A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters

Vinicius Garcia Pinto, Lucas Mello Schnorr, Luka Stanisic, Arnaud Legrand, Samuel Thibault, Vincent Danjean

To cite this version:

Vinicius Garcia Pinto, Lucas Mello Schnorr, Luka Stanisic, Arnaud Legrand, Samuel Thibault, et al.. A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters. 2017. hal-01616632v1

HAL Id: hal-01616632

https://inria.hal.science/hal-01616632v1

Preprint submitted on 19 Oct 2017 (v1), last revised 17 Jul 2018 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Received TBD; | Revised TBD; | Accepted TBD

DOI: xxx/xxxx

RESEARCH ARTICLE

A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters†

Vinícius Garcia Pinto[1,2] | Lucas Mello Schnorr*[1,2] | Luka Stanisic[4] | Arnaud Legrand[2] | Samuel Thibault[3] | Vincent Danjean[2]

[1] Institute of Informatics, Federal University of Rio Grande do Sul – UFRGS, Porto Alegre, Brazil
[2] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France
[3] Inria Bordeaux Sud-Ouest Bordeaux, France
[4] Max Planck Computing and Data Facility, Garching, Germany

Correspondence
Lucas Mello Schnorr, Email: schnorr@inf.ufrgs.br

Summary

Programming paradigms in High-Performance Computing have been shifting towards task-based models which are capable of adapting readily to heterogeneous and scalable supercomputers. The performance of task-based application heavily depends on the runtime scheduling heuristics and on its ability to exploit computing and communication resources. Unfortunately, the traditional performance analysis strategies are unfit to fully understand task-based runtime systems and applications: they expect a regular behavior with communication and computation phases, while task-based applications demonstrate no clear phases. Moreover, the finer granularity of task-based applications typically induces a stochastic behavior that leads to irregular structures that are difficult to analyze. This paper details a flexible framework combining visualization panels to understand and pinpoint performance problems incurred by bad scheduling decisions in task-based applications. Three case-studies using StarPU-MPI, a task-based multi-node runtime system, are detailed to show how our framework is used to study the performance of the well-known Cholesky factorization. Performance improvements include a better task partitioning among the multi-(GPU,core) to get closer to theoretical lower bounds, improved MPI pipelining in multi-(node,core,GPU) to reduce the slow start, and changes in the runtime system to increase MPI bandwidth, with gains of up to 13% in the total makespan.

KEYWORDS:
High-Performance Computing, Heterogeneous platforms, Task-based applications, Trace Visualization, Cholesky

1 | INTRODUCTION

High-Performance Computing (HPC) landscape have experienced a paradigm shift in the last years. The stagnation in the processor frequency has led the adoption of other ways to fulfill the ever-growing need for computation power of HPC applications. A common HPC platform formerly composed of homogeneous nodes now is composed of multicore computing nodes enhanced with multiple accelerator devices, e.g., 90 of the fastest supercomputers listed in the June 2017 Top500 list are using GPUs and/or Xeon Phi coprocessors to increase their computing power (1).

†See the acknowledgement section at the end for funding information.

This paradigm shift in the hardware has exposed the limitations of traditional tools for programming and analyzing applications running on HPC platforms. Efficiently programming such machines achieving portable and scalable performance has become extremely challenging. The use of explicit programming models demands a huge effort to develop and maintain the application and results in a code that is tightly coupled to the target hardware. Such programming model is unfeasible in the hybrid scenario where the accelarator's technology is changing fast, e.g., Cell Broadband Engine Architecture (2006), Graphics processing unit (2007), XeonPhi (2013).

The pressure on parallel programming tools has contributed to the popularization of the task-based programming model. While the traditional parallel programming paradigm relies on low-level abstractions like threads and explicit synchronizations, the task-based model describes the application in terms of dependent sequential (or parallel) tasks. Explicit synchronizations are replaced by tasks dependencies that can be, in several cases, inferred automatically from data access. The task-based model is implemented by several programming models: OpenMP 4 (2), OmpSs (3), PaRSEC (4), StarPU (5), etc. Despite the growing availability of tools to program and execute task-based applications on hybrid platforms, there are few analysis tools with a focus on such scenario. One of the main reasons is that the burden of getting the maximum performance from the machine shifts from the application to the runtime developer, who implements a given scheduling heuristic. Usually, tools that are capable of giving insights to developers, such as Paraver (6), Vite (7), and others (8, 9), are unsuited to scheduling specialists because assumptions and performance bottlenecks are different from classical parallel programs.

In this article, we explain how we designed a flexible and versatile framework that enables a better understanding and easy identification of subtle performance problems due to bad scheduling choices in task-based parallel applications. Most of the problems would go unnoticed and misunderstood with classical trace visualization approaches. We built the framework [1] on top of modern data analytics tools, combining the R programming language (and in particular the ggplot2 library) and org-mode (10). The framework comes at a very low development cost when compared to a traditional and monolithic performance visualization tool. The designed views depict many different facets of the program behavior, enabling the correlation of already available performance metrics with those derived from the traces. In summary, this article presents a flexible, extensible, and versatile visual performance analysis framework whose main strengths compared to other tools are the following: (a) combining the intuitiveness of visual trace exploration with the expressiveness of statistical queries (Section 4); (b) easily creating and synchronizing application-specific views that allow to better understand how the scheduling unfolds (Section 4.1.1); (c) automatically filtering dependencies to quickly identify which scheduling opportunities may have been missed (Section 4.1.1); (d) a two-phase implementation as a workflow where the first part can be run on the data server, for pre-processing, and the second part on the analyst laptop, for visualization (Section 4.3).

We demonstrate the effectiveness of our visualization approach by analyzing traces from the dense linear algebra Cholesky factorization of the Chameleon/MORSE package (11), implemented using the StarPU task-based runtime (5) and its MPI extension (12). Our experiments rely on two hybrid multi-core/multi-GPU/multi-node clusters and reveal many interesting behavior that lead us to pinpoint resource usage mistakes, to compare StarPU schedulers (DMDA, DMDAS and Work Stealing), and to conduct an extensive analysis of StarPU-MPI performance. The use of our framework have enable us to propose many optimizations to StarPU-MPI, ultimately leading to the following performance gains: (a) constraining certain task types to particular resource types may help reducing idleness in multi-GPU scenarios; (b) fixing the lack of message pipelining with MPI by increasing the eager mode limit of the OpenMPI implementation, hence reducing the slow start of the application; and (c) detecting that too many MPI concurrent operations are harmful when reaching the maximum of parallelism. Regarding the last item, we have proposed a two-fold solution including communication priorities and a limit on the number of concurrent MPI requests from StarPU. Combined, all these contributions bring several improvements that are already implemented in the main trunk of StarPU-MPI code, for the benefit of other parallel applications.

Section 2 provides a background on task-based runtimes for hybrid platforms, the tiled Cholesky algorithm used as case study, and the StarPU runtime and its MPI extension. Section 3 presents related work of traditional BSP-based and task-oriented trace visualization, motivating our own study. Section 4 presents our visualization framework and workflow, detailing all the panels, a discussion about scalability, the implementation and performance, as well as some limitations of our strategy. In Section 5, we detail three case studies demonstrating the effectiveness of our visualization strategy to debug the tiled task-based Cholesky application, and the MPI extension of the StarPU runtime. Section 6 gives a summary of results and future work.

[1] Code available at https://gitlab.in2p3.fr/schnorr/ccpe2017/

2 | BACKGROUND AND EXPERIMENTAL CONTEXT

2.1 | Background

Traditional HPC applications have been designed following the bulk-synchronous parallel (BSP) paradigm. In this model, the application execution has well-defined phases: computation, communication and barrier. The BSP design has been used for a long time in homogeneous platforms with identical nodes connected by a fast and stable interconnection. However, current HPC platforms rely on hybrid nodes where accelerator devices are attached to multicore processors to increase the computational power of the system. In this scenario, the use of explicit programming models where one should indicate where and when each computation should be done, as done in the BSP paradigm, becomes impractical as it requires to tightly couple the implementation to the hardware configuration to achieve good performances. Although this strategy might lead towards the maximal achievable performance, it suffers from bad performance portability, it is sensitive to variability and it is really hard to develop and maintain.

Task-based programming tools have emerged to address the heterogeneity of state of art HPC platforms. In this model, the application is designed in terms of hardware-independent tasks and their dependencies. Task-based tools rely on dynamic runtime systems to efficiently exploit the multilevel parallelism of the platform. Once the programmer describes the application in terms of abstract tasks and their dependencies, the runtime system is responsible for platform-related activities such as task scheduling, data transfers, and dependencies management. To perform these activities, the runtime system infers platform and code characteristics such as provided task implementations, available processing units (CPU cores, GPUs devices), estimated task duration, data locality and interconnection bandwidth. From these details, the runtime can use appropriate scheduling heuristics and perform optimizations (e.g., anticipating an overlap of data transfers) in order to achieve a better performance.

Several successful tools have been developed in the recent years. Initially, they focused on specific applications. MAGMA (13), for instance, supports the execution of linear algebra applications in a combination of multi-core with GPUs. OmpSs (3) provides an extension to OpenMP tasks through new directives that allow supporting multi-core systems enhanced with GPUs. PaRSEC (14) is a generic framework for architecture-aware scheduling of tasks on many-core heterogeneous clusters. StarPU (5) is designed to exploit hybrid architectures and offers an MPI-based extension (12) to exploit multiple nodes. Despite the initial focus on dense linear algebra applications, currently, these runtime systems are being used in many other domains, such as FEM applications (15), seismic wave modeling (16), sparse linear algebra (17, 18), aerodynamic simulations (19) and climate modeling (20).

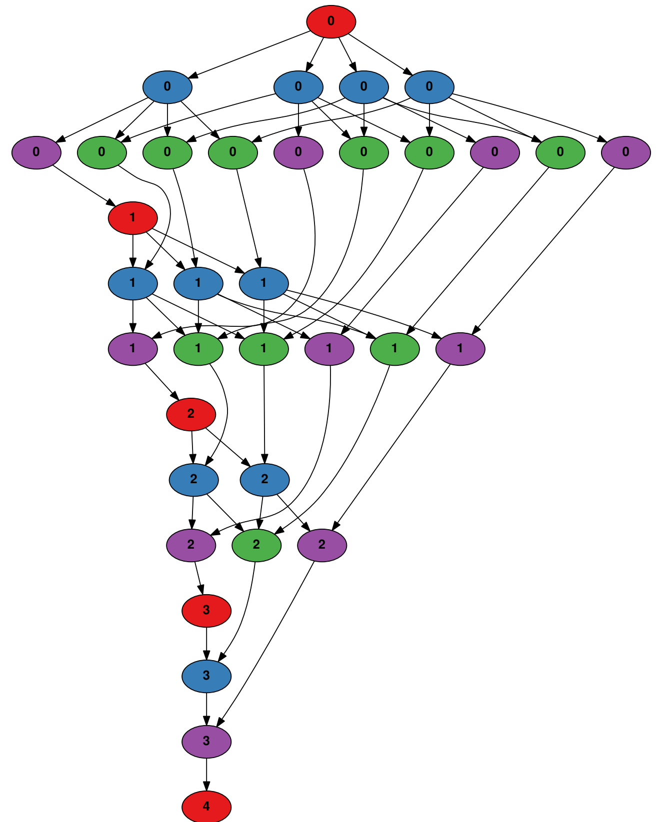
2.2 | The Tiled Task-based Cholesky decomposition

In the context of task-based applications, the overall performance is intrinsically related to the efficiency of the runtime system. For that reason, focusing our analysis on the runtime system can help us to identify important issues and mistakes that impact the overall performance of the application. However, to study the runtime system performance we should rely on a representative and already well-optimized application. Using a non-optimized application can hide runtime system performance issues while a non-representative one could lead us to problems and mistakes that have no significant influence on the performance of commonly-executed applications. In this work, we use a Cholesky decomposition as our case study application. This factorization is one of the most common linear algebra operations and is used by many scientific applications. In order to improve the reproducibility and the stability of our tests, we adopt the tiled Cholesky implementation provided by the Chameleon/MORSE package (11). This implementation is built on top of the StarPU runtime system (5) and compiled with standard BLAS (CPU) and CUBLAS (GPU) libraries.

Figure 1 shows a simplified version of this application. The lines with calls to `dpotrf`, `dtrsm`, `dsyrk` and `dgemm` (Figure 1a) represent the creation of StarPU tasks with double-precision implementations for CPUs and GPUs (except the `dpotrf` that has only the CPU version). The underlined labels RW and R indicate the access mode of the subsequent matrix block. From these access mode hints, the runtime can infer the dependencies and then build the Directed Acyclic Graph (DAG) of tasks. Figure 1b shows the corresponding DAG for a 5×5 matrix. In each iteration k of the outer loop, one `dpotrf` task enables the execution of $N - k - 1$ `dtrsm`, then $N - k - 1$ `dsyrk` tasks, followed by $\approx (N-k)^2/2$ `dgemm` tasks. From the dependencies, we can conclude that several iterations can be executed simultaneously and that the number of repetitions in the internal loops decreases at the same time as k increases. Finally, the execution time of a task highly depends on its type (`dpotrf`, `dtrsm`, `dsyrk`, and `dgemm`) and the target resource (CPU or GPU). Note that the color scheme used in this Figure to represent the task types is respected in all the following graphics of this paper.

```
for (k = 0; k < N; k++) {
   DPOTRF(RW, A[k][k]);
   for (i = k+1; i < N; i++)
     DTRSM(RW, A[i][k], R, A[k][k]);
   for (i = k+1; i < N; i++) {
     DSYRK(RW, A[i][i], R, A[i][k]);
     for (j = k+1; j < i; j++)
        DGEMM(RW, A[i][j], R, A[i][k],
                          R, A[j][k]);
   }
}
```

The Cholesky Algorithm.



Corresponding DAG for $N = 5$.

FIGURE 1 The Cholesky code and its DAG (for $N = 5$).

2.3 | The StarPU Runtime and the MPI extension for multi-node platforms

StarPU (5) is a runtime system for task-based programming on hybrid architectures. The runtime was initially designed to handle single-node hybrid platforms composed of multicore processors (CPUs) and accelerators (GPUs, Intel Xeon Phi). To efficiently exploit the parallelism of the platform, StarPU relies on multiple implementations of the same tasks e.g., with CPU and/or GPU versions. The runtime scheduler decides on-the-fly where to execute the tasks considering the available processing resources, their type, the current locations of data, and the provided task implementations. StarPU offers several task scheduling policies. The DMDA/DMDAS policies are based on precalibrated performance models, while the WS/LWS uses a work stealing design, stealing tasks from the most loaded worker (original) or the most loaded neighbor worker (locality version). The PRIO policy only relies on priority hints specified by the application programmer.

The **DMDA** (*Deque Model Data Aware*) and **DMDAS** (*Deque Model Data Aware Sorted*) algorithms are members of a family of StarPU schedulers that take the predicted task duration and data transfer duration into account when performing the task scheduling. These strategies are based on *list scheduling*, i.e., every time a resource is idle, if a task is ready, it will be scheduled on this particular resource. Such a scheduler therefore never leaves a resource idle on purpose, which ensures the well-known $(2-1/p)$ competitive ratio for homogeneous machines (21). Deciding which ready task to select has a major influence in practice and the classical heuristic consists in prioritizing tasks based on the critical path. However, the critical path notion is dynamic and obtaining a proper estimation can be quite challenging. With heterogeneous computing resources, such prioritization is generally done with variants of the HEFT (Heterogeneous Earliest Finish Time) strategy (22). The DMDA and DMDAS algorithms are greedy heuristics that schedule tasks in the order they become available, taking into account the predicted task duration, the estimated data transfer duration between CPUs and GPUs and the relative performance of resources on each computation kernel when making its decision. The DMDAS algorithm improves DMDA decisions by sorting tasks on per-worker lists by number of data-slices already transferred and by priority, which can be expensive when the number of tasks is large. It is therefore rather close to the original HEFT algorithm by respecting priorities and taking past scheduling decisions into account.

The **WS** (*Work Stealing*) and **LWS** (*Locality Work Stealing*) algorithms use one list per worker; new tasks are kept local by default. When a worker is idle, in the WS policy, it steals tasks from the most loaded worker. The LWS policy, on the other hand, imposes that the worker must steal first from the most loaded neighbor worker. This victim's choice differs from the classical work stealing algorithm (23) where the victim is chosen using a random strategy.

The **PRIO** algorithm uses a mere centralized list that is shared by all the workers. This list keeps the tasks sorted respecting the *priorities* specified by the application programmer.

The StarPU-MPI (12) extension provides additional support to handle multi-node architectures, adopting the MPI specification as a mean of access to the network. The main characteristic of the extension is that it has one independent StarPU scheduler per node. For example, if an experiment is configured with DMDA, this scheduler will be used in each of the nodes to process a part of the whole application DAG. Existing dependencies among nodes are satisfied through MPI send/receive point-to-point operations, and managed like any other task scheduled by the runtime. StarPU-MPI has one specific thread to handle MPI communications per node. Since the domain decomposition is static, in the beginning of the application, the MPI threads call asynchronous send/receive operations to satisfy all task dependencies of the node it runs on. After all operations have been posted, the thread keeps testing all requests to verify when they arrive, and then release the execution of tasks.

The classical Two-dimensional Block-Cyclic Distribution (24) has been previously modified to support multi-node runs with static decomposition under the auspices of StarPU-MPI. The decomposition depends on the P × Q parameter and the number of MPI nodes, governing how the input matrix is partitioned among nodes in a per-tile basis. The value of P can range from one to the number of nodes. Figure 2 depicts the four possible situations cases for a Cholesky factorization with eight nodes and a matrix with 16×16 tiles: the data decomposition shown in the left facet is obtained when P=1 × Q=8 and leads to a row based distribution of tiles (one color per node); for P=2 × Q=4 and P=4 × Q=2, shown in the center left and right facets, the data distribution is interleaved; finally, when P=8 × Q=1, data distribution is by column as shown by the right facet. In an ideal scenario, for a given number of nodes, the value of P should be defined so as to minimize the communication perimeter of each node as it is related to the total volume of communications.

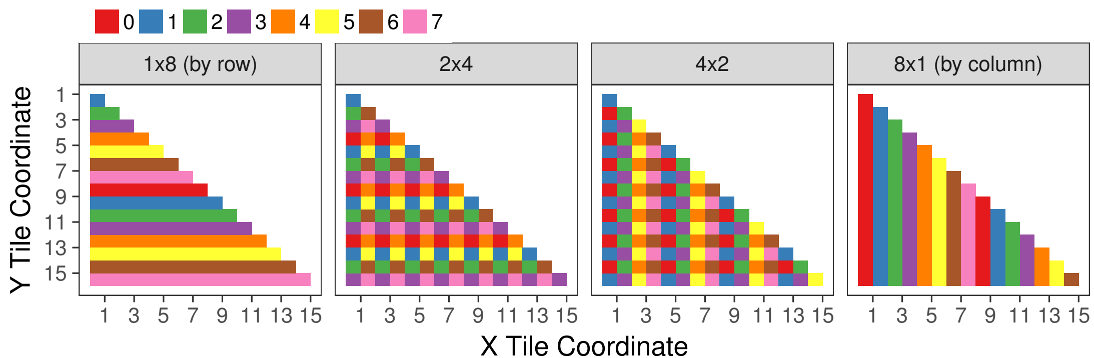


FIGURE 2 Different static partitioning schemes for `dtrsm` tasks as dictated by the P parameter when eight nodes are used to run Cholesky: P=1 (left, by row), P=2 (center left), P=4 (center right), and P=8 (right, by column).

3 | RELATED WORK AND MOTIVATION

There are very few established tools to conduct a proper task-aware analysis. This contradicts the latest trend towards runtime systems for task-based applications, with a plethora of these (see Section 2). As a consequence, developers rely on BSP-based tools, seeking unexpected heterogeneity where homogeneous behavior is common. These classic visualization techniques are unsuited to the performance analysis of task-based applications because heterogeneity and unstructured execution is the expected common behavior. We detail trace visualization strategies for both BSP and DAG-based applications highlighting their differences (Sections 3.1 and 3.2), and motivate our work with the design challenges (Section 3.3) of novel visualization techniques for DAG applications, as well as the typical questions raised during the analysis process.

3.1 | Traditional BSP-based Visualization

Many tools exist to visualize traces from BSP-based applications, where regularity among resources is common and clear application phases can be visually and automatically detected. Most of these tools focus on message-passing applications relying on the widely-known MPI interface (25). The most common technique is based on space/time plots, which get inspiration from the traditional Gantt charts (26), where computational resources or application entities (process, threads) are arranged vertically, sometimes hierarchically organized, while the application states (functions, actions) are laid out horizontally along time. Colors are extensively used to depict different thread states e.g., MPI operation. Interaction between application components are depicted as arrows whose width may correlate with the the amount of transferred data. Such technique has been implemented multiples times with different technologies to improve human perception and scalability.

Yet, the race towards Exascale computing brings thousands to millions of cores into play, turning slow the rendering of space/-time views when there are millions of states to be shown in a single screen. To mitigate this problem, Vite (7) uses an OpenGL canvas with GPU acceleration to improve rendering performance. Since Vite relies on the semantic-free Paje language (27) as trace input, it can depict virtually any kind of traces. Paraver (6) also tackles the scalability issue by implementing trace aggregation before the visualization. So, instead of sending all trace data to the rendering driver, it decides (based on user configuration) which element to draw in a specific location of the screen. Vampir (9) is a closed-source visualization tool with multiple views for MPI-based OTF2 trace files. It is considered to be more scalable than similar tools since it can leverage the nodes of a cluster to increase the data bandwidth towards the client node that renders the space/time view. In general, all these tools focus on interactivity, enabling one to filter, zoom, interact, query data with a GUI. More recently, some of these tools have been adapted to handle application traces collected in hybrid machines (28).

Space/time views are very useful to depict per-thread application states along time. However, the interaction among threads, represented by arrows from source to destination, usually makes the representation hard to understand. The problem is that asynchronous communication and priority communication queues turn the visualization into a clutter of graphical lines, without any kind of regularity. This problem has for example been addressed in Ravel (8). The tool fully replaces the notion of time (along the X axis of space/time views) by the order and simplicity of logical clocks. In exchange, a lateness metric is proposed to encode how much each state is delayed with respect to other states in the same logical step of the application. Since phases are clearly identified, the tool is capable of clustering thread behavior by similarity. The approach enables one to focus on the causal relationship among process and, at the same time, have a perception of bad performance with colors. An alternative approach to handle numerous communication arrows is given through graphical edge bundling (29), keeping the temporal scale untouched. Individual communication actions are bundled together to highlight the high-level communication patterns of the application.

3.2 | Task-oriented Visualization

Parallel applications that are described as a Directed Acyclic Graph (DAG) of tasks have different requirements during the performance analysis, mainly because task scheduling is naturally stochastic. Besides, task dependencies in the DAG impose a certain order on task execution. These are especially important in the beginning and end of the execution, when the number of tasks is not sufficient to occupy all resources. Such kind of perception is nonexistent in BSP-based performance analysis tools. As a consequence, there are very few tools that are truly oriented towards DAG-based application and runtime performance analysis. Very frequently, these tools inspire from the already established field of BSP-based trace visualization, using the intuitive view of Gantt charts supplemented with interactions i.e., mouse pointer. Haugen *et al.* (30), for instance, proposes an interactive Gantt chart enhanced with dependencies, drawn as edges between tasks, that are highlighted when the mouse pointer hovers. We believe this approach suffers from three issues. First, in terms of scalability, since (e.g., in Cholesky) tasks typically have many dependencies (up to `N` outgoing dependencies for `dtrsm` and `dpotrf` tasks, i.e., a total of $\Theta(N^3)$), drawing everything and finding *interesting* tasks and dependencies only through mouse interaction can be very tedious. In practice, only tasks close to the critical path are important. Second, only one-level dependencies are depicted, while several levels are required to understand the history leading to the scheduling problem. Third, this tool does not really account for the heterogeneity of resources.

An alternative approach to Gantt-like views is implemented in DAGViz (31). The tool offers a visual representation of the DAG, which is retrieved using macros (translated to Cilk, Intel TBB or OpenMP) and presented in a hierarchical way. The resulting representation can be folded/unfolded on-demand to show details and the node color indicates where they are executed. There is no way to retrieve the time dimension and task duration, which can make performance analysis difficult. The clear problem with such approach is the scalability: very often DAG with large inputs may be composed of millions of tasks. Even

with an artificial hierarchical organization, exploring the application structure (blocks, tiles), the representation will be very hard to understand.

Temanejo (32) provides similar timeless DAG interactive views for many task-based runtimes. These DAG views can be dynamically improved with information about the resource that executes the task, its scheduling status or its duration. The main features of this tool are the online debug capabilities, e.g., the user can put a breakpoint in a task and fix its dependencies. Temanejo relies on Ayaduame library to interact with the target runtime system and control the application execution at the task level. These capabilities are very useful during algorithm design on small scale, but unsuited for performance analysis.

3.3 | Challenges and Motivation

The design challenges for novel analysis techniques targeting task-based applications have two parts. The first one is behavioral, connected to hypothesis formulation and checking. For such a challenge, we need to be able to answer the many key questions regarding the runtime. The evaluation of scheduling decisions must enable the analyst to understand, for instance, the reason why one task that belongs to the critical path has been delayed. This kind of micro evaluation must be carried out together with a macro analysis, such as, what is global idleness of resources. One must also consider the DAG structure, compare to lower bounds, compare against other executions with slightly different configurations, and so on. To address these behavioral challenges, we believe that a declarative, flexible analysis, must be proposed (33). The second type of challenge is technical, and it is related to the feasibility when faced with large-scale scenarios where traces are voluminous. Any analysis framework should scale and be responsive, enabling one to rapidly iterate in the performance investigation loop.

The next section presents our visualization framework and workflow, addressing all these issues and challenges.

4 | VISUALIZATION FRAMEWORK AND WORKFLOW

Data visualization enables one to check many assumptions at once. Automatically checking some of these assumptions through computations would require to build on more hypothesis that would also have to be verified. This justifies an analysis with a list of various expectations made on the system or application under investigation. For the task-based Cholesky case used in this work, the usual expectations on uniformity, task dependencies, progress, potential improvements, aggregation and filtering, and multi-node data distribution, are as follows.

- **Uniformity.** Task duration is expected to depend solely on their type (`dgemm`, `dsyrk`, `dtrsm` or `dpotrf`) and on the type of resource (CPU or GPU) on which it is executed. Such assumption should be visually verified, highlighting all tasks whose duration is abnormally large compared to the others of the same type/resource. We expect outliers to be space/time location independent, unrelated to other tasks behavior. If not so, it may mean that the whole platform has been perturbed at particular moments or that some resource differs from the others. For this particular analysis, we tag a task as anomalous if its duration exceeds the sampled third quartile plus 1.5 times the sampled interquartile range. This outlier notion has shown to be an effective anomaly classifier for this analysis.

- **Dependency problems**. Large input matrices generate a parallelism explosion after the beginning of the application. We therefore want to monitor the number of ready and submitted tasks. For this Cholesky implementation, all tasks are expected to be submitted when the application starts on each node participating in the execution. On scale, the number of task dependencies is extremely large. Automatically selecting which ones to display is haphazard. If a detailed view becomes necessary for some task dependencies, we rely on the scripting capability of our framework to select and visualize the offending task dependencies from the performance point of view. A common way to understand problematic task dependencies is to select tasks in front of idle time, because there should not be idle time whenever there is enough parallelism.

- **Progress**. The task graph resulting from dense linear algebra always share a common structure (for instance, see Figure 1 in Section 2.2). In a classical semi-sequential execution, the DAG would be executed much similarly to a *breadth*-first search. However, it is also possible to carry out a *depth*-first traversal, favoring task execution on the critical path. Following the pipelining of the sets of tasks submitted by each outer loop iteration can be sufficient to get an overview of how the scheduler is handling the DAG and if it corresponds to the analyst's intuition or not.

- **Potential improvements**. Dependencies are expected to be easily handled with large workloads. To check whether improvements are still possible for a given run, one might rely on the classical scheduling bounds such as the *area bound* and the *critical path bound*. Such bounds, especially the area bound, are expected to be tight when the workload is large and allow to estimate how much further improvement can be expected. More accurate lower bounds (34) could be used as well, in particular for intermediate size workloads. Furthermore, an ideal task allocation can sometimes be infered from such bounds, which may allow to help understanding how scheduling could be improved.
- **Aggregation and Filtering**. Displaying information on hundreds of thousands of tasks on a small area in a blunt way generally leads to harmful visualization artifacts (35). For example, in a classical Gantt chart, visually estimating how much time was spent idle can be quite difficult. This is why it is generally important to filter useless information (e.g., with thresholds configured by the analyst) or to aggregate it in a meaningful and non ambiguous way.
- **Multi-node data distribution**. In an ideal scenario, the workload is evenly distributed among the participating MPI nodes (different machines of the computation pool of resources). If that is not the case, performance losses might be explained by the communication-boundness in the end of the execution, where the critical path going through very loaded nodes delays remaining MPI processes.

From such list of expectations, we propose a series of visualization panels designed to verify assumptions. They can be used by application and scheduler developers, assisting them to rapidly identify performance problems as well as potential solutions for task-based applications. The set of hypothesis to check is fairly rich in heterogeneous multi-node platforms targeted by task-based runtime systems. It is thus important to build a visualization framework that enables one to easily and rapidly **combine various views** and **propose new alternative views in an agile way**. Moreover, since dynamic scheduling and machine heterogeneity bring a lot of variability, the ideal visualization should **exploit** any **potential regularity** coming from the **application algorithm**. For example, as we have seen in Figure 1, each task can be identified by the loop indexes i, j, k. Such kind of information is much more useful than internal runtime identifier and should thus be provided by the application to the runtime so that it can be traced and further exploited during the visualization.

To meet these different goals, we decided to build our framework on top of modern data analytics tools, combining `pj_dump` (27), the `R` programming language (36), its `ggplot2` library (37) and the data manipulation functions provided by the `tidyverse` meta-package (38), org-mode (10), and plotly (39) for interactive exports. Section 4.1, along with Figure 3, presents an overview of the visualization panels proposed in our framework. Section 4.2 describes fundamental data aggregation techniques necessary for a proper data visualization on scale. The final workflow is presented in Section 4.3, and depicted in Figures 10 and 11. This approach allows to build static views in a fully automatic and very efficient way. Although such visualizations could probably be sped up even further by programming everything in C/C++, the used libraries are already well optimized and benefit from the know-how of data analysts. Furthermore, a combination of small scripts is easier to maintain and adapt to a new necessity or to a particular situation than a rigid monolithic visualization environment. We finally present limitations, in Section 4.4, discussing workarounds.

4.1 | Visualization Panels

Our framework enables an easy composition of multiple visualization panels to evaluate different performance scenarios. As an introductory illustration, we consider the Chameleon/Cholesky decomposition of an input matrix of dimension 72,000, divided in 75×75 tiles of size 960 (i.e., with 75 `dpotrf` tasks), executed on two nodes comprising five CPU and two GPU workers each, and interconnected through a 10 Gb/s Ethernet network. The StarPU runtime has been configured with the `prio` scheduler (with a central queue on each node, sorting tasks by priorities given by the developer), and dedicates, on each node, one core for task submission (using the Sequential Task Flow paradigm (40)) and another core to handle MPI operations. Figure 3 shows, from top to bottom, the composite image generated by our framework. The plots are temporally-aligned (the X axis represents time) and depict the behavior of three layers of the computation system: application, runtime, and platform metrics. The application behavior (Section 4.1.1) consists in the Cholesky Iteration (A), Application Workers (B), and Submitted Tasks (C). The runtime performance footprint (Section 4.1.2) is depicted with the StarPU workers behavior (D) and the ready tasks plot (E). Finally, performance metrics gathered from the platform (Section 4.1.3) have the observed GFlops rate (F), for both CPU and GPU resources, the GPU memory bandwidth (G), the MPI communication bandwidth (H) and the number of concurrent MPI operations (I). Albeit being representative, other plots may be added very easily according to the needs of a particular performance analysis. In what follows, we detail each of this plots individually, grouped by layer.

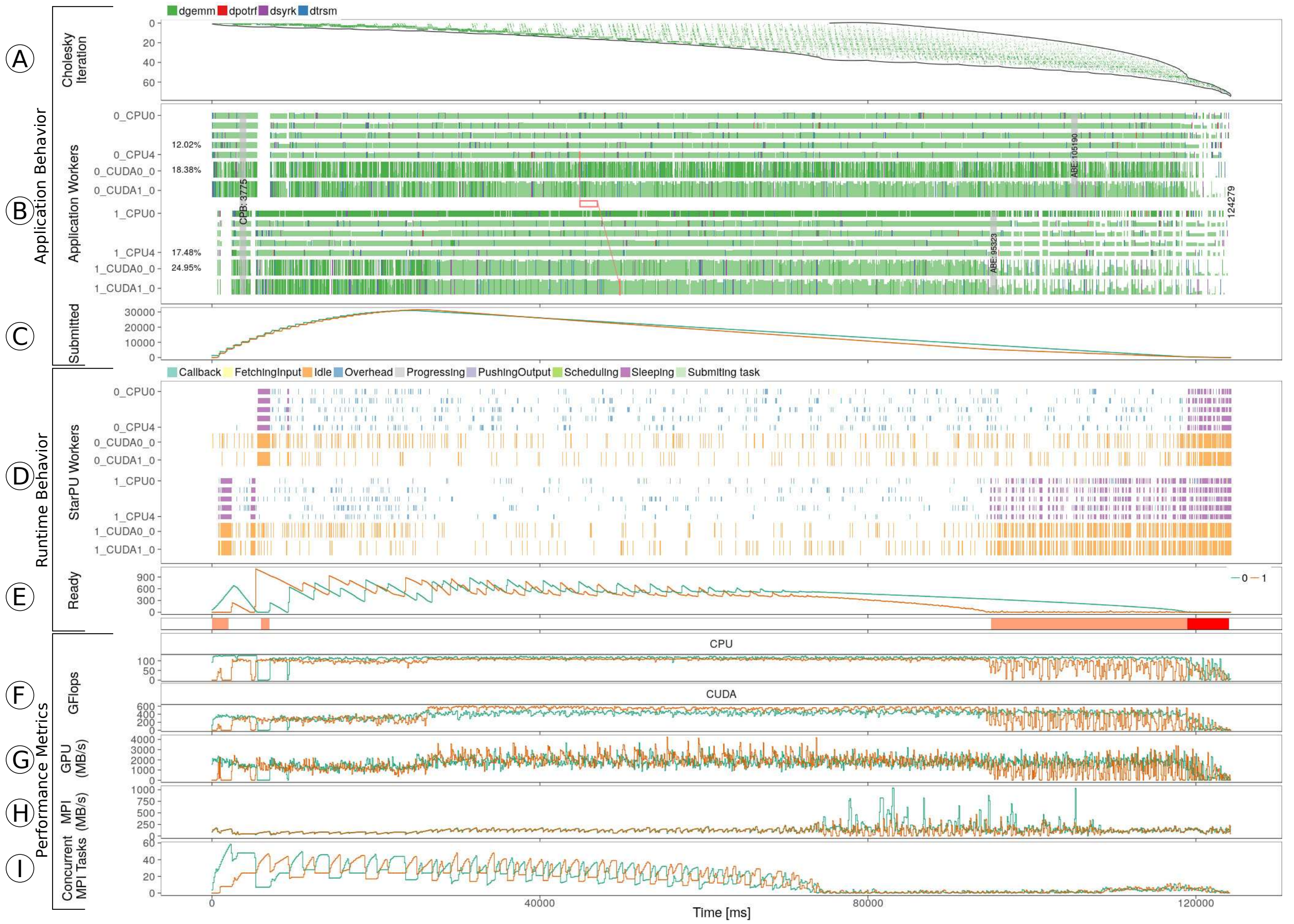


FIGURE 3 Overview showing the visualization panels available in our framework, for a 2-node multi-core and multi-GPU Chameleon/Cholesky experiment. Panels are grouped by system layer (application – top, runtime – middle, and platform – bottom): the application contains the Cholesky Iteration plot (A), the task behavior (B) and the submitted tasks (C); the runtime has the state of StarPU workers (D) and the amount of ready tasks (E); the platform performance metrics are composed of GFlops rate (F), GPU memory bandwidth (G), MPI network transfers (H) and the number of concurrent MPI operations (I).

4.1.1 | Application Behavior

The application behavior is detailed with three visualization panels, as shown in Figure 3: the application-specific Cholesky Iteration plot (A), the enriched space/time view with the application workers behavior (B), and the number of submitted tasks (C). All panels show information along the execution time (the horizontal axis).

As previously discussed, the tiled version of the Cholesky decomposition has an outer main loop which provides a hint of the general progression on the critical path. Tasks are tagged, when being submitted, according to their membership to a given loop. Using the tags in the visualization, the Cholesky Iteration plot, in further details in Figure 4, depicts the tasks (color) along time (on X axis) according to their loop number (on Y). The vertical coordinates run from 1 to 75 because there are 75 blocks in the case. Tasks may be drawn one on top of the others on each row because several workers might be computing tasks of the same loop. For a given row (one Y axis coordinate), the lack of tasks (white areas, as pointed by A.1) indicate that there are no workers computing that particular loop for that specific time frame. The two rounded borders (on the bottom, A.2; on the top, A.3) indicate the specific timestamp when a given iteration of the outer-most loop starts to be computed until the moment the last task belonging to this iteration is completed. The borders and the inner task glyphs provide an unique application signature directly influenced by the runtime scheduler. For example, it shows the number of loops that are active at the same time: at ≈75s, the `prio` scheduler is computing a little less than 40 loops at the same time, achieving high parallelism.

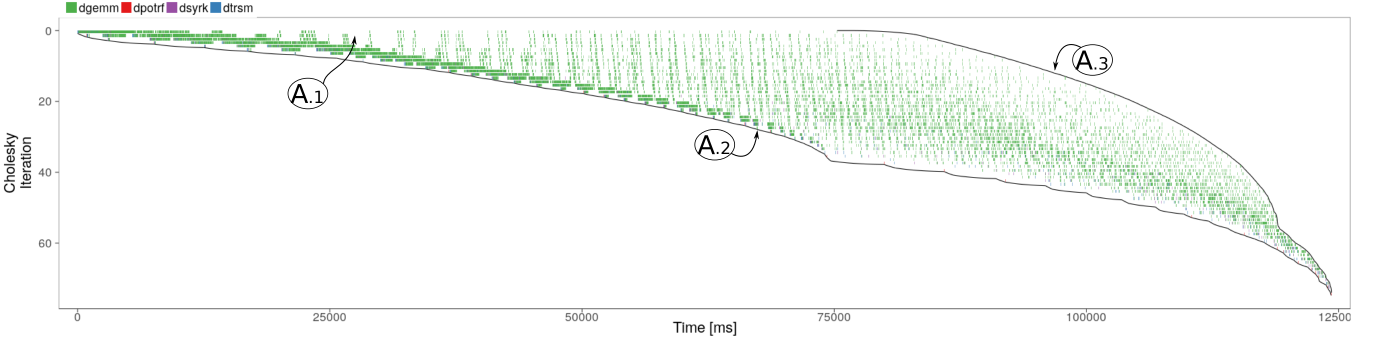


FIGURE 4 The Cholesky Iteration plot (A) shows the application-tasks (colors) along time (on X) using as Y coordinates the loop the tasks belong to. Periods of time where a loop is not being computed are shown as white areas (A.1); while the bottom (A.2) and top (A.3) borders illustrate the moments when loop iterations start and finish to be computed. These borders provide an application signature and can be used to compare runtime schedulers.

Figure 5 illustrates the main elements of the enriched space/time view proposed in our approach (left) and the solution of the Area Bound Estimation (right). The space/time view shows the task execution, represented by colors (for Chameleon/Cholesky: `dgemm` are green, `dpotrf` are red, `dsyrk` are violet, and `dtrsm` are blue), along time (X axis) by the resource which executed each task (Y axis). The height of the colored rectangles (B.0), in each resource row, correlates with the resulting values of the task-aware temporal aggregation (more details in Section 4.2.1). Besides this, the maximum height dedicated for GPUs is two times the height occupied by CPUs, since GPUs provide a higher GFlops rate than CPUs. The **resource hierarchy** (B.1) groups resources by node (Node 0 and top, Node 1 in the bottom). Our makespan definition is the time from the beginning of the first application task to the end of the last. The **observed makespan** (B.2) is drawn in the graphical position identified by its value. In the example of the figure, the total makespan was of 124,279 ms. The graphical representation might mislead the analyst to think that the resource occupation is good, since many small idle periods might be hidden because of the overwhelming number of tasks. Besides that, visually comparing two or more resources occupation is hard. For these reasons, we compute the **overall idleness** (B.3) of each resource. We can see that idleness of GPUs in Node 1 is higher (≈22%–25%) than those of Node 0 (≈12%–18%). For the tiled Cholesky decomposition we have used in this work, tile size is fixed, which enables to create simple performance models for each kind of task. We exploit this application characteristic to detect **task execution outliers** (B.4: see region marked), which are represented with darker colors. The outlier definition is provided by the analyst according to the knowledge of application. In this particular case, a task is considered anomalous if its duration exceeds the sample third quartile plus 1.5 times the sample interquartile range. Although this outlier notion is highly debatable and context-specific, other definitions could be easily incorporated (e.g., if the analyst has an a priori knowledge on the task duration distribution). In the example of the figure, we can see that CPU0 of Node 1 and the top GPU of Node 0 always demonstrate `dgemm` execution anomalies. Both GPUs of Node 1 generate outliers in the first 25 seconds of execution; after that, tasks are mostly normal. Besides considering the load balancing within a node, the multi-node tiled Cholesky decomposition must take care of equal load distribution among all nodes involved in the computation, considering CPU/GPU heterogeneity. Since one knows the average time $w_{r,t}$ needed to perform a task of type t on a resource of type r on a given node, as well as the total number n_t of tasks per type, one can consider that a fraction $\alpha_{r,t}$ of tasks of type t will be done on resource r and that the $\alpha_{r,t}$ should thus verify:

$$\forall r : \sum_t \alpha_{r,t}.n_t.w_{r,t} \leq T, \text{ where } T \text{ is the total execution time.}$$

Since such constraints are linear it is possible to compute the optimal makespan T and the corresponding allocation $\alpha_{i,k}$. The T value is called the **Area Bound Estimation** (ABE) (B.5) and is a lower bound for the execution time for the load assigned to each node. We can see in the example that load distribution is not equal: the estimation for Node 0 is ≈125 s, while for Node 1 is ≈95 s (24% less). We depict in the right part of the figure the solution of the linear program (B.6): the bars depict the observed load distribution (per node, resource, and task types), while the dots represent the ideal solution given by the linear program. For instance, a fair amount of `dsyrk` and `dtrsm` tasks were executed on GPU (CUDA), but the ideal solution tells us that much more should be given to the CPUs. Another way to evaluate the tightness of the schedule is the **Critical Path Estimation** (CPB) (B.7). It is calculated by summing the observed duration of all tasks in the critical path of the application DAG, assuming they are executed on the faster processing resource for that task. This can also be viewed as a minimal execution time if application

had unlimited number of resources of every type. In the Cholesky case with large matrices, this is a poor metric because there are no `dgemm` tasks in the critical path, while they represent the bulk of the workload and dominates the performance. Finally, application and schedulers developers might want to confirm that once all dependencies are satisfied, the task is executed right away. Since the DAG might be potentially large (in the example: 73,150 tasks and 427,432 task dependencies) we provide the developer an API to draw the **task dependency chain** (B.8). The backward chain is obtained by recursively searching the last task that releases a given task, up to a limit provided by the user. In the example of the figure marked by B.8, we calculate two backward steps starting with a task that has been executed in the bottom GPU of Node 1. We can see that it was released by an MPI task (center), which itself has been executed to receive data from a task that executed in one CPU of Node 0. We noticed that when the MPI task finishes, the task it releases is not immediately put into execution (see the diagonal red line connecting both). This can be a potential performance problem if there are no other tasks to be executed at that moment, which is not the case in this specific timeframe. Such kind of task dependency analysis can be carried out with any number of tasks.

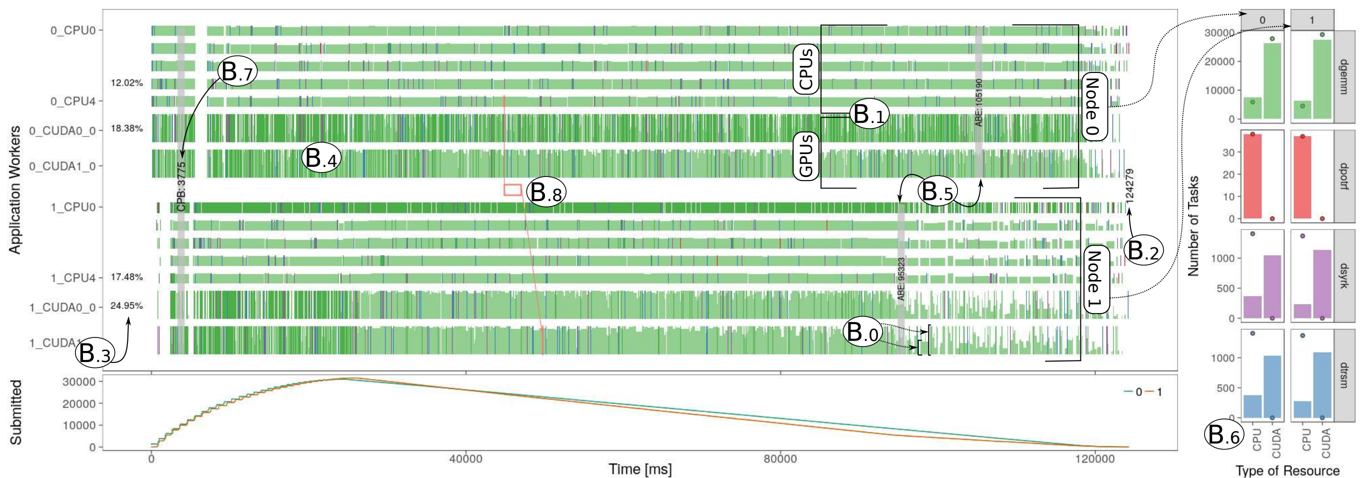


FIGURE 5 The enriched space/time view (B) depicting task execution (color) along time (the X axis) per resource (Y), hierarchically organized (B.1), with total makespan (B.2), per-resource idleness (B.3), outlier highlighting with darker colors (B.4), per-node area bound estimation (B.5) with the linear programming solution on the left (B.6), and critical path bound estimation (B.7). These features are combined with task dependencies chains (such as the example of B.8), selected by the analyst during the analysis to check if tasks are executed as soon as possible. The bottom panel shows the per-node task submission (C) with the number of submitted tasks along time.

In StarPU-MPI, there is one thread responsible for submitting tasks per node. The performance of these submission threads is very important to the overall behavior of one execution, since task availability is fundamental to make StarPU exploit efficiently all resources available. The number of submitted tasks per node in a given moment is traced by the runtime. We use this per-node data to provide the Submitted panel, as shown in the bottom of the Figure 5. It depicts the number of submitted and still unfinished tasks (in the Y axis) as a function of time (the X axis) per node (color). We can see the stair-case climb of submitted tasks until 25 seconds of execution, where the maximum number of submitted tasks is achieved in both nodes. After that, there are no more task submission with a steady decline, as the tasks are being run on processing units, until the end of execution. Task submission, in this case, is similar on both nodes.

4.1.2 | Runtime Footprint

The runtime behavior of each StarPU worker is depicted in Figure 6, where the states are represented by colors along time and hierarchically organized per resource (the vertical axis). In an ideal scenario, the footprint of the runtime behavior, represented by all these states, should be close to zero. Very often, however, such ideal scenario is hardly achievable because of the scheduler implementation, data copies between devices and nodes, and other fundamental activities of the runtime. The observable idle periods of the runtime with the Idle (green) and Sleeping (pink) states in both GPUs and CPUs (see the regions marked by D.1) coincide with the lack of ready tasks in each node (see the dashed arrows pointing to the ready tasks panel). They are caused

by task dependencies delays between the two nodes and may be considered normal during the kickoff of Chameleon/Cholesky executions in platforms with poor network interconnection. At the end of the execution (regions marked by D.2), the idle periods shown by StarPU highlight the decrease of task parallelism. We can note that Node 1 (bottom) is affected first because, as seen in Figure 5, the load distribution is not the same for both nodes.

The Ready tasks panel shown on the bottom of the Figure 6 complements the visualization of the StarPU workers behavior. It shows the amount of ready tasks at a given moment, per node (color). The horizontal bar (E.1) becomes redish whenever the amount of ready tasks is smaller than a given threshold related to the number of computing resources. This implies that the runtime system is (or will soon be) unable to generate enough ready tasks, causing resource idleness and potential performance loss. The threshold can be configured by the user but if unspecified (which is the case in the example of this figure), the threshold is automatically defined as the minimum amount of workers per-node (seven for this example, since there are five CPUs and two GPUs on each node). This bar becomes more useful when the number of nodes increases.

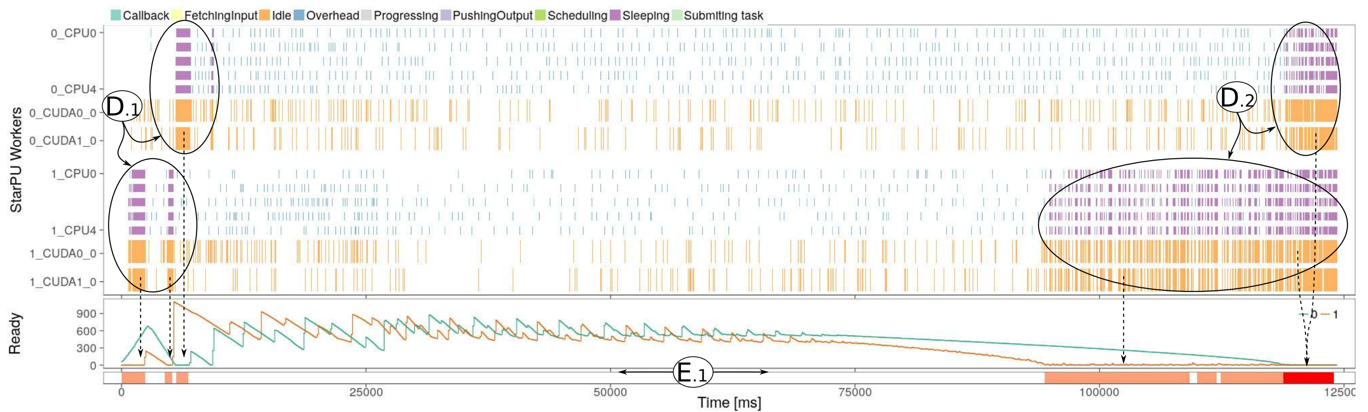


FIGURE 6 The space/time view (D, on top) depicting the StarPU workers states (color) along time (the X axis) per resource (Y), adopting the same hierarchical structure as in the application-level view shown in Figure 5. On the bottom, the ready task panel (E) depicting the number of ready tasks for each node, along with a horizontal bar (extreme bottom) to highlight when the ready tasks are smaller to a configurable threshold.

4.1.3 | Platform Performance Metrics

Many platform metrics can be combined with the application and runtime behavior. Figure 7 illustrates some examples that are used in the analysis of StarPU-MPI: the GFlops (F), GPU memory (G), MPI network bandwidth (H), and concurrent MPI operations (I). These metrics are either obtained directly from the runtime traces, requiring some post-processing to remove extreme data points that appear due to measurement uncertainty, or they are derived from other data traces using scripting techniques. Post-processing using smoothing techniques are employed in all rate metrics (GFlops, bandwidth). Regarding the **GFlops** panel (F), for each resource type (CPU or GPU), we calculate the average rate along time, using time slices that are sufficiently small to capture small temporal variations. The rates are depicted with horizontal facets (CPU on the top). We can see that CPUs achieve peak performance much faster than GPUs, whose plateau performance only appears after 25 seconds of execution. GFlops rate drop by the end of execution, because tasks dependencies limit the parallelism and that are not enough ready tasks. The **GPU memory bandwidth** (G) details the per-node device→host and host→device data copies. Values are averages considering all the GPUs available on each node. The phases are similar to those observed in the GFlops panel, with data copies peak going from ≈25 to 100 seconds of execution. The **MPI network bandwidth** panel (H) represents the outbound bandwidth achieved by MPI operations for each node (colors), while the **Concurrent MPI tasks** (I) panel depicts the number of asynchronous MPI task operations. In this example, a 10 Gb/s Ethernet interconnection has been used for the experiment. By consequence, we notice that, despite the many asynchronous operations managed by StarPU (up to 60), the effective network bandwidth never achieves the maximum theoretical upper bound. Different phases can also be perceived in such metrics. For example, at ≈75 seconds of execution, there is very few concurrent MPI operations, enabling a higher (but much more variable) effective MPI network bandwidth to be achieved.

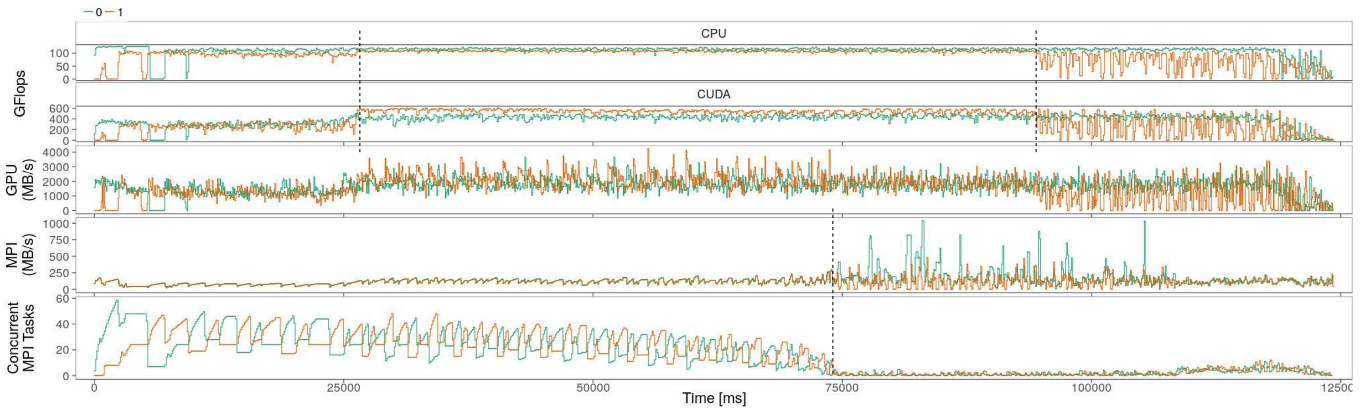


FIGURE 7 Four performance metrics: aggregated per-node GFlops rate (F), GPU memory bandwidth along time (G), effective MPI network bandwidth (H), and the number of concurrent MPI operations (I). We can notice how the different application phases (beginning, middle, closure), appear in these platform metrics. These can also be correlated to the phases observed in Figure 5.

4.2 | Addressing Scalability

When tracing the application behavior, even for fewer processing units counts, the volume of traces might be extremely large if lots of application details are registered. Some kind of data reduction must be carried out to scale the visualization of these traces. Several techniques exist, some reducing the complexity only in time, others both in time and space. In our framework, we have designed and implemented two temporal aggregation methods: the first for space/time plots, where the goal is to selectively aggregate only task types that are numerous; the second for line plots, where the goal is to reduce the excess of details that would be impossible to render in a computer screen. We detail these techniques below with real examples.

4.2.1 | Space/time plots: task-aware temporal integration

With DAG-based applications, some types of tasks might be critical for the overall performance of the application. That is the case of the `dpotrf` task in the tiled Cholesky decomposition as there is only one such task per tile and it releases many `dtrsm` at once. Tracking them is important since it enables one to evaluate how the runtime scheduling behaves. These rare tasks should be excluded from any kind of aggregation and kept in their original form, unaggregated. Another reason for not aggregating a task is when they are considered outliers, clearly demonstrating temporary performance problems. On the opposite, tasks that are numerous (in Cholesky, the `dgemm` tasks) should be aggregated if previous concerns are respected.

Our temporal aggregation algorithm works by merging consecutive tasks of specified types for each resource (CPU or GPU). We let the analyst decide which task types have to be merged. Figure 8 depicts an example showing how aggregation works for `dgemm` tasks. The considered area (marked on the top space/time panel) is shown with raw (non-aggregated task) without (A) and with outlier detection (C). The corresponding aggregated versions are shown in the bottom row, without (B) and with outliers (D). The circled region marked in the origin of arrow AB1 shows how consecutive `dgemm` tasks are merged together (the destination of arrow AB1). They are temporally limited (before and after) by `dtrsm` tasks that are out of the aggregation scope. Together, they become a single graphical object that represents the percentage of occurrence of the `dgemm` state within that time frame. As seen in the raw/outlier panel (C), the circled `dgemm` tasks are in fact outliers (as shown in C1, depicted by a darker green color). By consequence, when outlier highlighting is enabled, these tasks are not aggregated, as shown in the end of arrow BD1. Aggregated tasks are presented with lighter color while non-aggregated ones (`dpotrf`, `dsyrk`, `dtrsm` and the outliers) are drawn in their original colors. The percentage of occurrence of a given task is graphically depicted by the height of the rectangles that represented an aggregated state. This is shown in the AB2 arrow: the aggregated version indicates that for the given time slice, there is less than 50% of the time that is spent with `dgemm` tasks. This specific case also enables us to illustrate how the idle gaps (white areas) are kept in the aggregation process. When aggregating more than one type of tasks in the same time slice, we use stacked bars to represent the percentage of time spent in each state.

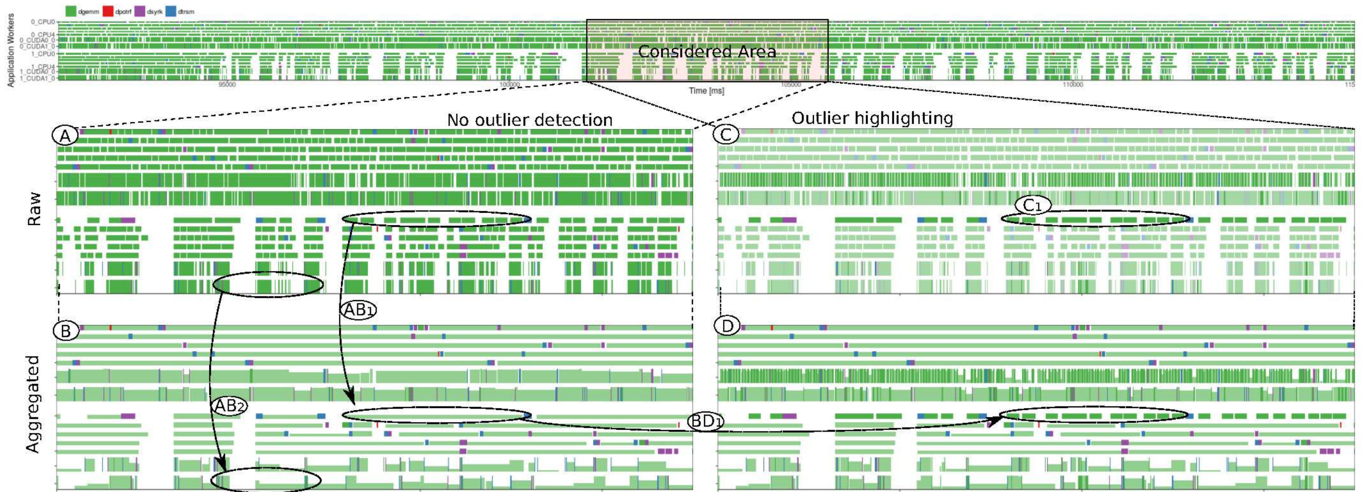


FIGURE 8 The task-aware temporal integration technique: raw states are depicted without (A) and with (C) outliers, and the corresponding task-aggregated versions are shown in B and D. The AB1 arrow demonstrates how consecutive tasks are merged together, a mechanism that is not carried out when original tasks (C1) are outliers (BD1).

4.2.2 | Line plots: smoothing and reducing the complexity of variables

Performance metrics are measured by the runtime system and recorded in trace files. The frequency on which the data points are measured is low enough to not harm performance, but is high enough to complicate the performance analysis. There are at least two problems that affect the data interpretation when visualizing the metrics. The first one is related to the visualization scalability, since the number of pixels on the screen is not enough to represent all the data points (35). The second problem is that many metrics gathered by StarPU, especially those related to the GPU (memory bandwidth, GFlops) and MPI (bandwidth) are in fact estimations. Tracing these layers lead to data points that can have peaks that appear even larger than the hardware peak performance.

To address these issues, we employ temporal integration to smooth metrics and effectively reduce the number of data points before the visualization. The temporal integration works by discretizing time. We calculate the average value for each metric, in each time slice, and each resource. By default, the time slice is equal to 0.1% of the trace makespan, but this value can be easily changed individually for visualization procedure. To illustrate how such temporal aggregation is important, we depict in Figure 9 three cases without the aggregation (top row), aggregation every 10 ms (middle), and aggregation every 150 ms (bottom). The left part (A) shows the first 10 seconds of the whole time series, depicted in the right (B) for two nodes (color). Since we are using the same case depicted in Figure 3, where each computational node is equipped with two GPUs, each line represents the bandwidth average of these for each time slice, and for each node. We can clearly see that raw data is completely unusable, with peak values going up to 20,000 MB/s, way beyond the peak bandwidth performance of the GPU devices. In this case, temporal integration every 10 ms is insufficient to detect the overall behavior, which only appears when a time slice of 150 ms is used. Only then, one can see the different phases, already discussed in previous subsections.

Despite aggregation being indispensable, the analyst must be aware that temporally integrating variables might hide anomalies or even create artificial phases (periods of time with constant values) that are not initially present. The rounded rectangle A1 (showing the period of time that ends at 2500ms) in Figure 9, illustrates such scenario. The (red) line contained in the rectangle demonstrates a series of very rapid bandwidth peaks. Using 10 ms and 150 ms time slices create an averaged value of $\approx$2,000 MB/s giving the idea that such average was constant for the whole period of time, which is not the case. Instead, a series of bursts of host-device communication occurred. For this reason, we provide the analyst a per-metric time slice threshold configuration to let the problem be mitigated when possible.

4.3 | Two-Phase Workflow Implementation

Instead of building a complex monolithic tool, we follow the UNIX philosophy and script many small tools that are combined together in a two-phase workflow. The first phase, depicted in Figure 10, is the pre-processing, where data is ordered, cleaned,

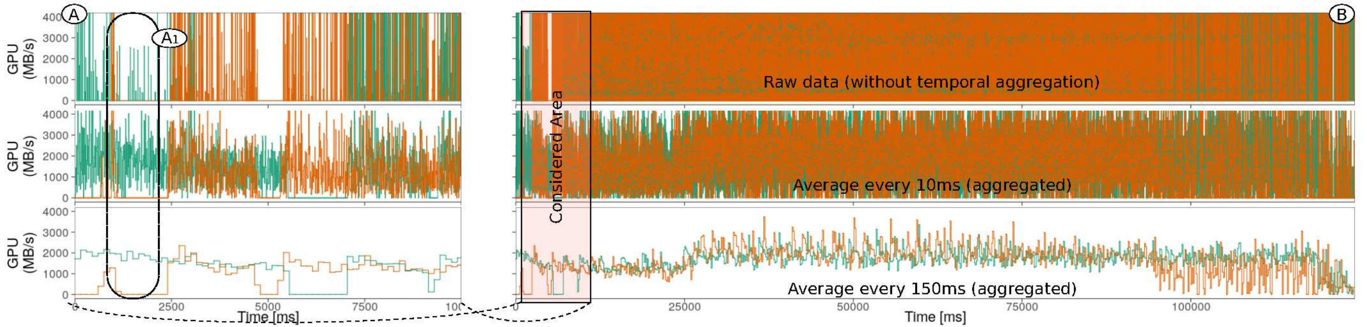


FIGURE 9 Dealing with performance metrics estimations: raw data (top row), temporal integration every 10 ms (middle) and every 150 ms (bottom) with a zoom in the first 10s (A, left) of the overall execution (B, right); one can notice how fundamental it is to conduct this kind of aggregation to extract useful patterns from the raw data.

filtered, and the different inputs are combined to create new static information. The second phase, shown in Figure 11, enables in-memory data visualization with code for the visualization panels that have been previously described. These two workload figures include only the main components (small black dots are used to depict other components) to explain the key manipulations. The whole source code for the workflow is available in the companion website https://gitlab.in2p3.fr/schnorr/ccpe2017/. All modules are written in the `R` language (36) except when marked otherwise. The glue connecting them is written either in `SHELL`, `R`, or a mix of both. The purpose of having two phases is scalability. The first phase contains all data manipulation, computed only once, and its results can then be used many times for various visualizations in the second phase. This first phase might eventually be slow at scale, when traces comprise runs with more than one thousand workers for small inputs; or much fewer workers but with larger workload. Nevertheless, this phase can be executed in parallel in the same nodes that had been used for the execution of the program being analyzed. In what follows, we detail the steps in each workflow phase.

The pre-processing phase (see Figure 10) gets as an input several `FXT` files (41) dumped from memory by StarPU at the very end of the application execution. With few minor modifications to the Chameleon/Cholesky, traces are enriched with application tasks tagged with loop indices at the task creation. (A) `FXT` files are exported to `DOT` and `PAJE` files using the `starpu_fxt_tool`. The `PAJE` trace file (27) contains timestamped events that describe the application behavior (task execution) for all workers. It also contains many important metrics from the StarPU runtime, such as the scheduler and MPI states, GPU bandwidth estimation, inter-node communication links, number of ready and submitted tasks, platform architecture, and so on. (B) The `pj_dump` tool parses the `PAJE` file to verify the structural and temporal integrity of the trace. As output, four Comma-Separated Values (`CSV`) files are registered with the states (application tasks with their unique ids and runtime behavior), entities (platform and worker hierarchy), links (MPI communication events, in multi-node runs), and variables (with runtime and platform performance metrics). The complete application `DAG`, written in the `DOT` file format, contains task dependencies and task identifiers coherent with the `PAJE` trace. It also gets converted to `CSV`. (C) These `CSV` files are read to memory using efficient parsing `R` functions from the `readr` and `data.table` packages. On scale, such process might be slow since `CSV` parsing involves many string manipulation, a process regarded as slow. Because of that, as output of the first phase, we use binary files instead of a textual file format such as `CSV`. (D) Thanks to the expressiveness and to the rich set of statistical libraries in the `R` language, many cleanups, filtering and statistic computations are done with only few lines of code. State data is read into memory first because we impose, just after outlier detection, a new `ZERO` timestamp for the data, which is defined as the moment when the first application task starts. All data before this specific moment, corresponding to StarPU warmup and initialization phase, are filtered out. The new zero definition is re-used as offset to correct timestamps for links and variables. The `CSV` entities file contains the application and platform hierarchical structure. Since we are interested in the behavior of processing units (StarPU workers), we keep only those and the MPI threads, calculating the Y coordinates for the space/time view. The coordinates for every resource are merged with the states using a left join operation based on the resource identification as key. As of result, all application tasks have X (time) and Y (space) graphical properties. Links and variables suffer minor manipulation. The `DAG` is enriched with temporal data from the states (tasks) through a left join operation using the task identification as keys. In multi-node runs, every MPI operation is a node in the DAG, but the temporal information for such MPI tasks is registered in the links `CSV` file. This requires another left join operation using the MPI unique task identification. (E) At the end of the pre-processing phase, the data is registered in binary code respecting the `FEATHER` (42) file format (using the `feather` package in `R`). The main advantage of this format is that

the reading is bounded by the hard drive performance, enabling a much faster second phase for data visualization, described as follows.

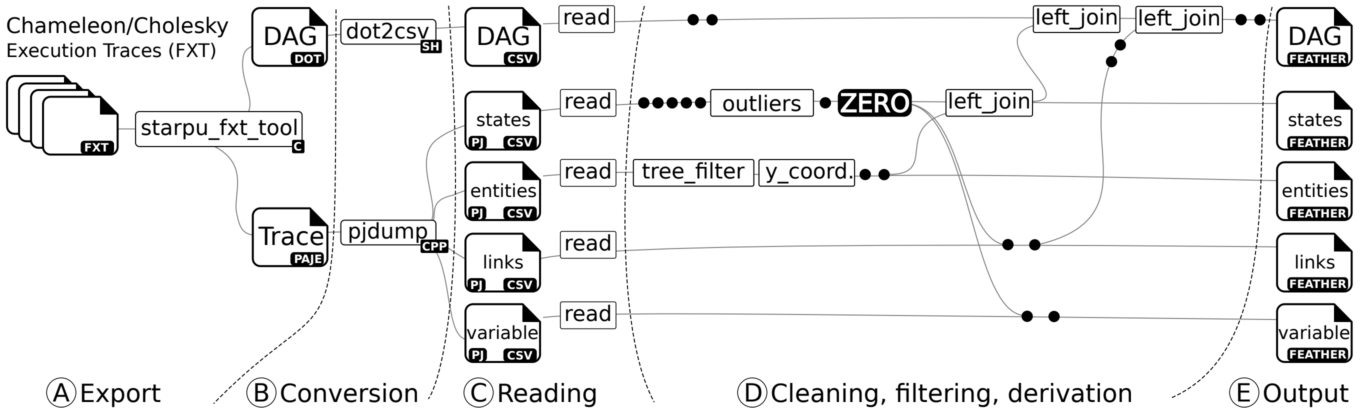


FIGURE 10 Simplified view of the workflow for data pre-processing: traces with timestamped events are dumped from FXT files to DOT and PAJE textual file formats using `starpu_fxt_tool`; the `dot2csv` and `pj_dump` tools are used to split trace data in multiple CSV files (DAG, states, entities, links, and variables), which is pre-processed in R, through a series of filtering, data derivation, graphical properties definition, and clean-up steps, and registered as tabular FEATHER binary files for fast reading during the second phase of the workflow.

The second phase (see Figure 11), for data visualization, gets as an input the FEATHER files previously created. As before, all modules are implemented in R except when marked differently. A C++ implementation is required for those modules where data manipulation with R is too slow to be operated on demand. Ⓐ The reading procedure loads quickly the data registered in files into memory, thanks to the `feather` binary format. Each file becomes one data frame. A single structure (represented by the black circle identified as `case` in the figure) binds all data related to a single trace execution. Multiple structures such as these might be kept in memory at the same time to enable multi-trace analysis. Such comparisons are possible by replicating the workflow, with different inputs, and finally combining the visualization output in a single figure. Ⓑ Creating the graphics for the analysis involves a data workflow that includes some data processing. This increases the analysis flexibility since some data is still calculated on-the-fly. Some examples of features calculated by demand include the ABE, CPE, Idleness, temporal integration granularity, drawing of specific task dependencies, task highlighting, and all aspects for the temporal/spatial navigation (zoom in, filtering, selection, etc.). All visualization panels (represented by rectangles with corner marks) are implemented using `ggplot2`. This library provides a grammar of graphics (43) and a very high-level way of building plots, enabling us to easily produce custom visualizations. Some panels are simpler than others. The Cholesky Iteration panel, for example, gets data directly from the states data frame and its implementation is straightforward since all coordinates it requires are pre-calculated in the first phase. On the other hand, the space/time view panel is much richer, since it can employ idleness, outlier and task highlighting, ABE, CPE, and other specific features. The possibly extensive task data is also time-integrated to render the panel faster. All variable-based panels undergo temporal integration according to a user-defined granularity. All metrics, except ready and submitted, require a time integration procedure since the measurements are uncertain, as they are only estimations provided by StarPU. Some spikes for example are commonly found for too-fast MPI (on low-latency networks) or GPU transfers. In such scenarios, time integration enables one to smooth the metrics prior to their visualization. Ⓒ Visualization assembly is customized according to a global YAML user configuration file. The `master` component is responsible to assemble the composite view following users' choices, and synchronize the temporal axis and vertical proportion of all panels. It has enough expressiveness to guarantee different but coherent views (colors, scales, etc.). Ⓓ The analysis is the final part where the graphical object with the composite view is registered in a file (PNG or PDF) or can be exported to HTML using the `plotly` library to provide user interaction with the data.

The workflow performance greatly depends on the size of the FXT files. For a 49-node CPU-only execution with 686 cores (14 cores per node), the FXT files take ≈18 GB of space. In a computer, equipped with one Intel(R) Xeon(R) CPU E3-1225 v3 @ 3.20GHz and 32 GB of main memory, exporting the FXT files to PAJE and DOT takes roughly 10 minutes, while the conversion using `pj_dump` and `dot2csv` takes about 9 minutes. Finally, the bulk of the data pre-processing in R takes approximately 13

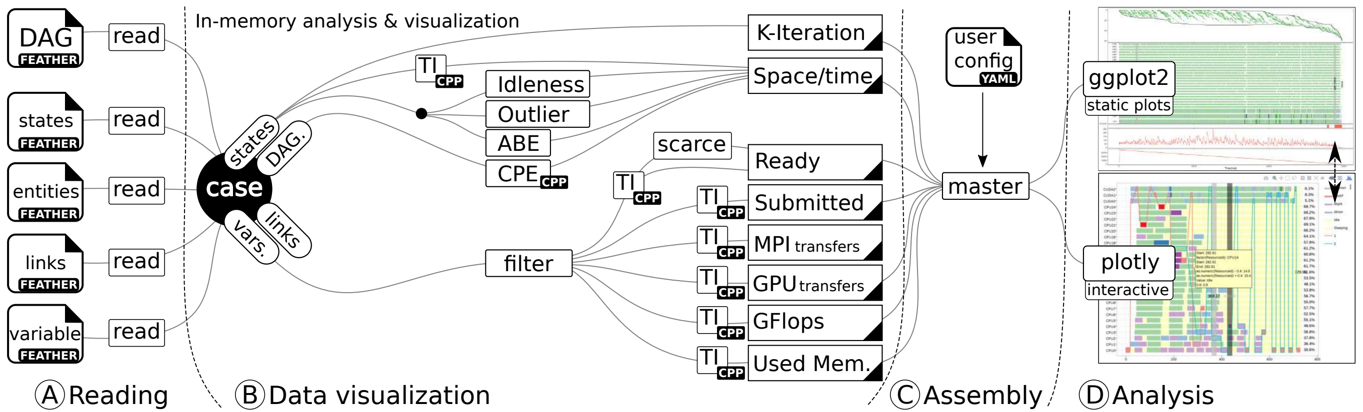


FIGURE 11 Simplified view of the workflow for data visualization: the FEATHER binary files related to one case study are rapidly loaded into main memory; interactively, using a YAML configuration file, the user can create composite views to generate static and interactive plots; all variables and states are integrated in time before plotting.

minutes to finish. The resulting FEATHER files occupy ≈13 GBytes of space. The first phase takes thus 32 minutes to complete. In the second phase of our workflow, reading the FEATHER files in R takes about 30 to 40 seconds. One overview plot considering all the contents (worst case) with the space/time view, ready and submitted variables takes about 42 seconds to complete, while writing the plot to a PNG output takes about 6 seconds. By consequence, since data is kept in main memory, the user has to wait for roughly one minute when a modification in the visualization is demanded. Since the time to render a new plot directly depends on the amount of data, generating new plots can be dramatically faster when zooming in parts of the trace.

4.4 | Limitations

Sometimes, user interaction is required in order to quickly point out tasks and get more information about them. For example, mouse hover capabilities are much more efficient than manual inspection to obtain further information about specific tasks in the representation. As previously described, we provide a way to export the composite views to HTML5 using `plotly` (39). However, the solution scales badly because application traces have too much information, with millions of events and task dependencies. Despite our efforts on data aggregation to reduce the information before creating such graphical objects, the procedure remains usable only for small scale scenarios. For example, a small case with a 4-node run (with 24 CPUs and 2 GPUs per node) of the Chameleon/Cholesky application with a 144K input matrix (tile size of 1,440 elements) generates 171,700 application tasks. Alternatives to tackle the problem do exist. The `bqplot` or the `googleVis` packages are some of them. The former follows the grammar of graphics philosophy (as `ggplot2`) for IPython/Jupyter interactive notebooks. The later (`googleVis`) has functions to generate HTML5 but are much simpler since they are not layered as we need to create customized views. The `bqplot` library seems more promising because they follow a true Model-View-Controller (MVC) approach, which might scale better. Being satisfied with static plots, we have not yet evaluated any alternatives for interactive visualization, adhering with `plotly` in the time being.

Another limitation for using our software to conduct performance analyses is the requirement to obtain traces with enough information from the runtime. During our work, the tracing system of StarPU was enriched with several new events and more detailed information. For example, internal MPI communication tasks now appear in the trace, allowing our software to correctly understand task dependencies when communications are involved. That said, until now, such information was not used by classical trace visualizers and analyzers, so runtime developers had few incentives to generate a detailed trace. We hope that our work demonstrates the interest to be able to collect such information. The StarPU runtime is now correctly instrumented and can be used as-is to analyze any application using it, other runtimes should have their trace easily enriched if needed.

5 | CASE STUDY: PERFORMANCE ANALYSIS OF TASK-BASED CHOLESKY

We describe three performance analysis scenarios of the task-based tiled Cholesky decomposition application (see Section 2.2). Together, they show different facets of how our visualization elements can be combined together to attain a successful performance analysis. The first one [2] provides a **comparison of runtime schedulers** and their impact on performance in a single yet heterogeneous node (24 cores and 3 GPUs). We also detail in this first scenario a slight change in the application to force some application tasks to execute exclusively on GPUs. We explain with our views the reason behind the resulting performance gains. The second scenario investigates the **MPI slow start**, demonstrating idle gaps caused by an unoptimized MPI layer. The analysis is conducted on hybrid multi-node setting, with a focus on task dependencies, outliers, ready/submitted variables, along with MPI bandwidth and the concurrent MPI operations plots. The third and final scenario investigates the potential negative impact of **static data distribution strategies** at scale. Per-node ABE is used to compare how data distribution and performance is affected by different values of the P parameter (see Section 2.3). These three scenarios have been chosen during many sessions of performance analysis and multiple tests at different scales.

Section 5.1 details the experimental platforms and the software that have been used in our experiments. The usual expectations for the performance analysis of the tiled Cholesky decomposition implemented as a DAG are detailed in Section 4. Section 5.2 discusses the first scenario about scheduler comparison. Section 5.3 presents the MPI slow start with major idle gaps during the kickoff of the application run, and the fixes that have been suggested to the runtime developers to fix the problem. Section 5.4 details the study about data distribution strategies on scale.

5.1 | Experimental Platform

Since hybrid heterogeneous nodes motivate the development of task-based runtimes, we execute this Cholesky implementation over clusters of computers that are composed of at least two GPUs. Table 1 provides hardware details about the two main platforms we have concentrated our experimentation on: **idcin2** and **chifflet**. In the former, only 25 CPU cores (out of 28) participate in the computation because StarPU requires one core to manage each GPU. In the latter, we have used six cores per-node to make the figures of the paper clearer, easier to understand. The chifflet experiments using all the cores demonstrate performance issues that are the same or worse than the results shown in the paper. In multi-node experiments, StarPU requires one core exclusively to run the MPI thread responsible for the message exchanges (see Section 2.3). Regarding software, the machines use recent versions of the Linux Kernel distributed in Debian testing (buster); and all the software related to experiments is installed via Spack (45): Chameleon (0.9.1, master), StarPU (developer, using specific revisions with the performance fixes proposed in this work), OpenMPI 2.0.2, OpenBLAS 0.2.19, and CUDA 7.0 (chifflet) / 7.5 (idcin2) with driver 375.66 (chifflet) / 352.39 (idcin2). All experiments are conducted using a reproducible methodology where all known variables acting as possible parameters for the experimentation remain as fixed as possible (disabling HT, TurboBoost, etc.), and the hardware/software configuration is recorded. This enables a faithful experimentation with more reliable results.

TABLE 1 Experimental platforms used for trace collection.

Name	Nodes	Node Processors	GPUs	Network	Section
idcin2	1	2 × 14-core Intel E5-2697v3	3 × Nvidia Titan X	–	5.2
chifflet	8	2 × 14-core Intel E5-2680v4	2 × Nvidia GTX 1080Ti	Ethernet 10GB	5.3, 5.4

[2] A preliminary summary of this scenario has been described in a previous work published in the 2016 Visual Performance Analysis Workshop, held during the Super Computing conference, under the title of "Analyzing Dynamic Task-Based Applications on Hybrid Platforms: An Agile Scripting Approach" (44). We present a more detailed analysis using a full rewrite of the R code to generate the views.

5.2 | Changing Schedulers and Constraining Tasks to get better Performance

Initial Motivation

The first scenario is based on a composite view with five panels (see Section 4.1 for details on these panels), shown in Figure 12: the Cholesky Iteration (top), the space/time view (middle), submitted and ready (plus scarce) variables (bottom), and the solution of the ABE (right). This choice of panels enables us to effectively compare runtime schedulers for two reasons: the Cholesky Iteration provides a signature of how the Cholesky DAG is traversed; and the remaining panels gives a sense on the performance and anomalies.

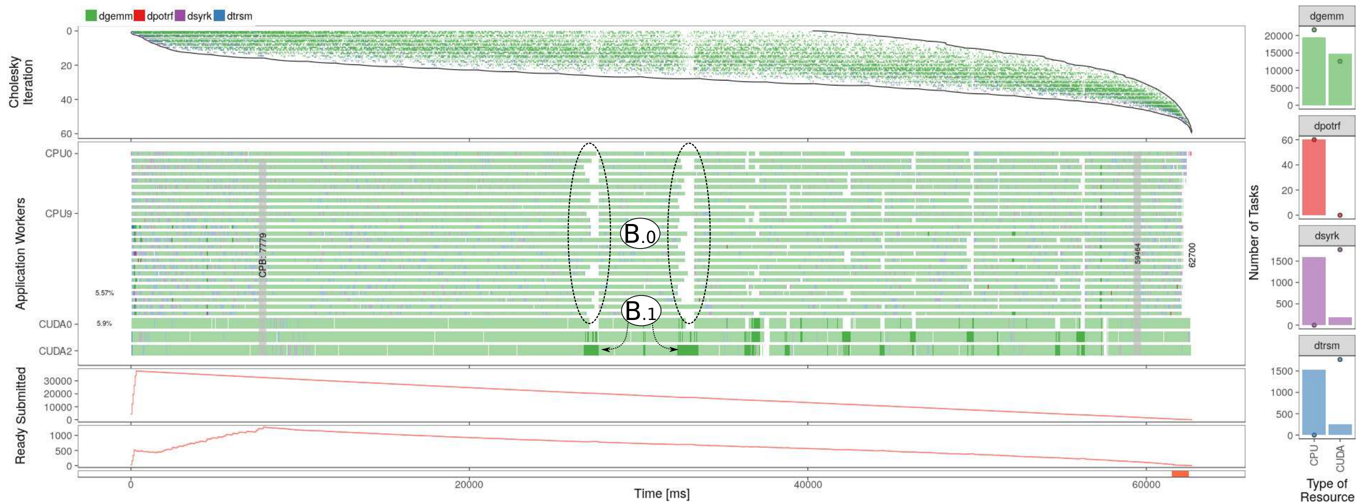


FIGURE 12 Cholesky factorization of a large (60×60 tiles of 960×960) matrix with the DMDAS scheduler with the Cholesky, Space/time, Submitted and Ready (plus scarce) variables, and the solution of the ABE (right).

Figure 12 shows the behavior of the Cholesky factorization of a large (60×60 tiles of 960×960) matrix with the DMDAS scheduler using 25 CPU cores (Intel Xeon E5-2697v3) and three identical Nvidia Titan X GPUs of the IdCin2 node. One can hope for a 5% improvement because the makespan is 62,700 ms while the ABE is 59,464 ms. The scheduling seems indeed inefficient since there are periods (white areas in CPUs, regions B.0) without any useful computation. These periods correspond to filtered states (not shown for clarity) where threads try to actively fetch data. The total idleness for CPUs varies from 3 to 6%, while for GPUs it ranges from 2 to 6%. This GPU inactivity is likely the main source of potential improvement, even if the idleness rate is similar to that shown on CPUs since GPU delivers a higher GFlops rate. It is clear, looking at the Ready panel, that this idle time does not come from a sudden lack of tasks ready to be executed. The submitted panel clearly indicates that all tasks have been submitted in the beginning and that task execution started immediately after, without waiting for fully unrolling the DAG. As suggested in the Cholesky Iteration (top), DAG traversal is rather depth-first. Many outer loop iterations are parallel (the maximum is 30 around 40s of execution), explaining why there is always a sufficient number of ready tasks.

Such GPUs starvation is more likely explained either by data prefetching problem (some tasks are ready but their input data is not yet transferred to GPUs) or possibly by some priority problem (the priorities, used by the scheduler to choose which task to schedule first when several of them are ready, might be inadequate). The first explanation is likely to be the right one here. Indeed, most large idle periods on GPUs and large periods of times where CPUs are not doing useful computations (in white) also coincide with abnormally long `dgemm` tasks (in dark green, B.1) on GPUs. An investigation has revealed that, for an unknown reason, the GPUs seem to freeze during a task execution inside the proprietary CUBLAS `dgemm` kernel, ultimately blocking tasks eagerly waiting for GPU data. Understanding why GPUs sometimes get stuck would certainly solve the issue but this clearly suggests a weakness of the chosen scheduler which assumes that tasks duration have small variability. Using **other schedulers may therefore alleviate this**.

The plot in the right of Figure 12 show the ideal allocation when calculating the ABE. They show how the GPUs have been overused with `dgemm` tasks and under-exploited for `dsyrk` and `dtrsm` tasks. It therefore suggests to **constrain the `dsyrk` and `dtrsm` tasks to run exclusively on GPUs**.

Performance gains when constraining some tasks to GPUs were already reported by Lima *et al.* (46). However, their results were achieved using scheduler hints provided by programmer annotations. In our case, the suggestion of when and which tasks to constrain to GPUs is inferred from the solution of the ABE without relying on programmer's knowledge about task's architecture affinity.

Comparing Scheduling Strategies and Task Constraints

The initial motivation lead us to vary the scheduler (DMDA, DMDAS, WS) and to force or not the `dsyrk`/`dtrsm` allocation on GPUs. Figure 13 provides the six-scenario comparison: the columns represent the different schedulers: DMDA (left), DMDAS (center), and WS (right); while the rows represent the original version (top) and the modified version where `dsyrk`/`dtrsm` tasks can only be executed on GPUs (bottom). The ABE for each case is depicted on each cell of the comparison matrix.

First of all, it is interesting to see, as shown by the Cholesky Iteration plot, how the three schedulers differ in their traversal of the DAG. While the DMDA algorithm has a breadth-first traversal (very few iterations of the outer loop are active at the same time), the DMDAS has a much more depth-first traversal as it takes the priority of the critical path into account. The traversal of the Work Stealing (WS) is even more depth-first as almost all outer loop iterations are still in progress at the end of the execution. Such way of progressing through the DAG is typical of WS and somehow favors local data accesses even though the algorithm is more dependency myopic than the two other ones. Second, when constraining the `dsyrk` and `dtrsm` to run solely on the GPUs (the plots on the bottom row of Figure 13), task allocation then corresponds to the ideal one. However, if such constraint allows both DMDAS and Work Stealing to obtain near optimal executions (within less than 2% of the lower bound as given by the ABE), this helped only moderately the DMDA algorithm. Many synchronized idle phases can be observed and imputed to both dependency issues (not enough parallelism is obtained from such a strict breadth-first traversal) and particularly slow tasks (probably slowed down by simultaneous data transfers). Interestingly, very few outlier tasks appear in the DMDAS and WS executions although the latter still seems a bit sensitive to this, as inactivity periods on CPUs (white areas) still correlate with the occurrence of `dgemm` outliers (darker green) on GPUs.

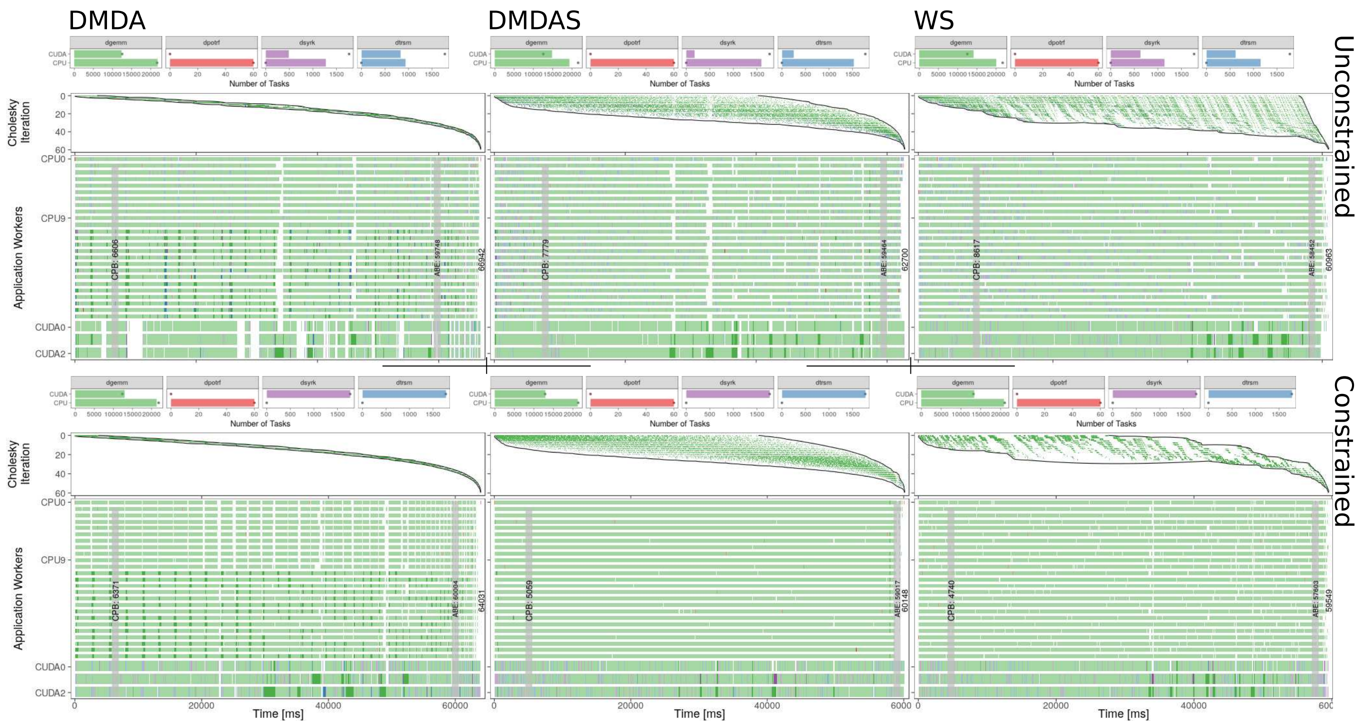


FIGURE 13 Six case comparison showing three different runtime schedulers – DMDA, DMDAS, and WS (columns) – when used with the original version (top) and by forcing `dsyrk`/`dtrsm` tasks allocation on GPUs (bottom). The solution of the Area Bound Estimation (ABE) is shown for each of the six cases.

Finally, we stress that such observations are no coincidence. We randomly ran similar scenarios ten times and although the numbers always slightly differ, the general behavior and conclusions are the same. We also highlight that the area bound estimations (ABE) can vary significantly between two scenarios (e.g., 60s for constrained DMDA vs. 57s for constrained WS), which can be initially surprising since these estimates only depend on the number of tasks and their per-type average execution time on the different resources. The observations can be explained by the use of sample execution time mean, which may vary a bit. From our investigation this variation is not explained by outliers occurrence but rather biased toward one or another scheduler. We think this is the consequence of a better locality (cache usage) but more complex measurements would be needed to fully evaluate this hypothesis.

5.3 | Lack of MPI Message Pipelining with TCP

Initial Motivation

The second scenario investigates idle periods affecting all the workers (CPU/GPU) in the same node when the application starts running with the multi-node StarPU-MPI. These periods are very frequent; we have observed in multiple executions with different worker combinations and number of nodes. It occurs more frequently in the beginning of the application, when the bulk of parallelism is not yet in place, or during the whole execution in runs where the problem size is too small for the number of resources. The problem is illustrated by regions marked by A and B in Figure 14, showing the behavior of Cholesky when a squared matrix of dimension 72,000 is used as input, with 75×75 tiles of size 960×960, and the PRIO scheduler. Two nodes with two GPUs and five cores each have been used in the experiment. The nodes are interconnected by a 10GB Ethernet network properly configured using OpenMPI 2.0.2 with the default TCP eager limit set to 65,536 bytes (indicating that the eager communication protocol is used until that size; afterwards the RDV protocol is adopted). The case represented in the Figure 14 is enriched with one representative backward task dependencies starting at task `0_5051` (in Node 0, red). The backward path (highlighted by the dashed arrow in the plot) indicates that the task `0_5051` has been released by a very long MPI operation (red-border rectangle in the top), which has been deblocked by tasks running in GPUs of Node 1 (bottom). These tasks were released by another MPI operation from Node 0, which has been released to execution by data provided by tasks on Node 0. We have interactively selected many tasks after these idle periods, for different schedulers and number of resources. No matter which configuration is used, tasks are always delayed by long MPI operations.

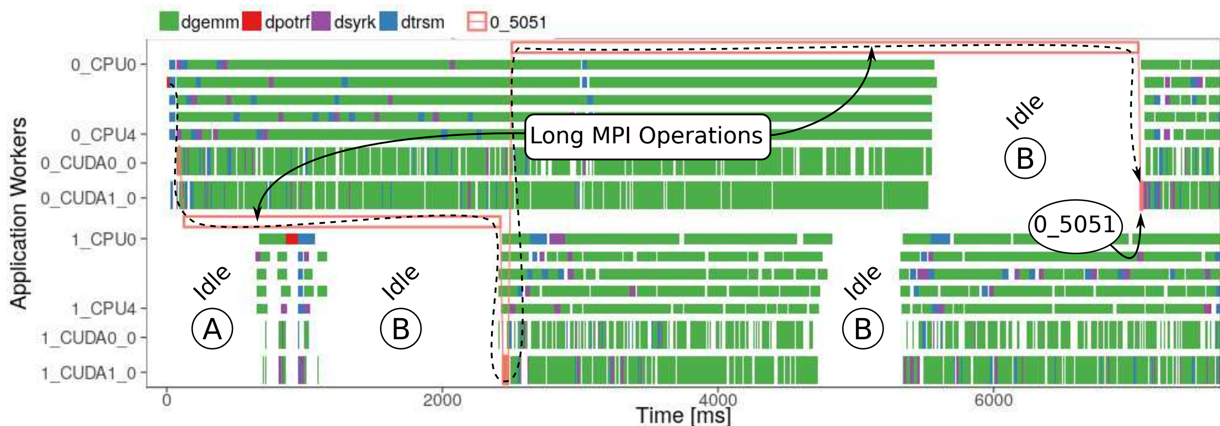


FIGURE 14 Backward task dependencies for the 0_5051 task, illustrating how the idle periods are caused by very long and abnormal MPI operations between the two nodes (2 GPUs, and 5 cores each) reserved to run Cholesky (72,000 square matrix, tiles of 960×960 elements, PRIO scheduler), both interconnected by 10GB Ethernet: idle time is caused by the use of the rendezvous mode instead of eager communication mode, as it was expected by application developers.

Our investigation has indicated that these idle periods are caused by a combination of two factors: first, the way the MPI thread of StarPU handles asynchronous communications during the beginning of the application (idle marked by region A in Figure 14); second, the MPI threshold configuration between the eager and the rendezvous communication modes (regions identified with B). The analysis is conducted on heterogeneous multi-node cases, with a focus on task dependencies, ready/submitted variables,

along with MPI bandwidth and the concurrent MPI operations plot. We detail how each of these factors are fixed to improve the message pipelining implemented in the MPI thread of StarPU.

Fixing the message pipelining to expose parallelism faster in remote nodes

The first problem we have identified is related on how StarPU handles the MPI asynchronous communications (see Section 2.2 for details on how StarPU-MPI works). Since the data partitioning is static, StarPU identifies all inter-node point-to-point communications to satisfy the data dependencies that cross a node border. Since they are known, the per-node MPI thread of StarPU issues multiple `MPI_Isend` and the corresponding `MPI_Irecv` operations from the start. Depending on the input size and the number of tiles involved in a specific run, the amount of these operations might be very large. The problem is that some of these operations might complete (i.e., the communication finishes) before posting all remaining asynchronous operations. When such scenario occurs, the application is delayed because the MPI thread handling the communication operations has not issued an `MPI_Test` to detect the reception and unlock the corresponding tasks. This negative behavior becomes worse when multiple nodes are used and the borders among tiles have more complex configurations.

The fix for the first problem was to interleave `MPI_Test` calls between each issue of MPI asynchronous communications. If the test call indicates that a message has been received, the MPI thread of StarPU can satisfy an inter-node data dependency. Since test calls provide a negligible overhead with potential great benefits in the beginning of the application, this solution is now mainstream in StarPU. After the fix, idle periods such as the one marked by A in Figure 14 disappear. However, the other idle periods (B) remain after the fix, indicating that the origin is elsewhere.

The root cause of the second problem (regions B in Figure 14) has been identified as the threshold to switch from the eager to the rendezvous communication modes. The eager mode allows a send to complete without an acknowledgment from the other side; while the rendezvous mode requires a reception acknowledgment. The change of communication mode is driven by the message size. The default value of the OpenMPI 2.0.2 installation used in all experiments is 64 KBytes. Messages smaller than such size will be sent using the eager mode, favoring asynchronism, while larger messages are sent using the rendezvous, for throughput. In our Cholesky case, the volume of data dependencies depends on the tile size. For example, a commonly used squared tile of 960 8-byte elements occupies ≈7.37 MBytes. This implies that only the rendezvous protocol is used throughout the experiments with the default eager limit. Unfortunately, the rendezvous protocol also introduces communication aggregation: MPI requests submitted closely enough will be sent and received together.

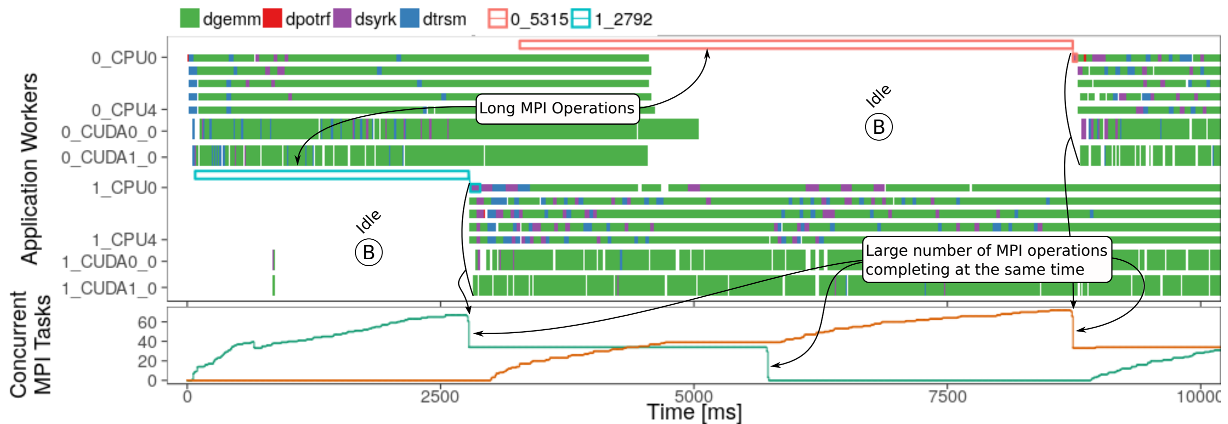


FIGURE 15 Identifying the nocive MPI communication aggregation of rendezvous mode and the correlation with the long MPI operations (using backwards dependencies of two delayed tasks) for a run with two nodes (2 GPUs, and 5 cores each) reserved to run Cholesky (72000 square matrix, tiles of 960×960 elements, with the DMDAS scheduler).

Figure 15 shows a case with the DMDAS scheduler where the long MPI operations during the idle periods for two representative backwards task dependencies get correlated with the end of multiple MPI operations at the same time. We have typically observed this kind of behavior, with as many as 40 7.37-MByte transfers aggregated together that finish at the same time. This

massive network operation may take ≈2.5s to complete altogether, instead of completing progressively. Such undesired behavior has been reported to the OpenMPI team, which agreed something needs to be fixed. This behavior is harmless during most of the execution, except in the beginning where little parallelism exists: the execution starts with a single `POTRF` task in the first node, followed by `TRSM` tasks, whose results need to be transferred to other nodes as quickly as possible, because all tasks from other nodes depend on these first tasks to start unrolling their DAG. Good schedulers tend to execute the `TRSM` tasks as quickly as possible, but that leads to submitting MPI requests very closely, and thus seeing them all aggregated, and thus received very late. The work-around proposed by the OpenMPI team is to force the eager mode, to avoid aggregation and instead get progressive reception and thus better reactivity, even if it leads to lower network efficiency, since clearly the beginning is very sensitive to pipelined delivery.

The fix for the second problem is to increase the eager limit to a value that encompass the tile size in bytes, enabling an asynchronous exchange for all data dependencies. We have increased the tile size to 8MBytes to confirm that a higher eager limit (using the `btl_tcp_eager_limit` option of OpenMPI) brings performance gains. The case used to illustrate is depicted in Figure 16, showing two executions of Cholesky with an input size of 67,200, tiles of 960, and static partitioning by row ($P = 1$). The LWS (work-stealing) scheduler has been used by StarPU. The figure shows the behavior difference (left/right) by changing the eager limit parameter: the original behavior is shown in the left; increasing the eager limit gives the behavior depicted in the right. We can see that the idle time in StarPU workers is reduced (from a 30-50% to 10-40%) and the large idle periods shared by all workers of the same node have disappeared. The left image has been generated with four backward dependencies crossing node boundaries with MPI, illustrating that the idle times are caused by abnormal long communication times. The parallelism explosion can be also verified in the ready tasks panel, where a high number of ready tasks is reached very quickly and sustained for a long time. The Cholesky Iteration shows that the number of tiles being processed at the same time is much higher. The MPI bandwidth performance (not depicted) is higher after the eager limit modification. Such changes enable a 20% execution time reduction, without any application change.

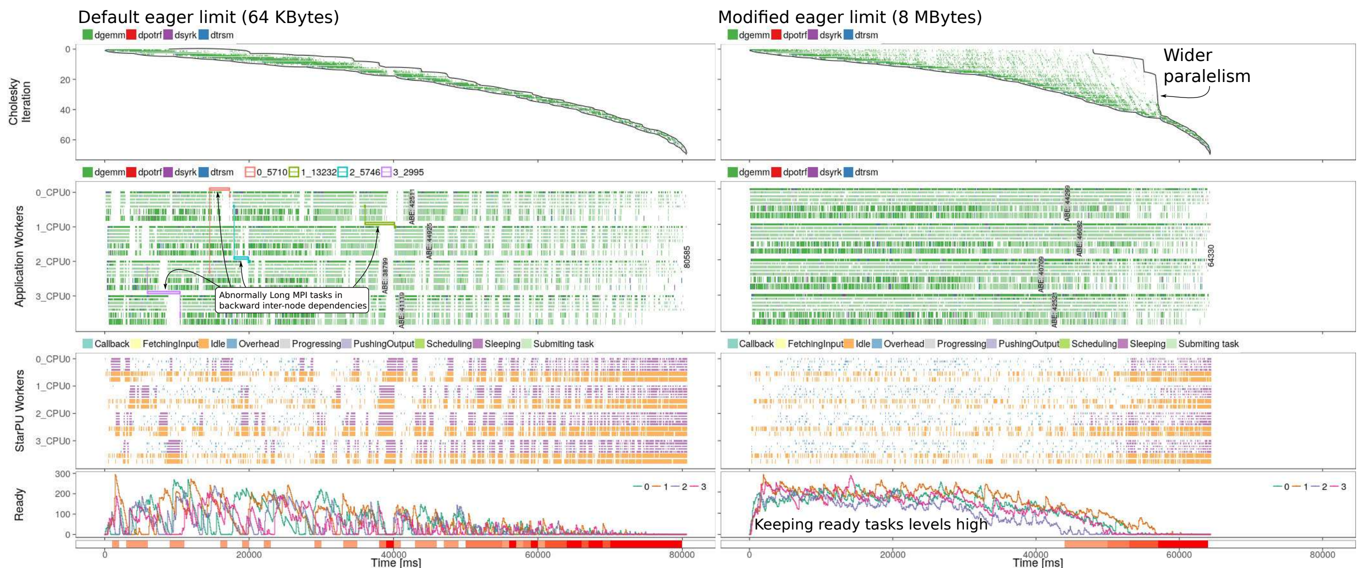


FIGURE 16 Cholesky execution using a squared matrix of 67,200, with squared tiles of 960 with four identical nodes, each one with 2 GPUs and 5 cores, LWS scheduler. The left image shows the behavior when the application executes with the default eager limit, obtaining a total makespan of ≈80s. It is annotated with four inter-node backward MPI dependencies showing abnormal long times. The right image (same scale) shows an unchanged case but with a larger eager limit, for a makespan of ≈64s (performance gain of 20%).

5.4 | Tuning StarPU's MPI requests and effect on data distribution strategies

Initial Motivation

Using 8 hybrid nodes (**chifflet** cluster), with 6 cores and 2 GPUs on each node (for a total of 48 cores and 16 GPUs), we study the influence of data distribution (P $\in \{1, 2, 4, 8\}$, see Figure 2) on the load balance and resulting performance. We have tested with two schedulers (LWS and DMDAS), a fixed input of 108,000×108,000 with tiles of 1,440×1,440 elements, and using the best known network configuration (Ethernet 10GB with appropriate kernel configuration, and an MPI eager limit which is higher than the tile size - in this case 32MBytes). We report here only the results obtained with the LWS scheduler, but the benefits brought by changes motivated by our analysis are scheduler independent. Per-node ABE is used as theoretical lower bound, and we use our framework to understand why makespans are disconnected from such bounds. With eight nodes (as presented in Section 2.3), it is expected to have the best performance with a P value between two and three, since this value minimizes the communication perimeter of the nodes.

StarPU's limit for simultaneous MPI requests and data distribution strategies

Figure 17 shows the visualization of the four cases (P $\in \{1, 2, 4, 8\}$) depicting with the Cholesky, application worker, ready tasks, MPI bandwidth and the number of concurrent MPI operations panels. The per-node area bound estimations (marked by ABE squared rectangles) shows that load balancing improves as we increase the value of P, up to the best case with P=8. The overall performance, represented by the makespan, shows that the performance is not totally related to a better load balancing. Better results are achieved with P=2 (top right) with a total makespan of $\approx$110 s, and P=4, $\approx$113 s, despite the fact that the load is unbalanced as shown by the per-node ABE. The reason behind the better performance of $P = 2, 4$ is that they show the largest number of tiles being computed in parallel, as shown by the Cholesky Iteration, have the lowest CPU/GPU idleness, and the MPI bandwidth is smoother and flat along the execution time. These facts demonstrate that unlocking parallelism on other nodes as early as possible is more important than a very good load distribution.

There are many negative observed factors for cases with P$\in \{1, 8\}$. The P=1 case is particularly interesting since it demonstrates the largest load imbalance among the four cases. The unequal load distribution force nodes to wait from each other, i.e., the slowest node with the largest load limits the performance of other nodes. This is confirmed through the ups and dows in both MPI bandwidth and number of concurrent MPI operations, like a ping-pong effect. This might indicate a rather sequential execution as shown by the low number of ready tasks along execution. The case with P=8 shows the highest idleness, almost 30% in CPUs and 60% in GPUs, and it is the case where the slow start is more representative, even after using the eager protocol in the MPI layer (see previous section). This is explained by the misplacement of initial `dtrsm` tasks, responsible for unlocking more parallelism. They are not evenly distributed among nodes, despite the better total load balance. This case is still very intriguing because of the small 20s window between 25s and 45s where a lot of parallelism is released while the remaining of the execution (before and after) suffers from lack of it. Before that time interval, the runtime is incapable of unlocking enough parallelism to feed all cores because of the column-based distribution, which makes e.g., node 0 responsible for computing all tasks of the column before letting other nodes to have some work to do. Moreover, it forces node 0 to send a lot of data through the network, as shown in the MPI curve. During and after that 20s-window, the column-based distribution keeps making one after the other responsible for unlocking all parallelism. As long as there are enough iterations computed at the same time, the runtime manage to keep up, but at some point the runtime is incapable of unlocking parallelism fast enough because of the data distribution, and performance is degraded with a very low number of ready tasks.

No matter which P value is used to equal load distribution, observed makespans are still very far from what the area bound estimations predict. We believe that matching ABE is very hard, since it disregards communication costs, but the scheduling and inter-node (MPI) communication could be optimized. The origin of this performance problem is related to the number of concurrent MPI tasks, seen in the bottom of each case of Figure 17. This in itself is not a problem since these operations are sometimes carried out by independent pairs of nodes employing independent network links. However, when the communication concurrency is too high, this might delay MPI operations that would unlock parallelism faster. What happens is that since MPI processes requests in the order they were submitted by the runtime, the MPI communications needed by the critical path get delayed by all the previously-submitted requests. This can be observed for the P=8 case, where the first node demonstrates an enormous amount of concurrent MPI operations, delaying all communications by as much corresponding time. Before our investigation, the StarPU runtime provided no API or configuration to define communication priorities per-task, nor to control the maximum number of concurrent MPI operations.

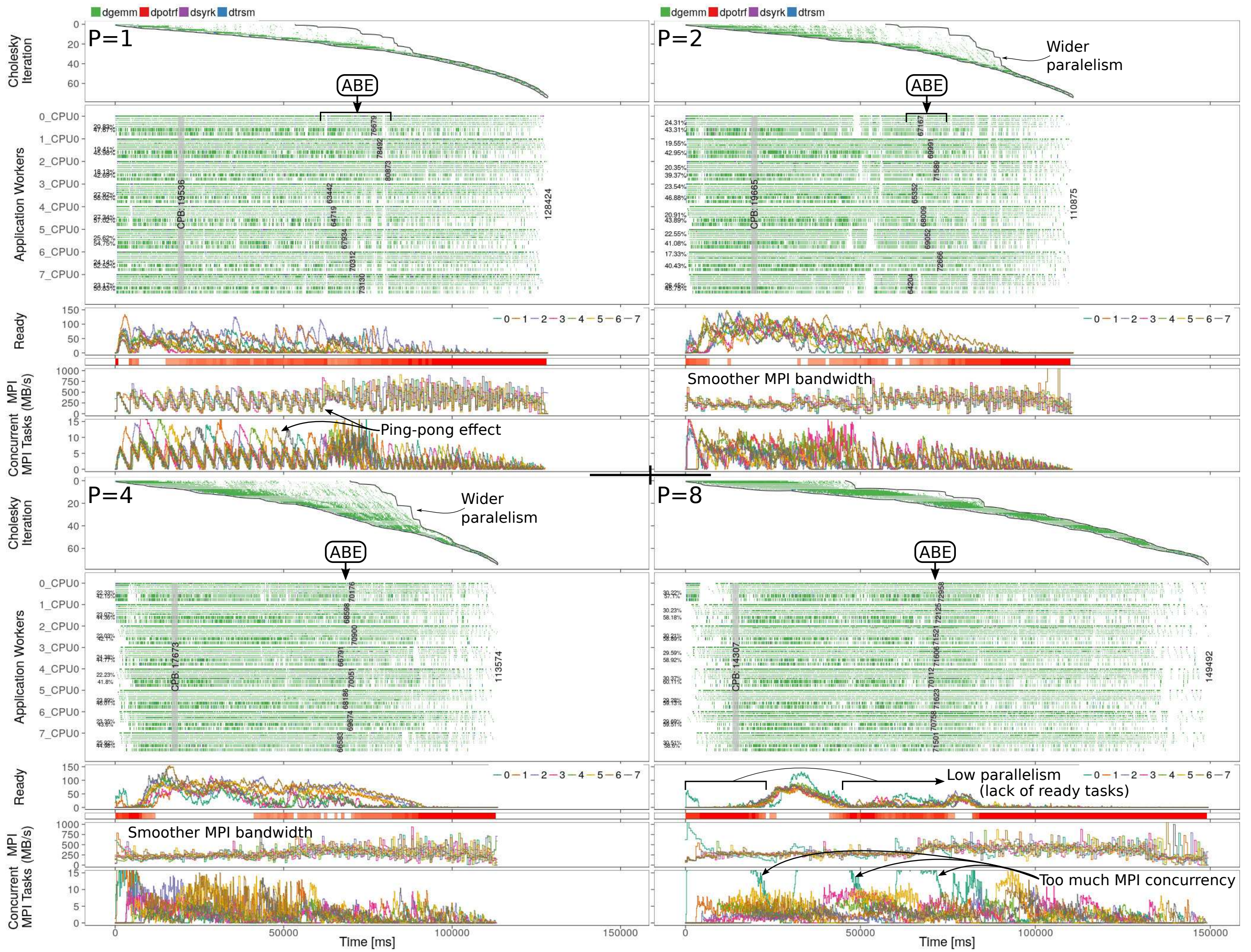


FIGURE 17 Cholesky execution using a squared matrix of 108,000, with squared tiles of 1,440 with eight identical nodes, each one with 2 GPUs and 6 cores, LWS scheduler. The grid shows four cases for P values of 1 (top left), 2 (top right), 4 (bottom left), and 8 (bottom right). All plots share the same scale (time, vertical).

We believe that a better control of MPI operations can be implemented in two ways in StarPU: (a) to give a higher priority to those communications that release parallelism in Cholesky (e.g., the first `dtrsm` tasks), and (b) to impose a limit on the number of MPI requests issued by StarPU, so that high-priority requests are delayed at worse by the few requests already issued. These two strategies have been implemented in the development branch of StarPU after we have identified the problem. Figure 18 depicts the scheduling behavior for two representative scenarios (LWS scheduler, P=2, 100×100 tiles of 960×960 elements) for two runs: (left) before such modifications and (right) after, using a limit of 10 concurrent MPI requests and giving `dtrsm` higher communication priority (simply inherited from the task priority, so without application modification). After the changes were introduced, we can see that MPI delivers a higher bandwidth for the application, especially in the crucial starting moments where parallelism is being unfolded. The more controlled MPI requests can be verified in the bottom plot, where the number of concurrent MPI operations is much more contained. The Cholesky Iteration plot shows that the parallelism unfolds much faster, in less than 10s, after the implemented changes. This behavior is confirmed by the ready tasks plot, where the number of ready tasks per node (colors) responsible for unfolding parallelism reaches a peak in the beginning of the application, while before (left) such behavior caused a minor yet damaging slow start of the application. This has the direct benefit of exposing in a much faster way the critical path towards application completion, and gives much more scheduling opportunities to improve performance. In conclusion, as can be observed in this comparison, these modifications reduce the total makespan by ≈8.5% (from 80s to 73s). Using the DMDAS scheduler, the gains are of ≈12.7% (from 76s to 67s).

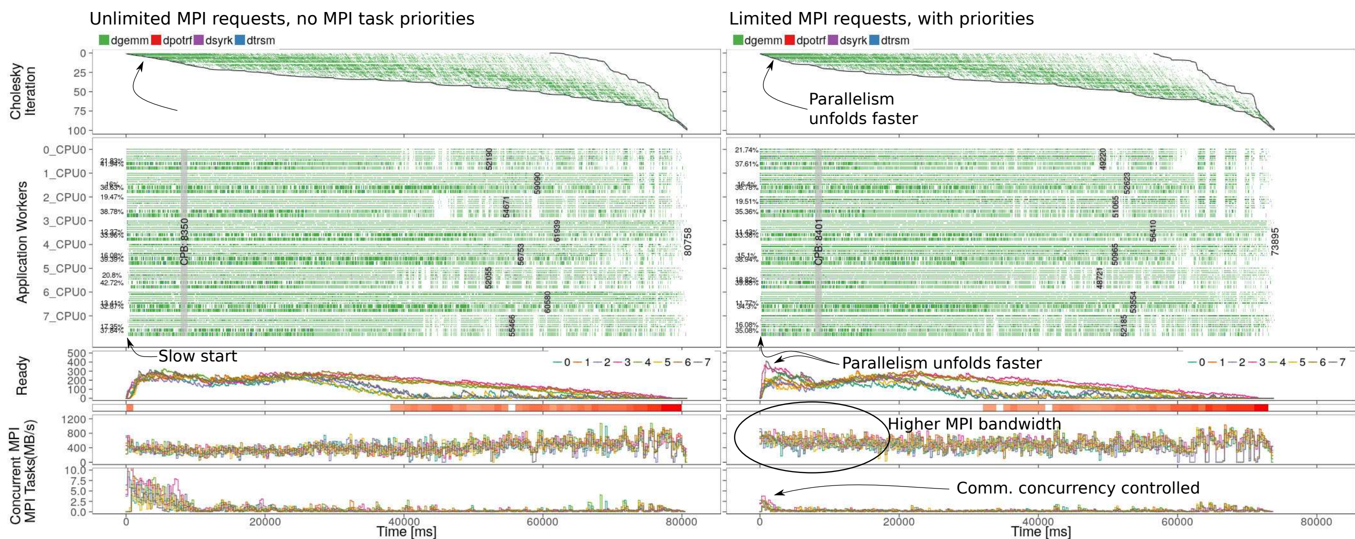


FIGURE 18 Cholesky execution using a squared matrix of 96,000, with 100 × 100 squared tiles of 960 with eight identical nodes, each one with 2 GPUs and 6 cores. The configuration of P=2, the vertical/horizontal scales, and the LWS scheduler are shared by both cases: (left) original runtime with unlimited MPI requests, without communication priorities; (right) MPI requests limited to 10, using priority for the dependencies leading to `dtrsm` tasks.

6 | CONCLUSION AND FUTURE WORK

This article presents a flexible and versatile framework to create faithful composite trace views for the performance analysis of task-based HPC applications running on multi-(node, core, GPU) platforms. The framework is built using modern data science tools such as R, `ggplot2` and the data manipulation functions of the `tidyverse` meta-package. Many other blocks are built using compiled languages for performance, to complete a scalable 2-phase workflow to deal with traces. We have shown the usefulness of this framework in the performance analysis of an already very optimized tiled Cholesky Factorization from the Chameleon/MORSE suite that uses the StarPU-MPI runtime. The optimizations that were motivated by our analysis are three-fold. First, it enabled changes in the application to force a better task equilibrium between cores and GPUs (using the solution suggested by an Area Bound Estimation). Second, to fix the application slow start, i.e., the lack of parallelism in the first seconds of execution when the DAG unfolds – by using the MPI eager mode and the preference for a communication algorithm that reduces latency. Third, to improve parallelism unfolding in all phases of the execution by limiting the number of MPI requests issued by the StarPU runtime when registering the inter-node static dependencies among task, when the domain is partitioned among nodes. These three strategies together bring several benefits for the applications based on StarPU-MPI. For the specific case of Cholesky factorization, the third solution alone brings a 13% makespan reduction when using the DMDAS scheduler.

As future work, we intend to exploit task dependency filtering further to pin-point scheduling mistakes that reduce performance. We also want to adapt our software to deal with variant of task-based programming models such as OpenMP that allows a running task to suspend itself until some other tasks complete. We will also investigate the behavior of more complex applications, such as qr_mumps or ScalFMM whose structure and computation kernels are particularly heteregeneous. We are also interested by the creation of a performance model for bimodal task duration distribution when running on GPUs, to improve our outlier detection mechanism.

ACKNOWLEDGMENTS

We thank CAPES/Cofecub 764-13, FAPERGS/Inria ExaSE, FAPERGS 16/2551-0000 354-8 (PPP 2014), FAPERGS 16/2551-0000 488-9 (PRONEX 2014), CNPq 447311/2014-0, CNRS/LICIA Intl. Lab, the EU H2020 Programme and the MCTI/RNP-Brazil under the HPC4E Project, grant 689772. Some experiments were carried out at the Grid'5000 platform (https://www.grid5000.fr), with support from Inria, CNRS, RENATER and several other organizations. Finaly, we thank Rémy Drouilhet who provided us with an efficient Rcpp implementation of trace aggregation at a fixed granularity. The companion material

of this paper, including experimental details, source code and code snippets to regenerate figures, is hosted by CNRS's IN2P3 (http://www.in2p3.fr/), for which we are also grateful.

References

[1] Meuer Hans Werner, Strohmaier Erich, Dongarra Jack, Simon Horst D.. *The TOP500: History, Trends, and Future Directions in High Performance Computing*. Chapman & Hall/CRC; 1st ed.2014.

[2] Ayguadé Eduard, Copty Nawal, Duran Alejandro, et al. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems.* 2009;20(3):404–418.

[3] Duran Alejandro, Ayguadé Eduard, Badia Rosa M, et al. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Par. Proc. Letters.* 2011;21(02).

[4] Bosilca George, Bouteiller Aurelien, Danalis Anthony, Faverge Mathieu, Hérault Thomas, Dongarra Jack J. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering.* 2013;15(6):36–45.

[5] Augonnet Cédric, Thibault Samuel, Namyst Raymond, Wacrenier Pierre-André. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Conc. and Comp.: Pract. and Exp.*. 2011;23(2).

[6] Pillet V., Labarta J., Cortes T., Girona S.. PARAVER: A Tool to Visualize and Analyze Parallel Code. In: Nixon Patrick, ed. *Proceedings of WoTUG-18: Transputer and occam Developments*, :17–31; 1995.

[7] Coulomb Kevin, Faverge Mathieu, Jazeix Johnny, et al. Visual trace explorer (ViTE) .

[8] Isaacs Katherine E, Bremer Peer-Timo, Jusufi Ilir, et al. Combing the communication hairball: Visualizing parallel execution traces using logical time. *IEEE transactions on visualization and computer graphics.* 2014;20(12):2349–2358.

[9] Knüpfer Andreas, Brunst Holger, Doleschal Jens, et al. The Vampir performance analysis tool-set. In: Springer 2008 (pp. 139–155).

[10] Schulte Eric, Davison Dan, Dye Thomas, Dominik Carsten. A Multi-Language Computing Environment for Literate Programming and Reproducible Research. *J. of Stat. Soft.*. 2012;46(3).

[11] Agullo E., Bosilca G., Bramas B., et al. Poster: Matrices over Runtime Systems at Exascale. In: :1332-1332; 2012.

[12] Augonnet Cédric, Aumage Olivier, Furmento Nathalie, Namyst Raymond, Thibault Samuel. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In: EuroMPI'12:298–299Springer-Verlag; 2012; Berlin, Heidelberg.

[13] Agullo Emmanuel, Demmel Jim, Dongarra Jack, et al. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series.* 2009;180(1).

[14] Bosilca George, Bouteiller Aurelien, Danalis Anthony, Herault Thomas, Lemarinier Pierre, Dongarra Jack. DAGuE: A generic distributed {DAG} engine for High Performance Computing. *Parallel Computing.* 2012;38(1–2):37 - 51. Extensions for Next-Generation Parallel Programming Models.

[15] Ohshima Satoshi, Katagiri Satoshi, Nakajima Kengo, Thibault Samuel, Namyst Raymond. Implementation of FEM Application on GPU with StarPU. In: SIAM; 2013; Boston, United States.

[16] Martínez Víctor, Michéa David, Dupros Fabrice, et al. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In: IEEE; 2015.

[17] Agullo E., Giraud L., Guermouche A., Nakov S., Roman J.. Task-Based Conjugate Gradient: From Multi-GPU Towards Heterogeneous Architectures. In: Desprez Frédéric, Dutot Pierre-François, Kaklamanis Christos, et al. , eds. *Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*, Cham: Springer International Publishing 2017 (pp. 69–82).

[18] Lacoste X., Faverge M., Bosilca G., Ramet P., Thibault S.. Taking Advantage of Hybrid Systems for Sparse Direct Solvers via Task-Based Runtimes. In: :29-38; 2014.

[19] Carpaye Jean Marie Couteyen, Roman Jean, Brenner Pierre. Design and analysis of a task-based parallelization over a runtime system of an explicit finite-volume CFD code with adaptive time stepping. *Journal of Computational Science.* 2017;.

[20] Christou Michalis, Christoudias Theodoros, Morillo Julián, Alvarez Damian, Merx Hendrik. Earth system modelling on system-level heterogeneous architectures: EMAC (version 2.42) on the Dynamical Exascale Entry Platform (DEEP). *Geoscientific Model Development.* 2016;9(9):3483-3491.

[21] Graham R. L.. Bounds for Certain Multiprocessing Anomalies. *Bell System Technical Journal.* 1966;45(9):1563–1581.

[22] Topcuoglu H., Hariri S., Wu Min-You. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Par. Distr. Syst.*. 2002;13(3):260-274.

[23] Blumofe Robert D., Leiserson Charles E.. Scheduling Multithreaded Computations by Work Stealing. *J. ACM.* 1999;46(5):720–748.

[24] Blackford L. S., Choi J., Cleary A., et al. *ScaLAPACK user's guide*. Society for Industrial and Applied Mathematics; 1997.

[25] Gropp W., Hoefler T., Thakur R., Lusk E.. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. Computer science & intelligent systemsMIT Press; 2014.

[26] Wilson James M.. Gantt charts: A centenary appreciation. *European Journal of Operational Research.* 2003;149(2):430–437.

[27] Schnorr Lucas Mello, Faverge Mathieu, Trahay François, Oliveira Stein Benhur, Kergommeaux Jacques Chassin. *The Paje trace file format.* : UFRGS; 2016.

[28] Dietrich Robert, Winkler Frank, William Thomas, Stolle Jonas, Henschel Robert, Berry Donald K. A Case Study: Holistic Performance Analysis on Heterogeneous Architectures using the Vampir Toolchain.. In: :793–802; 2013.

[29] Brendel Ronny, Heyde Michael, Brunst Holger, Hilbrich Tobias, Weber Matthias. Edge bundling for visualizing communication behavior. In: VPA:1–8; 2016.

[30] Haugen Blake, Richmond Stephen, Kurzak Jakub, Steed Chad A., Dongarra Jack. Visualizing Execution Traces with Task Dependencies. In: VPA '15:2:1–2:8ACM; 2015; New York, NY, USA.

[31] Huynh An, Thain Douglas, Pericàs Miquel, Taura Kenjiro. DAGViz: A DAG Visualization Tool for Analyzing Task-parallel Program Traces. In: VPA '15:3:1–3:8ACM; 2015; New York, NY, USA.

[32] Keller Rainer, Brinkmann Steffen, Gracia José, Niethammer Christoph. Temanejo: Debugging of Thread-Based Task-Parallel Programs in StarSs. In: Brunst Holger, Müller S. Matthias, Nagel E. Wolfgang, Resch M. Michael, eds. *Tools for High Performance Computing 2011: Proceedings of the 5th International Workshop on Parallel Tools for High Performance Computing, September 2011, ZIH, Dresden*, Springer 2012 (pp. 131–137).

[33] Wininger Florian, Ezzati-Jivan Naser, Dagenais Michel R.. A Declarative Framework for Stateful Analysis of Execution Traces. *Software Quality Journal.* 2017;25(1):201–229.

[34] Agullo Emmanuel, Beaumont Olivier, Eyraud-Dubois Lionel, et al. Bridging the Gap Between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms. In: IPDPSW '15:34–45IEEE Computer Society; 2015; Washington, DC, USA.

[35] Schnorr Lucas M, Legrand Arnaud. Visualizing More Performance Data Than What Fits on Your Screen. In: Springer 2013 (pp. 149–162).

[36] R Core Team . R: A Language and Environment for Statistical Computing. R Foundation for Statistical ComputingVienna, Austria2017.

[37] Wickham Hadley. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York; 2009.

[38] Wickham Hadley. tidyverse: Easily Install and Load 'Tidyverse' Packages2016. R package version 1.0.0.

[39] Sievert Carson, Parmer Chris, Hocking Toby, et al. plotly: Create Interactive Web Graphics via 'plotly.js'2016. R package version 4.5.6.

[40] Agullo Emmanuel, Buttari Alfredo, Guermouche Abdou, Lopez Florent. Implementing Multifrontal Sparse Solvers for Multicore Architectures with Sequential Task Flow Runtime Systems. *ACM Trans. Math. Softw.*. 2016;43(2):13:1–13:22.

[41] Danjean Vincent, Namyst Raymond, Wacrenier Pierre-André. An Efficient Multi-level Trace Toolkit for Multi-threaded Applications. In: Euro-Par'05:166–175Springer-Verlag; 2005; Berlin, Heidelberg.

[42] Wickham Hadley. feather: R Bindings to the Feather 'API'2016. R package version 0.3.1.

[43] Wilkinson Leland. The grammar of graphics. In: Springer 2012 (pp. 375–414).

[44] Garcia Pinto Vinicius, Stanisic Luka, Legrand Arnaud, Mello Schnorr Lucas, Thibault Samuel, Danjean Vincent. Analyzing Dynamic Task-Based Applications on Hybrid Platforms: An Agile Scripting Approach. In: ; 2016; Salt Lake City, United States. Held in conjunction with SC16.

[45] Gamblin Todd, LeGendre Matthew, Collette Michael R., et al. The Spack Package Manager: Bringing Order to HPC Software Chaos. In: SC '15:40:1–40:12ACM; 2015; New York, NY, USA.

[46] Lima João V.F., Gautier Thierry, Danjean Vincent, Raffin Bruno, Maillard Nicolas. Design and analysis of scheduling strategies for multi-CPU and multi-GPU architectures. *Parallel Computing.* 2015;44:37 - 52.

How cite this article: V. Garcia Pinto, L. Mello Schnorr, L. Stanisic, A. Legrand, S. Thibault, and V. Danjean (2017), A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters, *Concurrency and Computation: Practice and Experience*, *TBD*.