Seamlessly Scaling Applications with DAPHNE

Quentin Guilloteau, Jonas H. Müller Korndörfer, Florina M. Ciorba

Compas'2024 : Parallélisme / Architecture/ Système
IETR/LS2N/Nantes-U - Nantes - France, 2-5 juillet 2024

Seamlessly Scaling Applications with DAPHNE

Quentin Guilloteau, Jonas H. Müller Korndörfer, Florina M. Ciorba

Department of Mathematics and Computer Science, University of Basel, Switzerland
{quentin.guilloteau,jonas.korndorfer,florina.ciorba}@unibas.ch

---

Résumé
When developing scientific and/or data science applications, users might start by using a scripting language to quickly produce a prototype, but eventually need to rewrite their code in a higher performance language when performance becomes a bottleneck. A similar situation arises when the applications need to scale beyond a single node. Users need to consequently adapt their code to integrate domain decomposition, synchronization, and communication libraries such as MPI. In this paper, we present ongoing work on the distributed scheduler of DAPHNE (https://daphne-eu.eu), an infrastructure for optimizing data analysis pipelines. With DAPHNE, users can write their applications *once* in a high level language, as they would in Python or Julia, and benefit from seamless scalability across computing nodes.

Mots-clés : parallel applications, scalability, MPI, scheduling, load balancing

---

1. Introduction

When developing a new scientific or data science application, researchers are faced with a dilemma : either develop the applications quickly with scripting languages such as Python which might yield poor performance, or invest time and effort into an implementation in high performance compiled languages such as C++ or Rust. Moreover, if an application is initially written in Python as a proof of concept, but lacks performance, it will then need to be fully rewritten in a high performance language. This issue is known as the "Two languages problem" [3]. The Julia programming language [4] positioned itself as a solution combining both the ease of writing of a scripting language, while yielding high performance with *Just-in-Time* compilation.
In addition to efficient programming languages, current scientific and data science applications need to exploit multiple levels of hardware parallelism to achieve high performance and efficiently use resources. Expressing multilevel parallelism in such applications is not trivial and requires considerable effort independently of the programming language.
Thus, we claim that there exists a variation of the "Two languages problem" for scaling parallel applications. Users might achieve parallelization using third-party libraries that support multithreading and scale applications to several cores of a *single* node. However, scaling applications to multiple nodes requires much greater effort as no language or library offers seamless features for domain decomposition, synchronization, and communication. If users want to scale their applications by distributing computations onto several nodes, they will need to, once again, rewrite at least part of their application to integrate communication libraries such as MPI [25].
The DAPHNE framework [9] aims to solve this need for reimplementation by considering data representations and efficient execution plans, allowing holistic optimizations. DAPHNE is a

scalable infrastructure for executing integrated data analysis pipelines. Such pipelines can integrate steps of HPC, Data Management, and Machine Learning. As such, the main objects that DAPHNE manipulates are matrices. Users write their pipelines in the DaphneDSL [12], the syntax of which resembles Python, Numpy, Julia, and R. The DaphneDSL scripts are then converted into an LLVM MLIR [23] dialect from which, after several optimization passes, DAPHNE extracts the data and operations that can be performed in parallel. DAPHNE can also exploit and offload computations to accelerators such as GPUs and FPGAs [10].
In this paper, we present *ongoing work* on the distributed scheduler – **DaphneSched** – of the DAPHNE framework, which builds upon DAPHNE's local runtime and scheduler [15]. With DaphneDSL, users can write their application *once* in a high level scripting language and benefit from *seamless parallelism and scalability across computing nodes*.
This paper is structured as follows : the novel distributed scheduler of DAPHNE is presented in Section 2. Section 3 describes how to write an application in various programming languages, in sequential, parallel, and distributed versions, discussing the characteristics of each implementation. The performance of each implementation is compared in single and multiple node executions, in Section 4. The work is concluded in Section 5, with perspectives for future work.

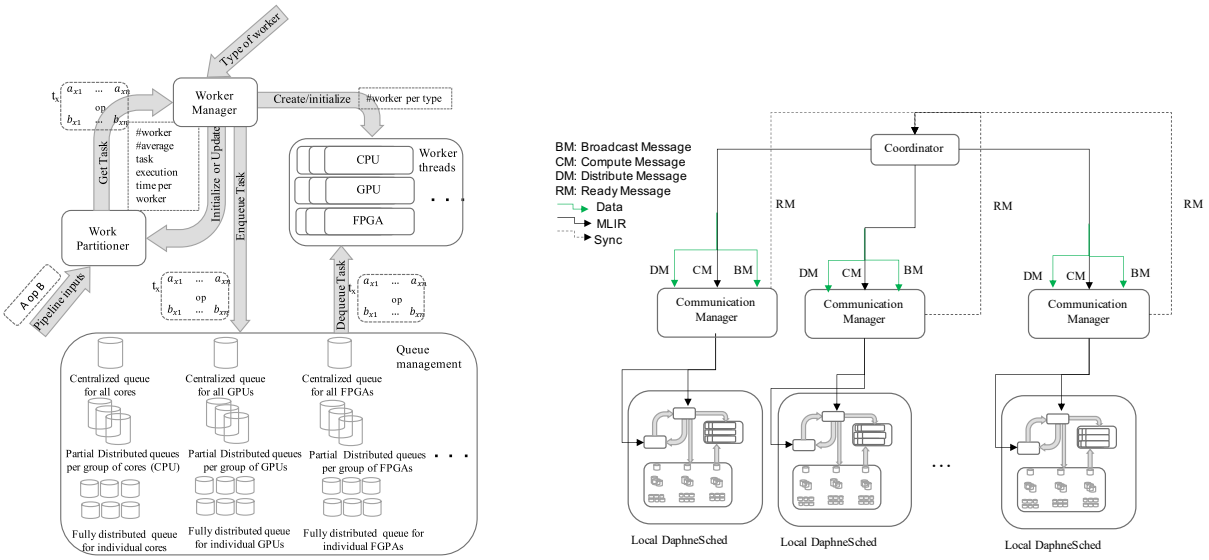
2. DaphneSched

2.1. Local Scheduler

The local DaphneSched has been presented in [15]. For completeness, we summarize its characteristics here and depict it in Figure 1a. DAPHNE combines *data and operator parallelism* where data are decomposed between workers and various operators are applied on the partitioned data simultaneously. Partitioned data and operators (MLIR code) are combined into *tasks*, thereby supporting task parallelism. The size of the different data partitions, *i.e.,* number of rows or columns, defines the size of a task. Tasks are formed using the input data following a given partitioning/scheduling technique that defines tasks of different size (*i.e.,* tasks with different amount of data). Currently, the local DaphneSched supports STATIC, SS, GSS, TSS, FAC2, TFSS, FISS, VISS, PLS, MSTATIC, MFSC, PSS, and AUTO [26] (further details about these scheduling techniques can be found in [16, 27]). Finally, the tasks are enqueued in central or distributed queues that workers query to obtain their next task to execute in a self-scheduling fashion.
The local DaphneSched also provides multiple queue layouts : `CENTRALIZED` (all workers query a single centralized queue), `PERGROUP` (all workers of the same NUMA domain query the same queue), and `PERCPU` (each worker has its own local queue). By using several task queues, load balancing is achieved in local DaphneSched through work-stealing. Several work-stealing victim selection strategies are supported : `SEQ` (steal from the next queue according to hardware topology), `RND` (steal from a random queue), and two NUMA-aware strategies, `SEQPRI` and `RNDPRI`, which prioritize stealing from queues in the same NUMA domain.

2.2. Distributed Scheduler

The distributed DapnheSched is based on its local counterpart as depicted in Figure 1b, providing two backends [31] : with MPI [25] or with gRPC [17]. In this paper, we only focus on the MPI backend as many data analysis pipelines have been shown to use MPI [2, 28]. With the MPI backend, each rank is an instance of the DAPHNE local runtime and its local scheduler. The only exception is MPI rank 0 – the *coordinator*, with the role of dividing, distributing, and collecting the tasks and their results to/from the local DAPHNE runtime instances. The coordinator does not perform any other computation. Currently, the coordinator equally divides the work between the local DapnheSched instances, and no work stealing occurs between these.



(a) Local DaphneSched (from [15]) (b) Distributed DaphneSched (this work)

FIGURE 1 – DaphneSched design for the local (left) and distributed (right) scheduler.

Hence, there is a trade-off between using fewer resources by co-locating the *coordinator* on a node alongside a computing rank, and using a distinct computing node for each MPI rank.

3. Methodology

3.1. Considered Application

We consider the *Connected Components* (CC) algorithm from Graph Theory which is part of the GAP benchmark suite [1], and compare its implementation for single and multiple nodes in C++, Python, Julia, and DaphneDSL. CC can be expressed as operations on the adjacency matrix of the input graph. Hence, all CC implementations will use matrix operations to be comparable to DAPHNE. As input data, we consider 3 sparse matrices from the *Sparse matrix collection* [13] the characteristics of which are included in Table 2 in the Appendix.

3.2. Considered Languages

We expect the C++ implementation of CC to outperform all other languages, at the cost of greater implementation difficulty. By difficulty we mean the 'distance' between the mathematical formulation of the algorithm and its actual implementation in a programming language. C++ requires a third-party library to perform linear algebra operations. We used the popular Eigen library [18], which proposes a user-friendly interface and achieves high performance.

For implementing CC in Python, we had to use two third-party dependencies. Numpy [20] for linear algebra operations and Scipy [30] to manage sparse matrices.

The Julia [4] implementation of CC required only a single dependency to load matrices stored in the MatrixMarket file format [21]. All linear algebra features and support for sparse matrices are present in the standard library of Julia.

No third-party dependency was needed to implement the CC algorithm in DaphneDSL.

For the distributed versions of all these implementations, we used MPI and the MPI wrappers associated with each language (*i.e.*, MPI4py [8] for Python, and MPI.jl [7] for Julia). Table 1 characterizes the different implementations according to their number of external dependencies and lines of code per implementation.

Language (abbrv.)	External dependencies	Lines of code per implementation		
		Sequential	Local Parallel	Distributed
C++ (`cpp`)	Eigen	$\simeq 25$	$\simeq 25$	$\simeq 120$
Python (`py`)	Numpy, Scipy	$\simeq 10$	$\simeq 10$	$\simeq 100$
Julia (`jl`)	MatrixMarket.jl	$\simeq 25$	$\simeq 25$	$\simeq 100$
DaphneDSL (`daph`)	$\emptyset$	$\simeq$ **10**	$\simeq$ **10**	$\simeq$ **10**

TABLE 1 – Characteristics of the implementation of the *Connected Components* algorithm in the considered languages for sequential, parallel on a single node, and distributed node executions.

3.3. Implementation : Sequential Version

We used the same CSR (Compressed Sparse Row) representation in memory for the sparse matrices to allow a fair comparison between all implementations. Using CSR had an impact on the implementations of CC.

The CC algorithm was implemented in DaphneDSL (see Listing 1 in Appendix) and Python in a sparse matrix format agnostic fashion. However, for Julia and C++, we encountered implementation challenges. For Julia, the column broadcast over a sparse matrix was silently converting the sparse matrix to a dense matrix to perform the broadcast, consuming all the memory of the machine. To solve this issue, we had to implement the broadcast operation by hand. In the case of C++, the API of Eigen does not allow the users to perform column-wise operations on a CSR matrix. Hence, we also had to implement the broadcast by hand. These custom-made broadcast functions reflect an increased number of lines of code for Julia and C++ in Table 1.

3.4. Implementation : Local Parallel Version

For C++, Julia, and Python, the libraries performing linear algebra, support multithreaded parallel execution as they rely on the BLAS implementations which use OpenMP. Hence, setting the environment variable `$OMP_NUM_THREADS` is sufficient to enable node-level parallelism.

One can exploit node-level parallelism with DAPHNE in two ways : either by deploying a local instance of DaphneSched using several threads (1 worker with N threads), or by using the distributed runtime of DAPHNE and deploying several local DaphneSched instances with a single thread each (N workers with 1 thread).

3.5. Implementation : Distributed Parallel Version

Most of the development effort for the MPI implementations in C++, Python, and Julia lies in the decomposition of the sparse matrix across computing nodes. We first *statically* decompose the matrix along rows and send chunks of rows to the different ranks. Then, each rank performs the local computation of CC, and then synchronizes with a `MPI_Allreduce` call, with a user-defined reduce function, at the end of each iteration of the algorithm.

4. Experimental Evaluation

We report here the experiments conducted to study the performance of the various implementations of CC. They were carried out on machines with two Intel Broadwell E5-2640v4 CPUs (one in each of the two sockets of the machine), each with 10 cores and 64GB of RAM [29]. Each experiment is repeated 3 times, and the average execution time of the main loop of CC is reported (*i.e.*, not including reading and scattering of the matrix). These experiments follow a factorial design, described in the Appendix in Section C. All data and scripts describing this work are available on Zenodo [19].

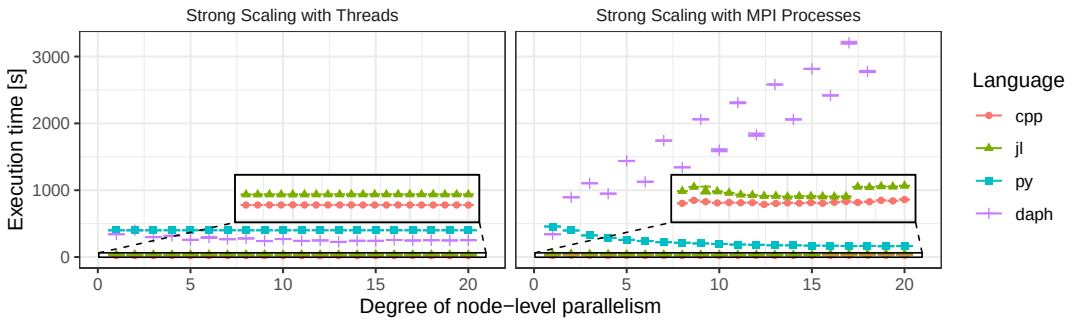


FIGURE 2 – Strong scaling on a *single node* (20 total cores) with threads (left) and MPI processes (right) for CC with `wikipedia-20070206` as input. The error bars represent 95% confidence intervals. DAPHNE used a `CENTRALIZED` queue with the `STATIC` scheduling technique.

4.1. Strong Scaling with Local Parallelism

We study the strong scaling performance of exploiting node-level parallelism using multiple threads (OpenMP or DAPHNE) or multiple processes (MPI). The results in Figure 2 (left) indicate that apart from DAPHNE, there is no strong scaling benefit from using multiple threads, despite the fact the linear algebra libraries are parallelized with OpenMP. With multiple MPI processes, Figure 2 (right), the Python CC version shows clear scalability improvements, while DAPHNE's performance degrades with a larger number of MPI ranks. We explain this performance degradation by the memory cost of the local DAPHNE runtime and the increased number of messages handled by the coordinator. This is also the reason why there are no data points for 19 and 20 MPI processes with DAPHNE, as the local node runs out of memory. We believe that the fluctuating DAPHNE performance for odd and even MPI rank counts is due to the rank placement pattern of Slurm across the NUMA domains.

4.2. Seamless Scaling with Distributed DaphneSched

We also studied the seamless scaling of CC with DAPHNE across four nodes, exploring various scheduling options and queue layouts with the `SEQPRI` victim selection, as preliminary experiments that show it frequently achieves higher performance than other victim selections. The results in Figure 3, indicate that for the `amazon0601` matrix, the queue layout and victim selection strategies have little impact on performance. However, for larger and sparser matrices, the importance of the choice of the scheduling scheme is emphasized. For example, for a `CENTRALIZED` queue, the default configuration (`STATIC`) is twice as slow as the highest performing scheme in the case of `ljournal-2008` (`TFSS`). We observe that the highest performing queue layout is `CENTRALIZED`, and that the highest performing scheduling scheme for one queue layout is not necessarily the same for another queue layout. This highlights the importance of efficient scheduling schemes to be available to users.

4.3. Strong Scaling with Distributed Parallelism

Figure 4 shows the strong scaling performance of the implementations of the CC algorithm on various node counts. Based on insights from Section 4.1, we used 20 MPI processes with 1 thread each on *each node* for C++, Julia, and Python. For DAPHNE, we used 1 MPI process per node with 20 threads per MPI process. We evaluated the default configuration of DAPHNE (`STATIC` with `CENTRALIZED` queue), and DAPHNE with `AUTO` scheduling on a `CENTRALIZED` queue, to highlight the potential gain to use a different scheduling technique. We observe that for small matrices (`amazon0601`), DAPHNE is outperformed by all other version, with almost no scaling beyond 4 nodes. However, for larger and sparser matrices,

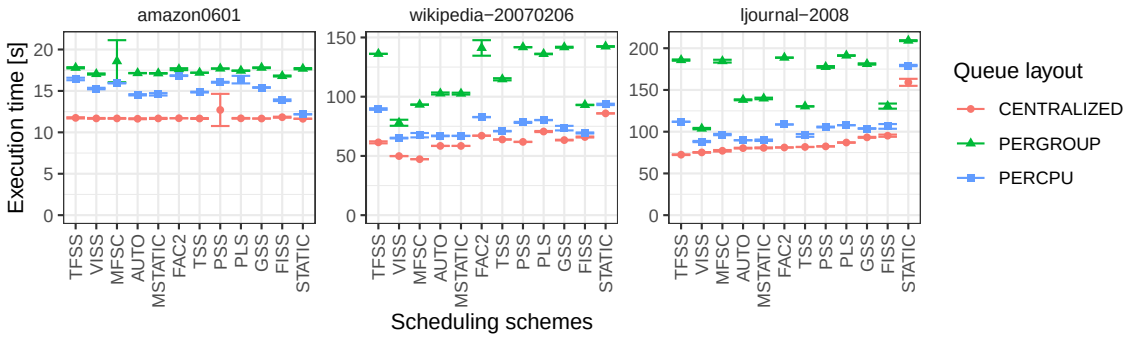


FIGURE 3 – Average execution time with 95% confidence intervals for CC with DAPHNE, for each scheduling scheme and queue layout, with the `SEQPRI` victim selection, executed on four nodes and one MPI process per node. Total degree of parallelism : (4 - 1) x 20 = 60 workers.

DAPHNE performs much better and even outperforms Python and Julia. C++ clearly outperforms all implementations.

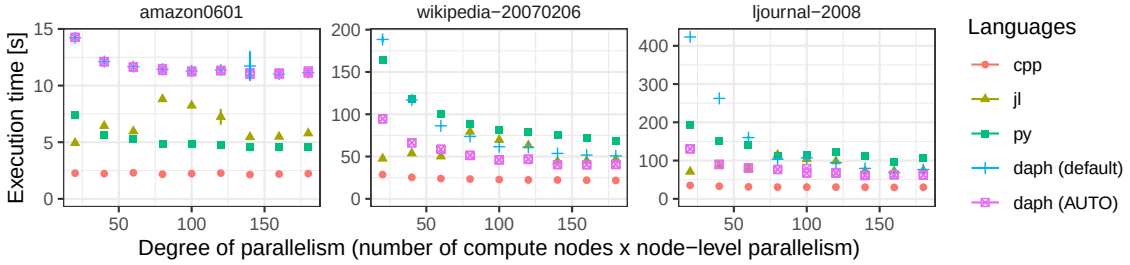


FIGURE 4 – Strong scaling from 1 to 9 compute nodes. Inside a node, the work is parallelized with MPI for C++, Julia and Python, and with threads for DAPHNE. We execute DAPHNE with its default configuration (`CENTRALIZED + STATIC`) and with the `AUTO` scheduling technique and a `CENTRALIZED` queue to show the importance of scheduling on performance.

5. Conclusion

We presented the ongoing work on seamlessly scaling applications with DAPHNE – an infrastructure for optimizing data analysis pipelines. After exposing the different components of the distributed DAPHNE runtime, we compared a DaphneDSL implementation for the *Connected Components* algorithm against Python, Julia, and C++ implementations along several dimensions : external dependencies, effort to adapt the code for parallel and distributed executions, and performance.

The results demonstrate the value of DaphneSched's various scheduling options. Although DAPHNE does not yield the highest performance in a distributed setting compared to C++, it has the benefit of requiring *no effort* for seamlessly scaling applications across nodes with MPI and in certain circumstances outperforms Python and Julia implementations. Hence, DAPHNE positions itself as a trade-off between "Ease of Use" and "Scaling Performance".

Ongoing work in DAPHNE includes a more comprehensive comparison of several applications and systems, the introduction of communication between the coordinator and the DAPHNE workers to improve resource utilization, and dynamic work partitioning by the coordinator.

Acknowledgments

This research was funded, in whole or in part, by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 957407 as DAPHNE.

Références

[1] S. BEAMER, K. ASANOVIĆ et D. PATTERSON. The GAP benchmark suite. *arXiv preprint arXiv :1508.03619*, 2015.

[2] T. BEN-NUN et T. HOEFLER. Demystifying parallel and distributed deep learning : An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4) :1-43, 2019.

[3] J. BEZANSON, A. EDELMAN, S. KARPINSKI et V. B. SHAH. Julia : A fresh approach to numerical computing. *SIAM review*, 59(1) :65-98, 2017.

[4] J. BEZANSON, S. KARPINSKI, V. B. SHAH et A. EDELMAN. Julia : A Fast Dynamic Language for Technical Computing. *arXiv preprint arXiv :1209.5145*, 2012.

[5] P. BOLDI, M. ROSA, M. SANTINI et S. VIGNA. Layered Label Propagation : A MultiResolution Coordinate-Free Ordering for Compressing Social Networks. In S. SRINIVASAN, K. RAMAMRITHAM, A. KUMAR, M. P. RAVINDRA, E. BERTINO et R. KUMAR, éditeurs, *Proceedings of the 20th international conference on World Wide Web*, pages 587-596. ACM Press, 2011.

[6] P. BOLDI et S. VIGNA. The WebGraph Framework I : Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*, pages 595-601, Manhattan, USA. ACM Press, 2004.

[7] S. BYRNE, L. C. WILCOX et V. CHURAVY. MPI.jl : Julia bindings for the Message Passing Interface. *Proceedings of the JuliaCon Conferences*, 1(1) :68, 2021. DOI : 10.21105/jcon.00068. URL : https://doi.org/10.21105/jcon.00068.

[8] L. DALCIN et Y.-L. L. FANG. mpi4py : Status update after 12 years of development. *Computing in Science & Engineering*, 23(4) :47-54, 2021.

[9] P. DAMME, M. BIRKENBACH, C. BITSAKOS, M. BOEHM, P. BONNET, F. CIORBA, M. DOKTER, P. DOWGIALLO, A. ELELIEMY, C. FAERBER et al. Daphne : An open and extensible system infrastructure for integrated data analysis pipelines. In *Conference on Innovative Data Systems Research*, 2022.

[10] DAPHNE. Design of integration HW accelerators. 2022. URL : https://daphne-eu.eu/wp-content/uploads/2022/05/DAPHNE_Deliverable_7.1.pdf.

[11] DAPHNE. Docker Image. Mai 2024. URL : https://hub.docker.com/r/daphneeu/daphne.

[12] DAPHNE. DSL Runtime Design. 2021. URL : https://daphne-eu.eu/wp-content/uploads/2021/11/Deliverable-4.1-fin.pdf.

[13] T. A. DAVIS et Y. HU. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1) :1-25, 2011.

[14] E. DOLSTRA, M. DE JONGE, E. VISSER et al. Nix : A Safe and Policy-Free System for Software Deployment. In *LISA*, tome 4, pages 79-92, 2004.

[15] A. ELELIEMY et F. M. CIORBA. DaphneSched : A Scheduler for Integrated Data Analysis Pipelines. In *2023 22nd International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 53-60. IEEE, 2023.

[16] A. ELELIEMY et F. M. CIORBA. A distributed chunk calculation approach for self-scheduling of parallel applications on distributed-memory systems. *Journal of Computational Science*, 51 :101284, 2021. ISSN : 1877-7503. DOI : https://doi.org/10.1016/j.jocs.2020.101284. URL : https://www.sciencedirect.com/science/article/pii/S1877750320305792.

[17] GRPC. Website. 25 jan. 2021. URL : https://grpc.io/.

[18] G. GUENNEBAUD, B. JACOB et al. Eigen. *URl : http ://eigen. tuxfamily. org*, 3(1), 2010.

[19] Q. GUILLOTEAU, J. H. MÜLLER KORNDÖRFER et F. M. CIORBA. Data and analysis scripts for the submission "Seamlessly Scaling Applications with DAPHNE". Zenodo, mai 2024. DOI : 10.5281/zenodo.11126714. URL : https://doi.org/10.5281/zenodo.11126714.

[20] C. R. HARRIS, K. J. MILLMAN, S. J. VAN DER WALT, R. GOMMERS, P. VIRTANEN, D. COURNAPEAU, E. WIESER, J. TAYLOR, S. BERG, N. J. SMITH et al. Array Programming With NumPy. *Nature*, 585(7825) :357-362, 2020.

[21] [Logiciel] JULIASPARSE, 2024. URL : https://github.com/JuliaSparse/MatrixMarket.jl, SWHID : ⟨swh:1:dir:3e3499c94ae2294fb210842c38d215c001368d9e;origin=https://github.com/JuliaSparse/MatrixMarket.jl⟩.

[22] J. KÖSTER et S. RAHMANN. Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19) :2520-2522, 2012.

[23] C. LATTNER, M. AMINI, U. BONDHUGULA, A. COHEN, A. DAVIS, J. PIENAAR, R. RIDDLE, T. SHPEISMAN, N. VASILACHE et O. ZINENKO. MLIR : Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2-14. IEEE, 2021.

[24] J. LESKOVEC et A. KREVL. SNAP Datasets : Stanford Large Network Dataset Collection. http://snap.stanford.edu/data, juin 2014.

[25] MESSAGE PASSING INTERFACE FORUM. *MPI : A Message-Passing Interface Standard Version 4.1*. Nov. 2023. URL : https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[26] A. MOHAMMED, J. H. MÜLLER KORNDÖRFER, A. ELELIEMY et F. M. CIORBA. Automated Scheduling Algorithm Selection and Chunk Parameter Calculation in OpenMP. *IEEE Transactions on Parallel and Distributed Systems*, 33 :12, 2022. DOI : 10.1109/TPDS.2022.3189270.

[27] J. H. MÜLLER KORNDÖRDER, A. ELELIEMY, A. MOHAMMED et F. M. CIORBA. LB4OMP : A Dynamic Load Balancing Library for Multithreaded Applications. *IEEE Transactions on Parallel and Distributed Systems*, 33(4) :830-841, 2022. DOI : 10.1109/TPDS.2021.3107775.

[28] OPENAI. Scaling Kubernetes to 7,500 nodes. 25 jan. 2021. URL : https://openai.com/index/scaling-kubernetes-to-7500-nodes.

[29] UNIVERSITY OF BASEL. MiniHPC. Mai 2024. URL : https://hpc.dmi.unibas.ch/research/minihpc/.

[30] P. VIRTANEN, R. GOMMERS, T. E. OLIPHANT, M. HABERLAND, T. REDDY, D. COURNAPEAU, E. BUROVSKI, P. PETERSON, W. WECKESSER, J. BRIGHT et al. SciPy 1.0 : fundamental algorithms for scientific computing in Python. *Nature methods*, 17(3) :261-272, 2020.

[31] A. VONTZALIDIS, S. PSOMADAKIS, C. BITSAKOS, M. DOKTER, K. INNEREBNER, P. DAMME, M. BOEHM, F. CIORBA, A. ELELIEMY, V. KARAKOSTAS et al. DAPHNE Runtime : Harnessing Parallelism for Integrated Data Analysis Pipelines. In *European Conference on Parallel Processing*, pages 242-246. Springer, 2023.

A. Considered Matrices

The characteristics of the three matrices considered in Section 4 are presented in Table 2 in terms of number of rows and columns, non-zero elements and degree of density (calculated as the percentage of non-zero elements).

Matrix	Size	Non-zero elements	Density (%)
`amazon0601` [24]	403394×403394	3387388	2.08×10^{-3}
`wikipedia-20070206`	3566907×3566907	45030389	3.54×10^{-4}
`ljournal-2008` [6, 5]	5363260×5363260	79023142	2.75×10^{-4}

TABLE 2 – Characteristics of the input matrices considered in this work.

B. *Connected Components* Implementation with DaphneDSL

Listing 1 shows the implementation of the *Connected Components* algorithm in DaphneDSL. Note that the *Connected Components* algorithm typically completes upon reaching convergence of the vector `c` Listing 1. To allow a fair comparison of its various implementations, we fixed its number of iterations to 100 for all languages versions.

```
G = readMatrix($f); // read sparse matrix from CLI argument
maxi = 100; // maximum number of iterations
start = now();

c = seq(1.0, as.f64(nrow(G)), 1.0); // initialization

for(iter in 1:maxi) {
    c = max(aggMax(G * t(c), 0), c);
}

end=now();
print((end-start) / 1000000000.0);
```

Listing 1 – Implementation of the *Connected Components* algorithm in DapneDSL

C. Design of Experiments

For the results presented in Figures 2, 3, and 4, we employ a factorial design of experiments, described in Tables 3, 4, and 5, respectively. In total, we conducted 1'209 experiments.
The software environment is controlled by Nix [14] and the Docker image built by the DAPHNE consortium [11]. The workflow of the experiments is managed by Snakemake [22].

Factor	Value	Details
Application	1	*Connected Components*
Input data	1	`wikipedia-20070206`
Languages	4	C++, Julia, Python, and DaphneDSL
Types of Scaling	2	with threads or with MPI processes
Parallelism configuration	20	from 1 to 20 threads or MPI processes
Number of nodes	1	1 node
Experiment repetitions	3	
Total number of experiments	480	

TABLE 3 – Factorial design of experiments for the results in Figure 2

Factor	Value	Details
Application	1	*Connected Components*
Input data	3	`amazon0601`, `wikipedia-20070206`, `ljournal-2008`
Languages	1	DaphneDSL
Scheduling techniques	12	`STATIC`, `GSS`, `TSS`, `FAC2`, `TFSS`, `FISS`, `VISS`, `PLS`, `MSTATIC`, `MFSC`, `PSS`, and `AUTO`
Queue layouts	3	`CENTRALIZED`, `PERGROUP`, `PERCPU`
Victim selection	1	`SEQPRI`
Number of nodes	1	4 nodes
Experiment repetitions	3	
Total number of experiments	324	

TABLE 4 – Factorial design of experiments for the results in Figure 3

Factor	Value	Details
Application	1	*Connected Components*
Input data	3	`amazon0601`, `wikipedia-20070206`, `ljournal-2008`
Languages	5	C++, Julia, Python, DaphneDSL with the default configuration (`CENTRALIZED` queue and `STATIC` scheduling technique), DaphneDSL using a `CENTRALIZED` queue and the `AUTO` scheduling technique
Number of nodes	9	From 1 to 9 nodes
Experiment repetitions	3	
Total number of experiments	405	

TABLE 5 – Factorial design of experiments for the results in Figure 4