*	1.0 *111’054*	5.3 *588’888*	1.0 *7*	2.3 *16*
DaphneDSL (`daph`)	∅	1.0 *12*	1.0 *12*	1.0 *117*	1.0 *117*	1.0 *6’884*	1.0 *6’884*	1.0 *3*	1.0 *3*

The work is equally distributed among the workers. The DAPHNE runtime system allows for various scheduling settings [6].

To arrive at comparability between languages in terms of work distribution and scheduling, we used the default (static)scheduling configuration for each language. By design, rank 0 in DAPHNE is only a coordinator and not a worker. Therefore, all DAPHNE experiments include an additional MPI process. The exact specification for each set of experiments is shown in Table 1.

3 RESULTS AND DISCUSSION

3.1 Normalized and absolute metric values

The normalized and the *absolute* metrics values are shown in Table 2. As expected, *no coding effort is required to use DaphneDSL in distributed environments*. When adapting code for distributed execution, Python is the least productive of the studied languages. Python has the highest normalized values across all metrics (SLOC, TOC, CCN, EFF), with EFF being the most pronounced.

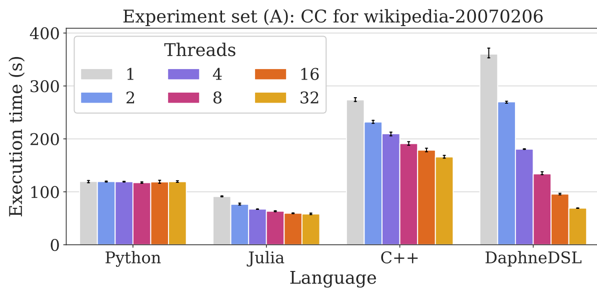
Writing MPI-aware Python code requires more than 23× the effort needed to write the local parallel version. Julia has lower normalized values than C++ and is, therefore, considered more productive.

3.2 Performance

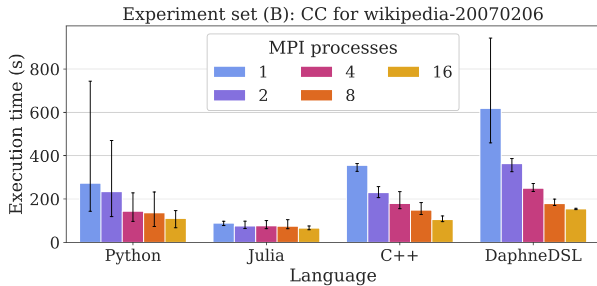
The results of the experiment set (A) in Figure 1a show that there is some strong scaling benefit from using multiple threads, with DAPHNE profiting the most. Python does not exhibit scaling benefit because certain functionality from external dependencies (e.g., Numpy's `maximum`) are serialized by the Global Interpreter Lock, and only a single Python thread can run at a time. Interestingly, even though DAPHNE is the slowest when employing a few threads, it outperforms Python and C++ with an increased thread count. Julia exhibits the highest performance, while Python is faster than C++.

Similar results can be observed in the experiment set (B), but with Python and C++ performing in the same way (Figure 1b).

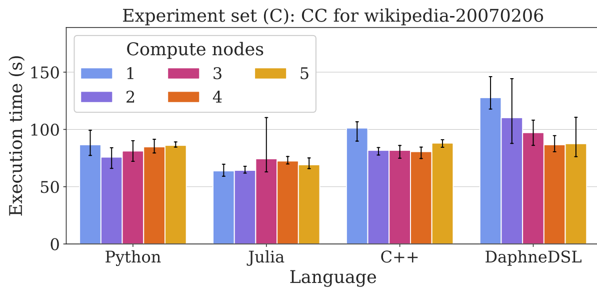
The results of the experiment set (C) in Figure 1c reveal that there is no strong scaling benefit from additional compute nodes for Python, Julia, and C++. DAPHNE is outperformed again, despite showing benefits from strong scaling. In summary: (1) Julia clearly



(a) End-to-End execution time for experiment set (A)



(b) End-to-End execution time for experiment set (B)



(c) End-to-End execution time for experiment set (C)

Figure 1: Results over 5 repetitions with various levels of parallelism for connected components benchmark with the wikipedia-20070206 matrix. Error bars show minimum and maximum over the 5 repetitions. Executed on Vega EuroHPC CPU standard partition, each node with 2× AMD Rome 7H12 (64 cores, 4 NUMA domains, 2.6 GHz, 280 W) 256 GB RAM.

shows the best performance. (2) Python outperforms C++ when executed with threads and equals the performance of C++ with MPI processes. (3) C++ outperforms DAPHNE in most cases.

4 CONCLUSION AND FUTURE WORK

In this study, we evaluated the productivity and performance of the Connected Components graph algorithm implemented in Python, Julia, C++, and DaphneDSL. Julia is the language with the highest performance and decent productivity. In contrast, DAPHNE with DaphneDSL has high productivity but lower performance in most cases. Work is currently ongoing to implement other graph algorithms from the GAP benchmark suite (e.g., PageRank, Breadth First Search) and evaluate their coding productivity and performance on other input data and HPC systems.

ACKNOWLEDGMENTS

This research was funded in part by the European Union's Horizon 2020 research and innovation program under grant agreement No. 957407 as DAPHNE. Vega access was granted under the approved EuroHPC proposal ID 2025D03-111.

OPEN RESEARCH AND REPRODUCIBILITY

All artifacts related to this work are available on Zenodo [9].

REFERENCES

[1] Gabriella Andrade, Dalvan Griebler, Rodrigo Santos, Marco Danelutto, and Luiz G. Fernandes. 2021. Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, Palermo, Italy, 291–295. https://doi.org/10.1109/SEAA53835.2021.00044

[2] Gabriella Andrade, Dalvan Griebler, Rodrigo Santos, Christoph Kessler, August Ernstsson, and Luiz G. Fernandes. 2022. Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, Gran Canaria, Spain, 229–232. https://doi.org/10.1109/SEAA56994.2022.00043

[3] Scott Beamer, Krste Asanović, and David Patterson. 2017. The GAP Benchmark Suite. https://doi.org/10.48550/arXiv.1508.03619

[4] Patrick Damme, Marius Birkenbach, Constantinos Bitsakos, Matthias Boehm, Philippe Bonnet, Florina Ciorba, Mark Dokter, Pawel Dowgiallo, Ahmed Eleliemy, Christian Faerber, Georgios Goumas, Dirk Habich, Niclas Hedam, Marlies Hofer, Wenjun Huang, Kevin Innerebner, Vasileios Karakostas, Roman Kern, Tomaž Kosar, Alexander Krause, Daniel Krems, Andreas Laber, Wolfgang Lehner, Eric Mier, Marcus Paradies, Bernhard Peischl, Gabrielle Poerwawinata, Stratos Psomadakis, Tilmann Rabl, Piotr Ratuszniak, Pedro Silva, Nikolai Skuppin, Andreas Starzacher, Benjamin Steinwender, Ilin Tolovski, Pınar Tözün, Wojciech Ulatowski, Yuanyuan Wang, Izajasz Wrosz, Aleš Zamuda, Ce Zhang, and Xiao Xiang Zhu. 2022. DAPHNE: An Open and Extensible System Infrastructure for Integrated Data Analysis Pipelines. In *Conference on Innovative Data Systems Research*.

[5] Jorge Fernandez-Fabeiro, Arturo Gonzalez-Escribano, and Diego R. Llanos. 2019. Simplifying the Multi-GPU Programming of a Hyperspectral Image Registration Algorithm. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, Dublin, Ireland, 11–18. https://doi.org/10.1109/hpcs48598.2019.9188064

[6] Quentin Guilloteau, Jonas H Müller Korndörfer, and Florina M Ciorba. 2024. Seamlessly Scaling Applications with DAPHNE. In *COMPAS 2024 - Conférence francophone d'informatique en Parallélisme, Architecture et Système,*, Vol. hal-04637841. antes, France.

[7] T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. 2017. Software Complexity Analysis Using Halstead Metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*. 1109–1113. https://doi.org/10.1109/ICOEI.2017.8300883

[8] IZUM. 2025. HPC Vega. https://en-vegadocs.vega.izum.si/introduction/ Accessed August 16, 2025.

[9] Reto Krummenacher, Quentin Guilloteau, Jonas H Müller Korndörfer, and Florina M Ciorba. 2025. Artifacts for SC25 Poster Seamless Scaling of Applications Across Programming Models. *Zenodo* (2025). https://doi.org/10.5281/zenodo.17201042

[10] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (Dec. 1976), 308–320. https://doi.org/10.1109/tse.1976.233837

[11] Biagio Peccerillo and Sandro Bartolini. 2019. Task-DAG Support in Single-Source PHAST Library: Enabling Flexible Assignment of Tasks to CPUs and GPUs in Heterogeneous Architectures. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*. ACM, Washington DC, USA, 91–100. https://doi.org/10.1145/3303084.3309496

[12] Gabriel Rodriguez-Canal, Yuri Torres, Francisco J. Andújar, and Arturo Gonzalez-Escribano. 2021. Efficient Heterogeneous Programming with FPGAs Using the Controller Model. *The Journal of Supercomputing* 77, 12 (Dec. 2021), 13995–14010. https://doi.org/10.1007/s11227-021-03792-7

[13] Yahya Tashtoush, Mohammed Al-Maolegi, and Bassam Arkok. 2014. The Correlation among Software Complexity Metrics with Case Study. *International Journal of Advanced Computer Research* 4 (Aug. 2014).

[14] Konrad Weihmann. 2025. multimetric. https://github.com/priv-kweihmann/multimetric Version 2.2.2. Accessed July 10, 2025.

[15] Wikipedia-20070206. 2007. SuiteSparse Matrix Collection. https://sparse.tamu.edu/Gleich/wikipedia-20070206.