StarVZ: Performance Analysis of Task-Based Parallel Applications

Lucas Leandro Nesi, Vinicius Garcia Pinto, Marcelo Cogo Miletto, Lucas Mello Schnorr

Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

StarVZ: Performance Analysis of Task-Based Parallel Applications

Lucas Leandro Nesi
UFRGS
Univ. Grenoble Alpes, CNRS, Inria

Vinicius Garcia Pinto
UFRGS
Univ. Grenoble Alpes, CNRS, Inria

Marcelo Cogo Miletto
UFRGS

Lucas Mello Schnorr
UFRGS

Abstract

High-performance computing (HPC) applications enable the solution of compute-intensive problems in feasible time. Among many HPC paradigms, task-based programming has gathered community attention in recent years. This paradigm enables constructing an HPC application using a more declarative approach, structuring it in a direct acyclic graph (DAG). The performance evaluation of these applications is as hard as in any other programming paradigm. Understanding how to analyze these applications, employing the DAG and runtime metrics, presents opportunities to improve its performance. This article describes the **StarVZ** R-package available on CRAN for performance analysis of task-based applications. **StarVZ** enables transforms runtime trace data into different visualizations of the application behavior. An analyst can understand their applications' performance limitations and compare multiple executions. **StarVZ** has been successfully applied to several study-cases, showing its applicability in a number of scenarios.

Keywords: performance analysis, high-performance computing, trace visualization, task-based applications, runtime systems, heterogeneous CPU/GPU computing, **tidyverse**, **ggplot2**, R.

1. Introduction

High-performance computing (HPC) is becoming more heterogeneous with a combination of traditional multi-core processors with multiple general-purpose graphical processing units (GPGPU) in computing nodes. Such combinations represent 70% in the first ten systems listed in the TOP500 list (Strohmaier *et al.* 2020). Parallel applications running on these computer systems are increasingly complex, with an irregular workload on top of different architectures with varying performance capabilities.

The performance analysis of parallel applications frequently uses timestamped events in execution traces to identify bottlenecks, load imbalance, and communication problems. While some tools like **Paje** (de Kergommeaux *et al.* 2000), **Paraver** (Pillet *et al.* 1995) and **Vite** (Coulomb *et al.* 2012) have extensibility features, others, like **Vampir** (Knüpfer *et al.* 2008) and **Projections** (Wilson 2003), target specific programming environments. Most of them fail to be adaptable because they rely on compiled languages, demanding users to consider scripting languages to test performance analysis hypothesis.

Task-based parallel applications consist of a directed acyclic graph (DAG) of tasks with edges representing task dependencies. A runtime system is responsible for scheduling the DAG to the computing resources, respecting such dependencies. Consequently, the application performance depends on the parallelism of the DAG and the scheduling heuristic of the runtime. Traditional tools for such parallel applications are generally unfit because they lack support for a DAG-oriented analysis. The existing solutions, such as **DAGViz** (Huynh *et al.* 2015), **Grain Graphs** (Muddukrishna *et al.* 2016), **Temanejo** (Brinkmann *et al.* 2013), **TaskInsight** (Ceballos *et al.* 2018), remains monolithic, built on top of compiled languages.

Combining factors above – solely the availability of monolithic tools and the lack of script-based performance analysis for parallel applications – motivate the necessity of a modern workflow based on established data science tools. The requirements for such workflow include the possibility of analyzing large execution traces gathered in different levels of the computing system (application, runtime, operating system), and the freedom to extend the tools according to each parallel application's specific necessities.

This paper presents the **StarVZ** package, a framework, and toolkit, implemented in the R programming language, tailored for the performance analysis of task-based parallel applications. Since the package relies on established tools from the **tidyverse** meta-package, all internal data and views can be exported with the **patchwork** package (Pedersen 2020) to fit the users' needs. As far as we know, **StarVZ** is the first R package dedicated to the performance analysis of task-based parallel applications using execution traces. More than that, it is probably the first framework that employs the flexibility of scripting languages with the performance delivered by the **dplyr** package to deliver synchronized panels built with the **ggplot2** package. Our experience with multiple case studies from parallel task-based linear algebra solver and other types of applications serves as validation of our strategy.

Section 2 presents the background concepts about HPC and task-based programming. Section 3 presents the methodology and philosophy of the **StarVZ** package, while Section 4 details the available visualization panels. Section 5 presents case studies to illustrate how the package can be used to conduct performance analysis.

2. Background concepts

This section presents background concepts on high-performance computing and task-based parallel programming. These concepts are useful to understand the applicability of the **StarVZ** package, its requirements, and design choices.

2.1. High-performance computing landscape

Parallelism is a long-established strategy to increase the processing power of computing systems. It has been part of the hardware-design of general-purpose CPU architectures through programmer-transparent techniques as the instruction level parallelism (ILP) (Hennessy and Patterson 2017). Higher-levels of parallelism is present in the development of scientific applications as weather forecasting and computational fluid dynamics (CFD). Their characteristics favored the use of what is called high-performance computing. The HPC community has developed programming models as the message passing interface (MPI) (Gropp *et al.* 2014) to execute computations on physically distributed computers and the OpenMP specification (Chapman *et al.* 2008) to explore different processors in the same computing node.

The HPC landscape has changed in the last decade. While CPUs with an increasing number of multiple processing cores have become ubiquitous, power constraints have motivated the use of hardware accelerators as Graphical Processing Units (GPU). A GPU is a specialized processing device with its computing cores and memory acting as a hardware accelerator. Current GPUs can process much more than graphical workloads and can carry out general-purpose parallel processing as linear algebra operations. An analysis of the TOP500 list (Strohmaier *et al.* 2020), which ranks the powerful supercomputers globally, shows the growing presence of hybrid platforms combining accelerators with multicore processors.

Widely-spread parallel programming tools target only one parallelism layer at once. Despite many research efforts, the HPC software landscape remains an assortment of independent tools. The MPI specification is a *de facto* standard when handling parallelism in distributed platforms, i.e., a cluster of independent computers cooperating to solve a problem by communicating through a local network. MPI includes operations to exchange point-to-point or collective messages among processes. The specification supports C and Fortran, but there are binds for other programming environments as R (Yu 2002). OpenMP is the reference model when exploring parallelism among processor cores sharing a single memory space. As MPI, OpenMP focuses on C/C++ and Fortran. CUDA (NVIDIA 2020) is the most used tool to program general-purpose applications that explore GPUs while OpenCL (OpenCL Working Group 2020) offers a vendor-independent approach. Both CUDA and OpenCL focus on C, but only OpenCL offers a bind to R (Urbanek and Puchert 2020).

HPC software stack frequently comprises an *ad hoc* combination of independent programming tools with hand-made optimizations and weak performance portability. Since the hardware platforms continue to become more heterogeneous, there is room for better software abstractions. The task-based model, discussed in the next section, is an example of programming abstraction for heterogeneous platforms.

2.2. Task-based programming

When implementing an algorithm in the task-based programming model, the programmer breaks down the program in small portions of work called *tasks*, potentially executed in parallel while respecting dependencies among them. A runtime system handles load balancing and scheduling at execution time, effectively decoupling application code from the target hardware. **Cilk** (Blumofe *et al.* 1996) was one of the first tools to implement this concept by adding two keywords (`spawn` to and `sync)` to a standard C code. A `spawn` allows a function to be executed in parallel, while `sync` acts as a local barrier to wait for the end of local `spawn`ed functions. The OpenMP standard offers the syntactically similar equivalents `task/taskwait`.

Despite efficient use for recursive algorithms, this approach implies artificial synchronizations in other widely-used algorithm structures as loop-based ones. As a result, the task-based programming model's current implementations offer data-flow dependencies where each task includes the access mode of its parameters (Augonnet *et al.* 2011; Duran *et al.* 2011; OpenMP Architecture Review Board 2018). At runtime, based on sequential consistency, the runtime system detects actual dependencies, thus enabling fine-grained synchronizations.

Hybrid resources, composed of CPU and GPU cores, are captured by the task-model by providing multiple implementations per task. This feature is available on task-based runtime systems such as **StarPU** (Augonnet *et al.* 2011), **XKaapi** (Gautier *et al.* 2013) and **OmpSs** (Duran *et al.* 2011). The programmer provides additional implementations to the same task to target each hardware device (e.g., CPU core, GPU card, vectorial units, manycore accelerators). When scheduling the tasks on computing resources, the runtime system dynamically decides which implementation will run based on the implementations available and the platform characteristics. Accelerator devices as GPUs have their own memory address space, which is inaccessible from the host system processor. Since task-based applications are decoupled from the hardware, modern runtime systems as **StarPU** automatically handle these separated address spaces using the MSI cache-coherence protocol (Hennessy and Patterson 2017). Some runtimes also implement several optimizations to reduce computing resources idleness, overlapping data transfers with computations and considering data locality.

Figure 1 shows an example of the tiled-LU factorization algorithm in the task-based model. Each call to `sgetrf_nopiv()`, `strsm()`, and `sgemm()` functions represents the creation of a new task. Dependencies among them are inferred from the runtime system using each parameter's provided access mode (e.g., `R`, `RW`). The DAG on the right shows the dependency chain when $N = 4$. The number of each circle identifies the outer-loop iteration of the task creation. A task may have many implementations that are dynamically chosen by the runtime according to data locality, resources availability, scheduling policies, and load balancing.

The task-based programming model remains an excellent alternative to tackle heterogeneous computing nodes since it simplifies the programming effort by relying on established scheduling heuristics. Nevertheless, the performance analysis of such applications may provide hints to runtime and application developers on possible optimizations. The next section presents the **StarVZ** package, which provides a scriptable framework to create visualizations to guide the analyst intuition on performance problems.

3. StarVZ methodology and philosophy

StarVZ aims to provide comprehensive performance analysis for task-based applications supported by runtime systems on hybrid CPU-GPU platforms. Unlike other analysis tools, our tool remains open to alternative methodologies besides graphical representations of application traces gathered by the runtime. To the best of our knowledge, **StarVZ** is the first HPC performance analysis tool conceived on top of modern data science tools. It relies on the expressiveness of the R language (R Core Team 2017) and its rich set of statistical and data manipulation packages (Wickham *et al.* 2019; Wickham 2016; Wickham *et al.* 2020; François *et al.* 2020; Berkelaar *et al.* 2020; Glur 2019). Thanks to that, we can process raw input data, perform cleanups, filtering, merges, statistic computations, and correlations, producing much richer visualizations than those that could be made by directly plotting the raw traces. Earlier

```
for (k = 0; k < N; k++) {
  sgetrf_nopiv(RW, A(k, k));
  for (m = k+1; m < N; m++)
    strsm(R, A(k, k),
          RW, A(m, k));
  for (n = k+1; n < N; n++) {
    strsm(R, A(k, k),
          RW, A(k, n));
    for (m = k+1; m < N; m++)
      sgemm(R, A(m, k),
            R, A(k, n),
            RW, A(m, n));
  }
}
```

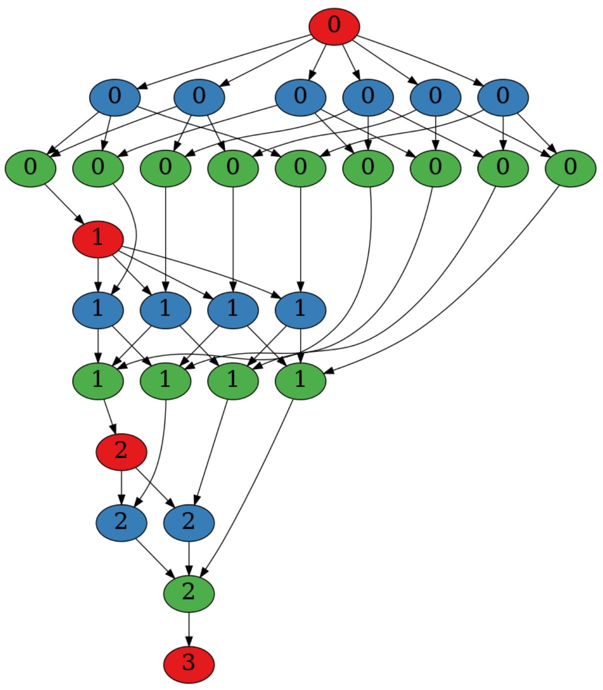


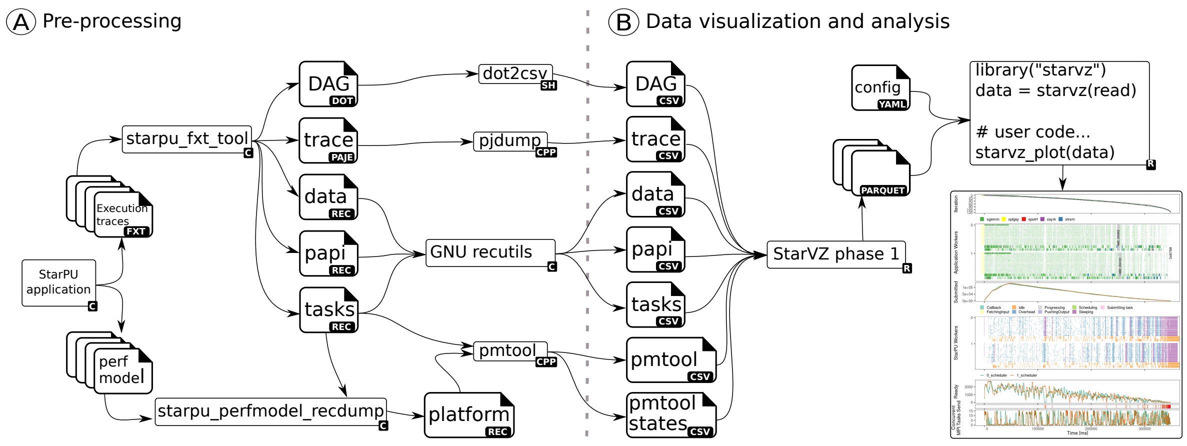
Figure 1: Pseudo-code of the LU algorithm and its corresponding DAG (N = 4); each node is a task of a certain type (color) with its outler loop iteration identification (the k value).

publications (Pinto *et al.* 2016; Garcia Pinto *et al.* 2018; Leandro Nesi *et al.* 2019) present the initial steps and the first contributions related to the creation of the **StarVZ** R-package.

3.1. Workflow overview

Figure 2 presents a detailed representation of the workflow. **StarVZ** input consists of a set of text files in the comma-separated values (CSV) format. These files can be directly obtained from the runtime system raw traces using standard and open-source tools such as **paje**, **sort**, **sed**, and **recutils**. These pre-processing steps appear in the left part of the Figure 2 and are outside of the scope of the **StarVZ** R package (represented in the right side). In the first run with a given dataset, a call to `starvz_phase1()` will process these CSV files performing several filtering, joins, and statistical operations to build the package's internal tables. Such tables are dumped to compressed parquet binary files to save time at further runs. Once this initial processing is the more costly step of the package workflow, additional runs on the same dataset are faster, allowing quick exploration of complete visualizations. Since HPC experiments usually run on powerful dedicated platforms (e.g., clusters), this initial phase can be optionally executed as a post-processing step of the application's execution.

Using these internal tables' data, detailed in Section 3.3, **StarVZ** can generate several different panels visually representing the application's performance. Section 4 presents the most relevant panels. A user can customize the final visualization by setting properties on a configuration file at the YAML format. A default setup takes place when a configuration file is absent. Online fine-tuning is also possible by changing settings before a call to `starvz_plot()` or passing alternative arguments to the functions calls creating the visualization panels.



```
library("starvz")
data = starvz(read)

# user code...
starvz_plot(data)
```

Figure 2: Detailed view of the workflow with its the pre-processing phase (A, left part) and data visualization and analysis (B, right), transforming execution traces into visualization.

3.2. Basic usage with code snippets

The **StarVZ** package is available from CRAN at https://CRAN.R-project.org/package=starvz. The installation procedure is as simple as:

```
R> install.packages("starvz")
```

In R, a basic usage is as follows:

```
R> library("starvz")
R> starvz_phase1(directory = "data/basic-usage",
                 whichApplication = "lu", app_states_fun = lu_colors)
R> jss_sample_lu <- starvz_read(directory = "data/basic-usage",
                                selective = FALSE)
```

Once the data is at the `jss_sample_lu` variable, one can produce a basic plot (see Figure 3 for the output) with:

```
R> starvz_plot(jss_sample_lu)
```

3.3. Data

The origin of all input data is the runtime specific traces. In the case of **StarPU**, this is a **FxT** file. As described in Section 3.1, the first phase of **StarVZ** applies runtime-specific tools and functions to these traces translating them to CSV files. The scripts ultimately transform these text files into a list of tables that constitutes the primary **StarVZ** data. The structure of these CSV files reassembles the final data structure. However, many transformations and filtering are applied to reduce and prepare them for the analysis/visualization. These transformations are specific for each case and are unimportant for the end-user. We provide only a brief overview of the principal operations that we do.

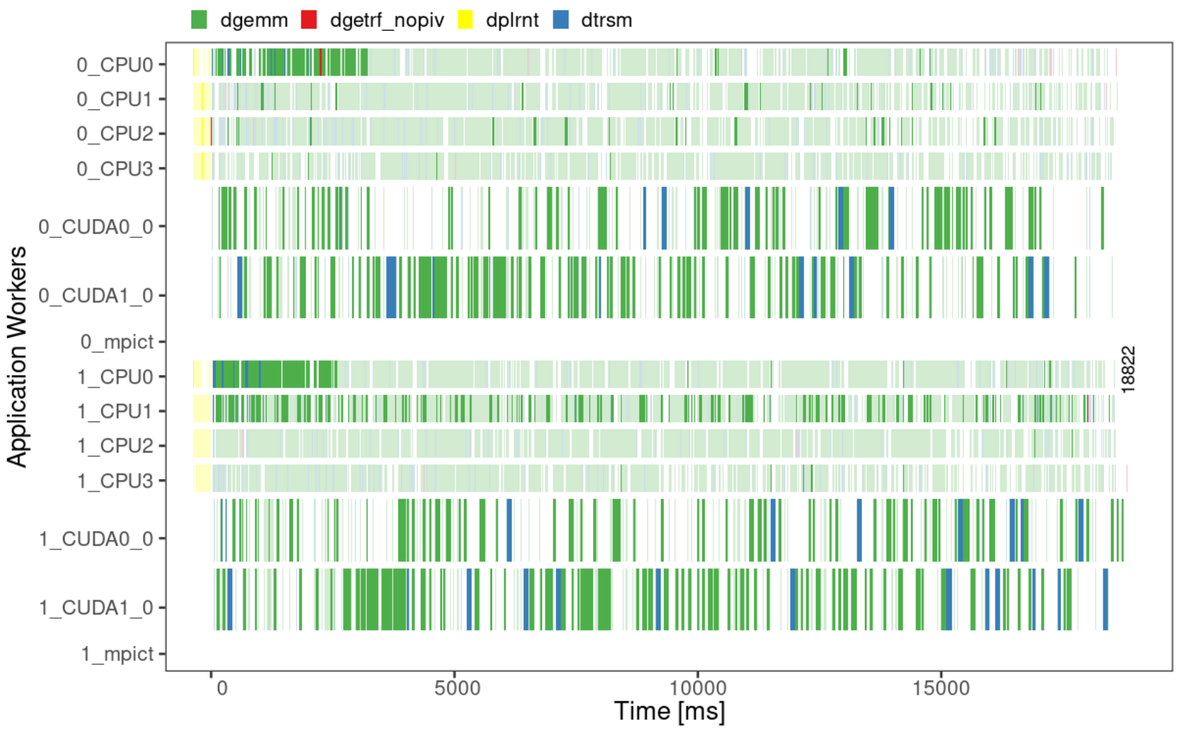


Figure 3: The space/time panel depicting the behavior of the LU factorization when executed with two computing nodes, each one with two GPUs and four CPUs.

The common transformations that the first phase executes are popular data-science operations. For example, many character columns are converted to R factors to save space. Furthermore, the CSV might be split in more than one table because they refer to very distinct data categories with different columns. Also, we select and derive detailed information. For example, **StarPU** may inform one entity as `NAME_NODE`, a redundant as we may already have the node, and all components of that table have that `NAME`. The first phase computes relevant structures like the application DAG. We also shift the timestamp of all operations setting as zero the beginning of the first appropriate application task. Events that occur before this will have a negative timestamp. This time shift is relevant because we are usually interested in the application execution. We keep all negative timestamped information as we do not want to lose track of prior initialization and overhead. Also, this phase may apply application-specific functions during the CSV reading. For example, the **QRMumps** application uses a generic **StarPU** field to store application-related data properly converted by **StarVZ**. Furthermore, some aesthetics decisions remain fixed in these transformations, like colors of tasks, names of entities, and vertical ordering of elements. Moreover, most of first phase's operations, when possible, are isolated from each other. They load the CSV file, do the procedures, save the parquet files, and free all their data, keeping the memory footprint as low as possible.

The **StarVZ** primary data is an R 'S3' Class with a list of '`Tibbles`'. Not all tables need to be present, and their structure may differ depending on the runtime and version. Each panel function requires different tables/columns for their views. That means that the structure is flexible and lazy and could contain a subset of the data to execute only desirable functions. Only the views required data must be present or be loaded. Panel functions enforce the data requirements by the internal function `starvz_check_data()`. Table 1 lists the supported tables as available in the **StarVZ** version `0.4.0`.

Table	**Description**
Version	Single value describing the **StarVZ** data version
Colors	Table containing tasks color information
Y	The vertical structure of workers used in the Y-axis of different plots
Application	The application tasks that include Start, End, and name of the Task
Starpu	The internal tasks of the **StarPU** runtime
Gaps	The computed temporal intervals between tasks
Link	The data transfers or entities among workers/nodes
Dag	The application directed-acyclic graph
Pmtool	The performance bounds computed by **pmtool** (Eyraud-Dubois 2019)
Pmtool_states	The tasks scheduling computed by **pmtool** (Eyraud-Dubois 2019)
Atree	Information about the elimination tree present in some algorithms
Tasks	Additional information about tasks
Data_handles	Information about application data registered by the runtime
Task_handles	Information about the data used by tasks
Memory_state	States of the memory controller entities of the runtime
Events_memory	Events related to memory operations
Events	Generic table for events
Variable	Statistics and metrics computed during execution, like ready tasks
Comm_state	States of the MPI communication entity
Other_state	Generic table for other states of other entities
Papi	Information about the tasks' hardware counters
Origin	The original directory of the trace
Zero	Timestamp of the beginning of the first application task

Table 1: Summary of data tables and their description.

We provide a sample **StarVZ** data in the package stored in the variable `starvz_sample_lu`. The tables' data can be accessed directly by accessing a named list element in R. For example, to access the table Application we could use:

```
R> starvz_data$Application
```

Although the amount of data registered in the data is vast, **StarVZ** also computes additional information dynamically. We decided to keep them absent from the registered `parquet` data because they are relatively inexpensive to compute or require user configuration. For example, **StarVZ** can classify outlier tasks using different performance models. It may also be necessary to apply aggregation functions per time step because of the excessive amount of data. Some panels may present metrics such as critical path bound (CPB), area bound estimation (ABE), and Pmtool's performance bounds (externally computed by the **pmtool** (Eyraud-Dubois 2019) application). The parameters that tailor this additional information might be passed by the user or defined in the configuration file. One example of such a parameter is the time-step interval of data aggregation.

4. Views

This Section exposes the main functionalities of the **StarVZ** package, documenting the functions that create performance visualization plots. All plotting functions produce **ggplot** objects and have their names prefixed with `panel_`. These functions always receive as first argument the `data` value (see Section 3.3 for details), which represents the contents of the execution trace after post-processing during the first phase. Table 2 lists the panel functions organized by category. The arguments shared by all functions include `data`, `legend`, `x_start`, `x_end`, and `step`. The `legend` boolean parameter indicates whether to enable/disable legends. The `x_start` numeric parameter indicates the X-axis start value and defaults to the first application task's beginning timestamp. The `x_end` numeric parameter indicates the X-axis end value and defaults to the last application task's end timestamp. All time-aggregated panels share a **step** parameter that contains a numeric value to define the size in milliseconds of the time aggregation interval. The default value for this parameter is 0.1% of the application makespan. The next Sections provide details about each of the panels, including an explanation about their parameters. These Sections use **Chameleon** (Agullo *et al.* 2010) and **QRMumps** (Agullo *et al.* 2013) executions traces as examples.

4.1. Worker view

The `panel_st_raw()` function represents application and runtime states in a space/time organization. The panel has the workers listed on the Y-axis and their states in the X-axis, associating colors to state type. We use this function as follows:

```
R> panel_st_raw(data = jss_sample_lu, labels = "FIRST_LAST",
                ST.Outliers = TRUE, runtime = FALSE, idleness = TRUE,
                makespan = TRUE, abe = TRUE, taskdeps = TRUE,
                tasklist = c("1_6849"), levels = 7)
```

Figure 4 presents a view with many additional features to the classical space/time plot. As an example, we list some parameters that activate them. Option `labels` controls workers labels in the Y-axis. It accepts, for example, `"ALL"` to display all resources names, and `"NODES_only"` to show just the node ID. Parameter `ST.Outliers` highlights (with darker colors) those tasks classified as anomalous by the performance models. The option `idleness` plots per-resource idleness quantification in percentage. The option `abe` plots a vertical gray bar showing a makespan estimation computed with a linear program. Such estimation is also helpful to check load balancing among nodes. The options `taskdeps`, `tasklist`, and `levels` enable plotting of last-solved task dependencies (as red edges).

The performance of HPC task-based applications is heavily dependent on runtime system decisions which makes relevant to visualize its states when analyzing the overall performance. The function `panel_st_raw()` with parameter `runtime = TRUE` generates a plot (see Figure 5) with the runtime-oriented space/time view:

```
R> panel_st_raw(jss_sample_lu, runtime = TRUE,
                labels = "FIRST_LAST")
```

Category	Function
Worker	`panel_st_raw()`
	`panel_st_agg_node()`
	`panel_st_agg_dynamic()`
	`panel_st_agg_static()`
Application	`panel_atree()`
	`panel_activenodes()`
	`panel_nodememuse()`
	`panel_utiltreedepth()`
	`panel_utiltreenode()`
	`panel_kiteration()`
Variable	`panel_usedmemory()`
	`panel_gflops()`
	`panel_gpubandwidth()`
	`panel_mpibandwidth()`
	`panel_mpiconcurrentout()`
	`panel_mpiconcurrent()`
	`panel_ready()`
	`panel_submitted()`
Memory	`panel_handles()`
	`panel_memory_snap()`
	`panel_memory_state()`
MPI	`panel_mpistate()`
Overview	`panel_lackready()`
	`panel_node_summary()`
	`panel_node_events()`
Pmtool	`panel_pmtool_kiteration()`
	`panel_pmtool_st()`

Table 2: Summary of functions by category.

4.2. Application view

Elimination tree

Function `panel_atree()` plots the structure of the elimination tree along time, defining the start and end of each tree node, connecting them with arrows to represent the child and parent relationship. The function parameters can control the representation of computation, communication, initialization, and anomalies in the tree. The Figure 6 shows the resulting panel when we call the function as follows:

```
R> sample_qr_trace <- starvz_read(directory = "data/qr-trace")
R> panel_atree(data = sample_qr_trace,
               step = 100, computation = TRUE,
               pruned = TRUE, initialization = TRUE, anomalies = TRUE,
               communication = TRUE)
```

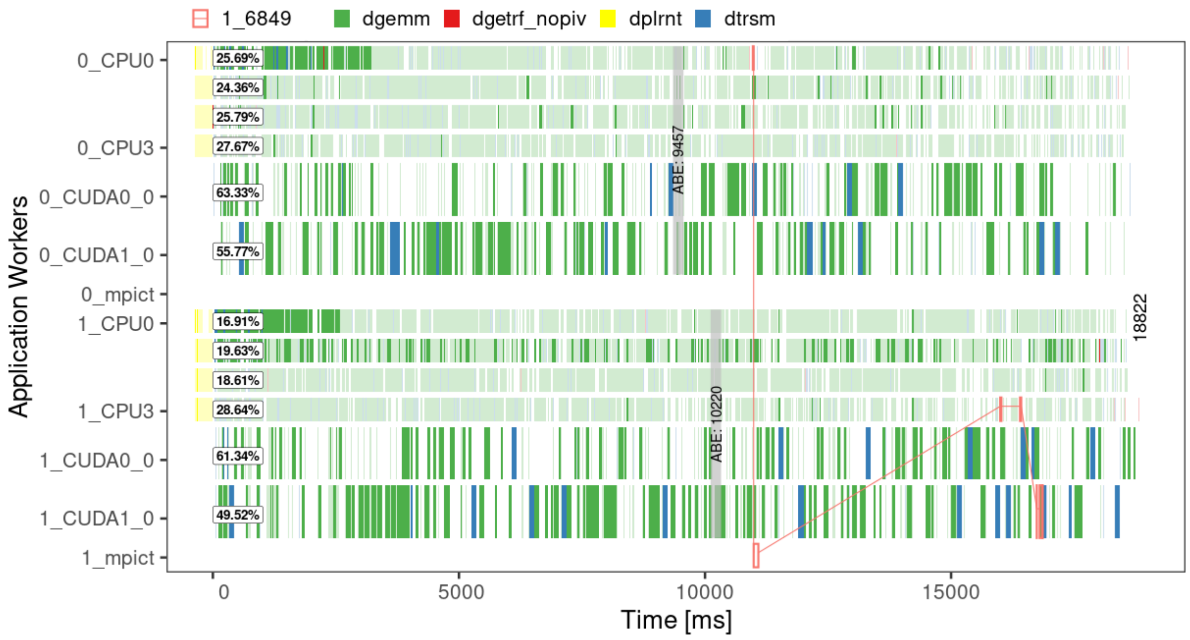


Figure 4: The space/time panel depicting application states.

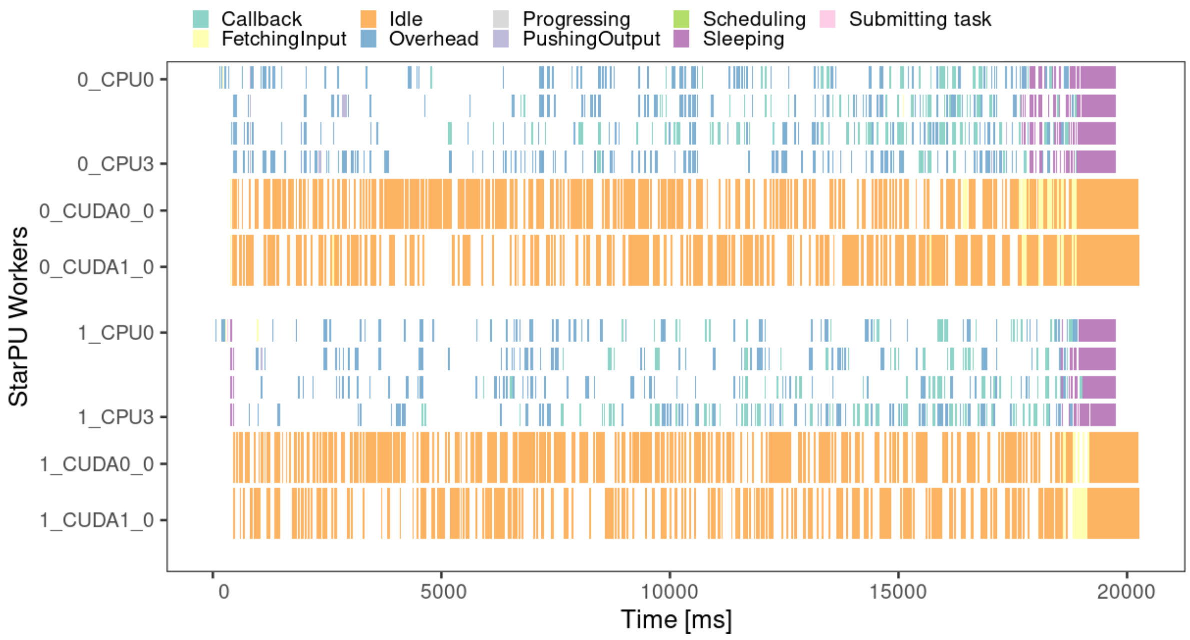


Figure 5: The space/time panel depicting runtime states.

The function parameters described here are all booleans that enable/disable the tree structure's visual appearance. The `computation` and `pruned` parameters control the resource utilization representation by factorization tasks for tree nodes and pruned tree nodes. The blue/red color gradient, positioned in the Y-axis according to the tree nodes' submission order, depicts each tree node's aggregated computational intensity. Rectangles with smaller

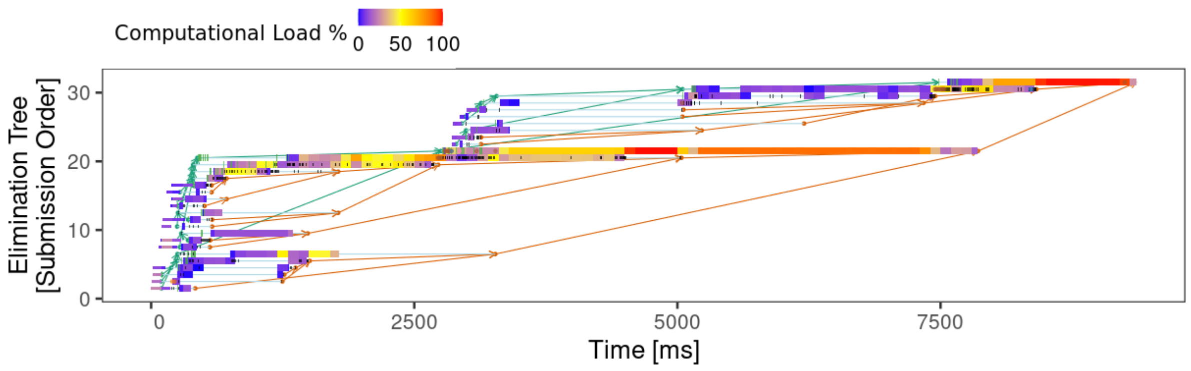


Figure 6: The elimination tree behavior along time.

height represent the pruned nodes, which are grouped and aggregated by a common parent. The `initialization` controls the representation of node memory allocation tasks (green rectangles in Figure 6). The `communication` parameter is responsible for controlling communication tasks, presented as black rectangles in a node line. Lastly, the `anomalies` parameter indicates whether to plot the location of anomalous tasks in the tree structure, using a point with the same color of the tasks in `panel_st_raw()`.

The functions `panel_activenodes()` and `panel_nodememuse()` represent the application memory behavior related to the tree structure. The first one shows the number of allocated nodes in memory at a given time, and the other presents the total memory usage in MegaBytes of these active nodes. Their parameters allow plotting the raw data values over time when `aggregation` is `FALSE`, or plot the aggregated values in time by a given `step` size. Usage and output (see Figure 7) of these functions are as follows:

```
R> library("patchwork")
R> library("ggplot2")
R> active <- panel_activenodes(data = sample_qr_trace,
                                  step = 100, aggregation = TRUE) +
             theme(axis.text.x = element_blank(),
                   axis.title.x = element_blank())
R> memuse <- panel_nodememuse(data = sample_qr_trace,
                                step = 100, aggregation = TRUE)
R> active / memuse
```

To represent the computational resource utilization related to the tree structure we can use the functions `panel_utiltreenode()` and `panel_utiltreedepth()`. They depict how many computational resources (Y-axis) the compute tasks are using inside a given time `step` (X-axis). The difference between them is that the former paints the area representing the resource utilization by individual tree nodes, while the latter paints by the tree depth. Hence, these views allow us to observe how much tree and node parallelism the algorithm is exploring. Usage and output (see Figure 8) of these functions are as follows:

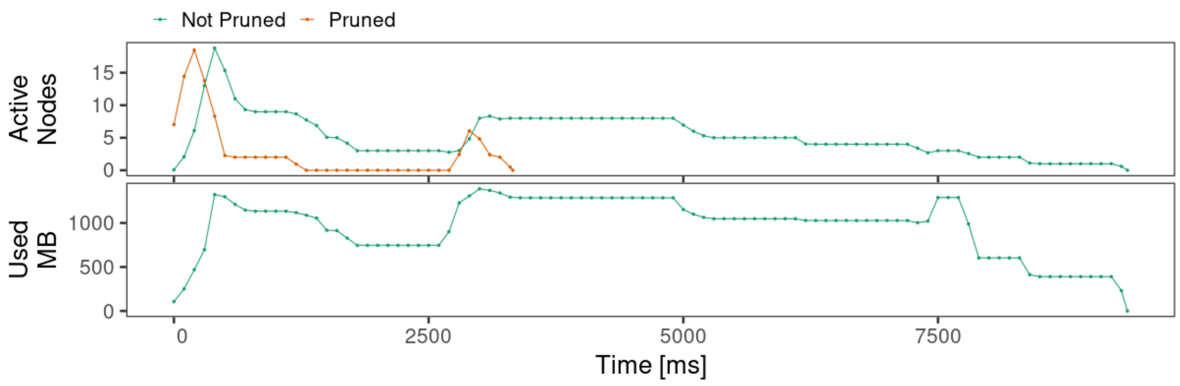


Figure 7: Number of active elimination tree nodes along time (top), and the total memory utilization along time (bottom).

```
R> library("patchwork")
R> library("ggplot2")
R> node <- panel_utiltreenode(data = sample_qr_trace, step = 100) +
                             theme(axis.text.x = element_blank(),
                                   axis.title.x = element_blank())
R> depth <- panel_utiltreedepth(data = sample_qr_trace, step = 100)
R> node / depth
```

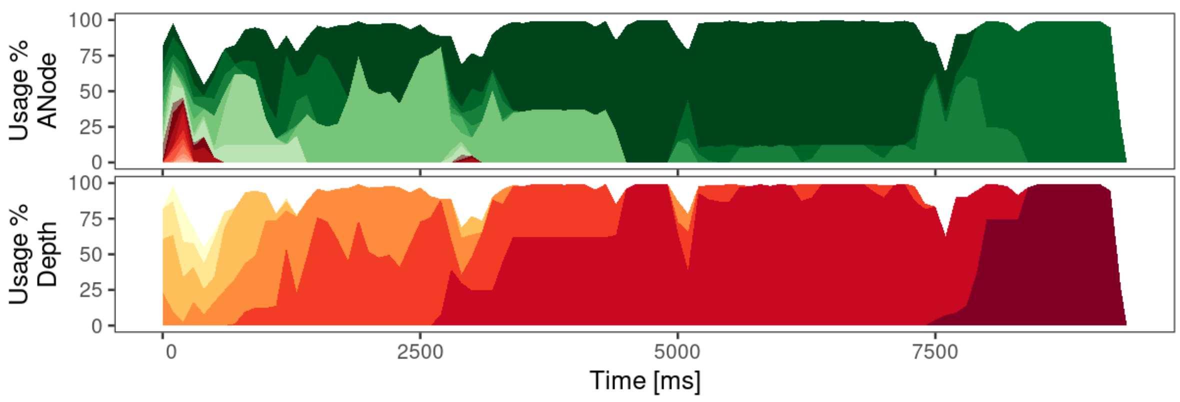


Figure 8: Resource utilization along time by elimination tree node (top), and by elimination tree depth (bottom).

K-iteration

Many parallel applications use loop-based algorithms. The `panel_kiteration()` function enables one too check iterations overlap along time. This overlap is possible in a task-based model since it uses fine-grained synchronizations, i.e., there is no global barrier among loop-iterations. The Y-axis depicts the iteration index while the X-axis is the time. Black curves represent iteration start (left) and iteration end (right). Each running task is plotted with an alpha value between these curves once many tasks can run simultaneously on different resources. Blank spaces mean no task of this iteration is running. The shape of this plot changes according to the DAG traversal implemented by the scheduling policy. A wider shape represents more parallelism, giving flexibility to the runtime system. Figure 9 depicts an example obtained with the following call:

```
R> panel_kiteration(jss_sample_lu)
```

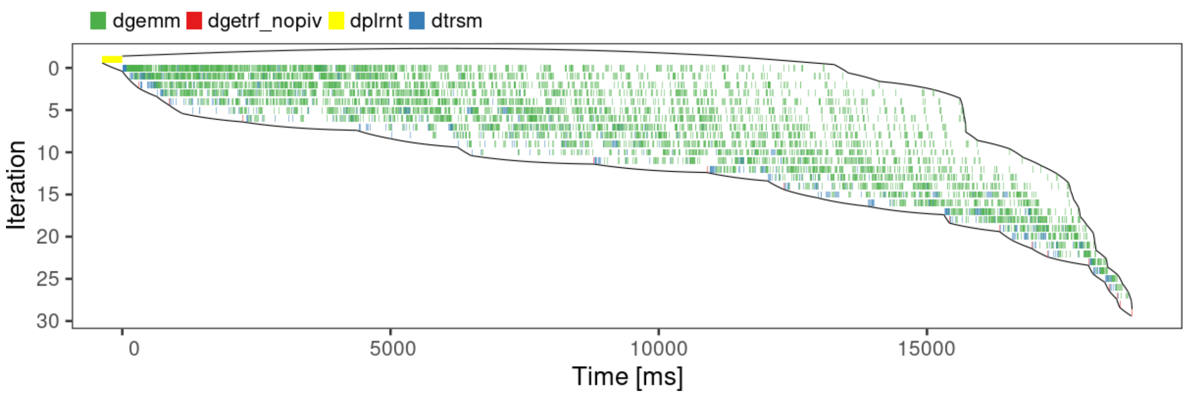


Figure 9: The k iteration panel showing iteration timespan and application states along time.

4.3. Runtime memory management

Memory management is a crucial aspect of HPC applications' performance. Some runtimes, including **StarPU**, automatically controls the application memory using the DAG structure. In the case of **StarPU**, the minimal memory unit (used by tasks) is the data handle, and each resource memory (RAM, GPUs) has an entity called memory manager to control all the related operations (transfers, allocations, and frees). **StarPU** uses the MSI protocol to guarantee the coherence of data handles across all managers. Each data unit has one of the possible three states (own/share/invalid) on each manager. **StarVZ** provides two time-oriented visualizations for enabling inspection of the automatic memory control of **StarPU**. These views are a result of an earlier publication (Leandro Nesi *et al.* 2019).

The first view, `panel_memory_state()`, presents a **StarPU**'s memory manager oriented view using Gantt Charts. Figure 10 depicts the resulting panel, where the Y-axis lists the memory managers, and the X-axis is time. The states are issued operations like allocation of data, transfers, and free. The `combined` option enables the visualization of transfers between entities using arrows where the horizontal arrow length corresponds to its duration. The following command gives an example of such a view:

```
R> panel_memory_state(data = jss_sample_lu, show_state_total = FALSE,
                      show_transfer_total = FALSE,
                      combined = TRUE, x_end = 200)
```

The second view, obtained with the `panel_handles()` function, provides a **StarPU**'s data handle-oriented visualization. It may be valuable to visualize the progress of a memory unit across all the execution. The visual structure of this panel is similar to the previous one (see

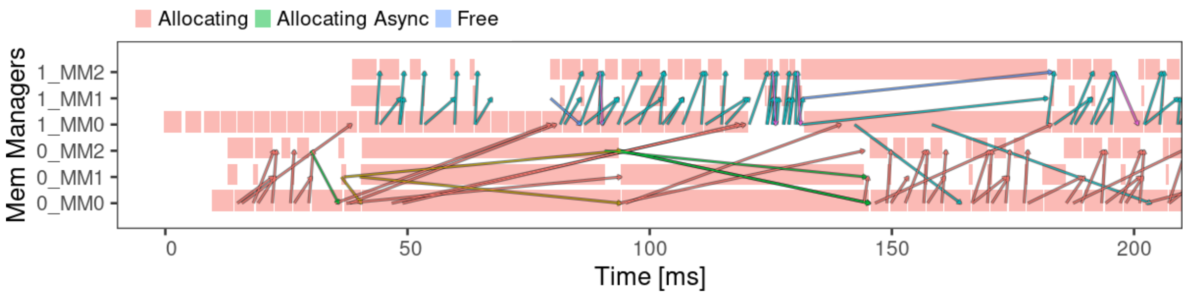


Figure 10: The memory states panel depicting memory-related actions along time.

Figure 10). The Y-axis has the list of memory managers, and the X-axis is time. However, space/time states represent the data handle's MSI situation on that entity. Additionally, this panel presents all the handle events, including allocation/transfer requests and events such as fetches or pre-fetches. The view also shows an inner state inside the MSI to show current tasks accessing the data handle. The colors of the task states are equal across all **StarVZ** plots. These states can present two borders; solid lines indicate that the task had write permissions, while dashed ones the task only had read permissions. Again, this view uses arrows to show communications between memory managers. Because many blocks exist, a simultaneous visualization in the same image using this panel is not scalable and not advisable. Instead, the panel function provides the `lHandle` and `JobId` arguments to select a list of handles or handles of a specific job, respectively. The handles are identified per node using its address and across multiple nodes using a Tag. This crude information seems not human friendly, so the user may replace it using custom functions (using the argument `name_func`) to name them while **StarVZ** matches the handles across all nodes. If the application provides the coordinates of data handles to **StarPU**, **StarVZ** will use them, as default, to create names of the form: "Memory Block X Y" for 2D-cases. A code example of such panel utilization (see Figure 11 for the output) selecting the data handles of task `1_300` is given by:

```
R> panel_handles(data = jss_sample_lu, JobId = "1_300",
                 x_start = 15000, x_end = 16500)
```

4.4. Variables

All the Variable panels share the same principle: to filter the **StarVZ** data for timestamped numerical metrics, apply temporal aggregation to smooth and reduce the number of data points, and depict them as a line chart where the Y-axis is the variable, the X-axis is the time, and each line color distinguishes the correlated entity (for instance, the scheduler of different compute nodes). Besides aesthetics arguments, variable panels require the **step** argument that indicates the temporal aggregation's granularity. An example of a variable panel is the `panel_ready()` that shows the number of ready tasks per node scheduler, as shown in Figure 12. We can obtain it using a step of 50ms with the following code:

```
R> panel_ready(data = jss_sample_lu, step = 50)
```

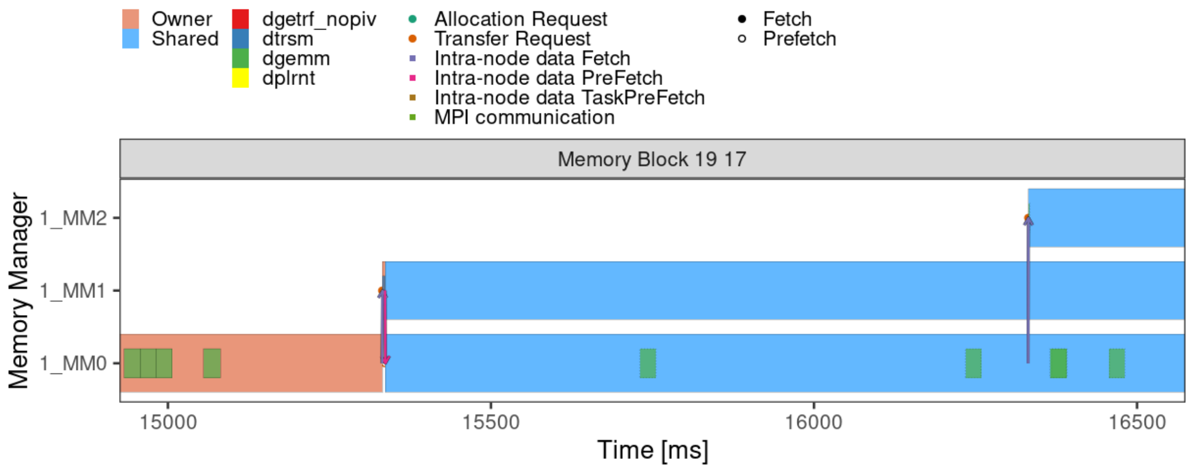


Figure 11: The data handles history plot illustrating events related to a specific data block.

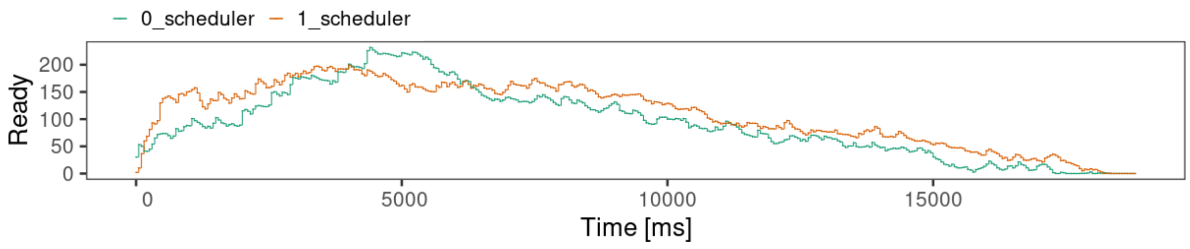


Figure 12: The ready plot, as an example of variable panel, to depict values along time.

The other variable panels are the following. The `panel_usedmemory()` depicts the used memory per entity. The `panel_gflops()` inform per worker achieved GFlops. The communication bandwidth of the GPUs is presented by `panel_gpubandwidth()`. The `panel_mpibandwidth()` shows the communication bandwidth for MPI nodes. The `panel_mpiconcurrent()` and `panel_mpiconcurrentout()` functions depict the total number of concurrent MPI operations being received and send. Finally, the `panel_submitted()` shows the number of submitted and pending tasks per node scheduler.

4.5. Aggregated views

Sometimes the number of events recorded in the traces is too high to be graphically represented in the plot, leading to known rendering issues (Schnorr and Legrand 2013). Too much detail can also hide relevant behavior, reducing the plot readability and usefulness. **StarVZ** handles that with explicit aggregation controlled by the user. The function `panel_st_agg_static()` slices the time in fixed intervals and computes how much each task type appears within the interval. Optionally, outliers can be print on top of aggregated areas. The function `panel_st_agg_dynamic()` offers a more dynamic approach, aggregating tasks until it reaches an exclusion criterion (e.g., a specific task type, outlier, or minimum task duration) which prevents relevant states from being eclipsed by aggregation. Figures 13 and 14 illustrate the static and dynamic temporal aggregation as created by the following code snippets:

```
R> panel_st_agg_static(jss_sample_lu, outliers = FALSE, step = 500)
```

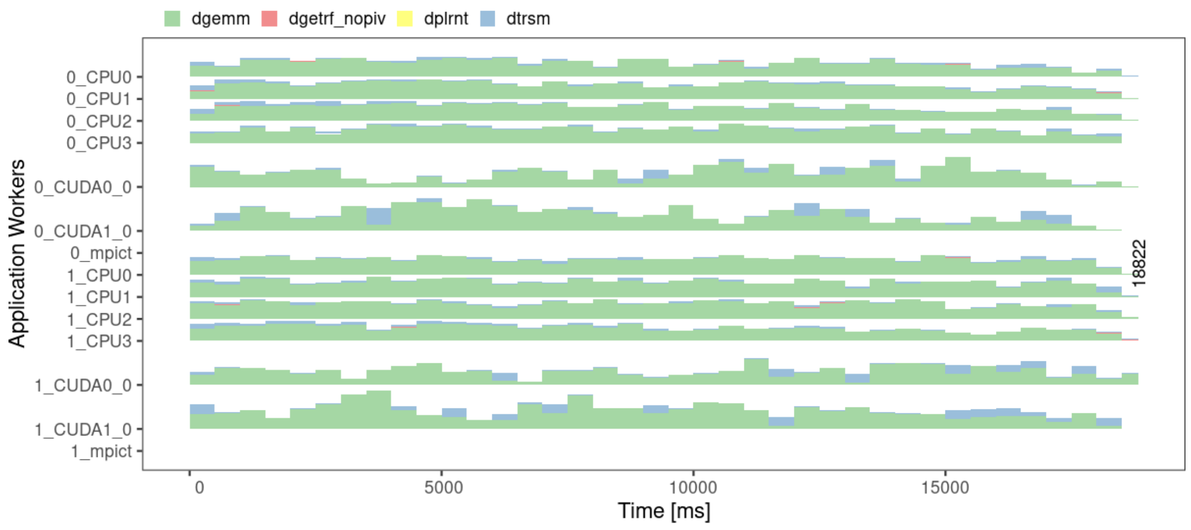


Figure 13: The space/time plot with static temporal aggregation.

```
R> panel_st_agg_dynamic(jss_sample_lu)
```

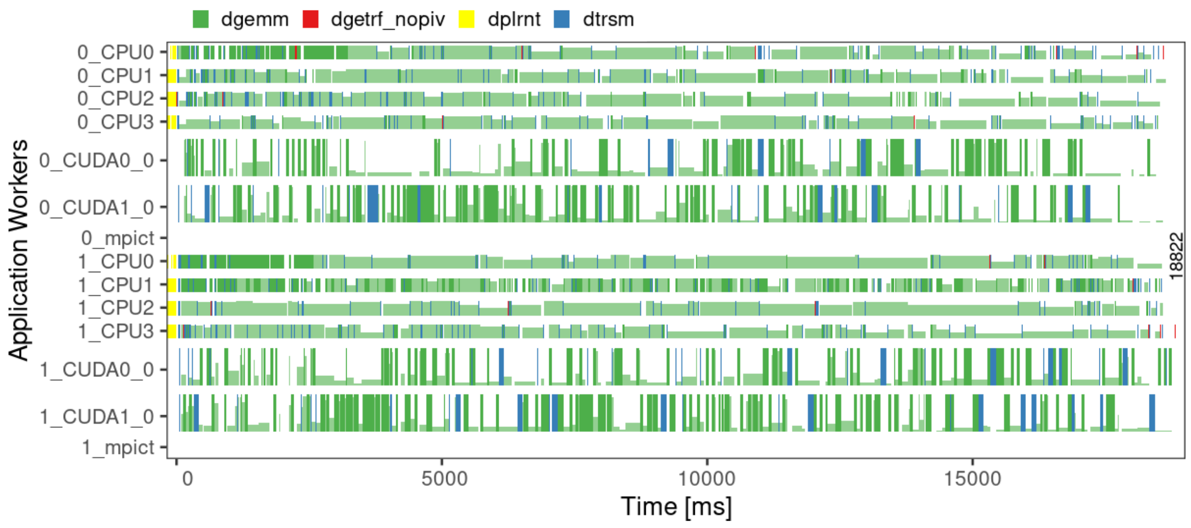


Figure 14: The space/time plot with dynamic temporal aggregation and outliers identification.

The function `panel_st_agg_node()` offers a summarized per-node view that aggregate tasks running on resources of the same type. Such a panel enables a much more scalable visualization of large platforms with high CPU/GPU-count. The code to obtain Figure 15 is as follows:

```
R> panel_st_agg_node(jss_sample_lu)
```

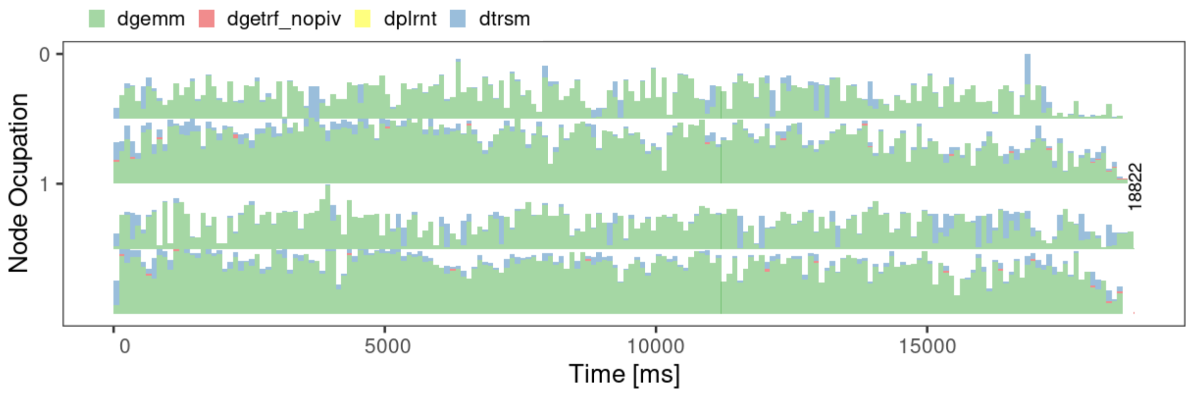


Figure 15: The application space/time plot with node aggregation.

The function `panel_lackready()` presents another use of the temporal/spatial aggregation feature, as depicted in Figure 16. It indicates moments when the amount of ready tasks is not enough to fill all compute resources, i.e., so we expect idleness to appear.

```
R> panel_lackready(jss_sample_lu, x_start = 0)
```

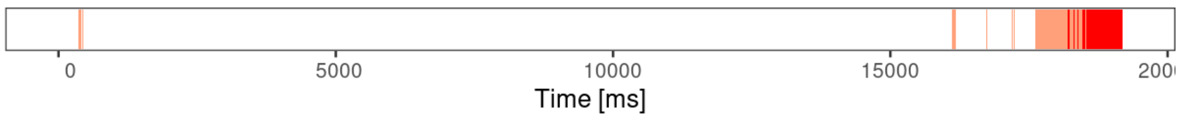


Figure 16: The plot to pinpoint when the number of ready tasks is smaller than the number of computing resources, indicating insufficient parallelism from the application.

4.6. Non-temporal views and metrics

This Section presents some auxiliary views and metrics' functions with an non time-oriented output. For many applications, the data has a spatial meaning. For example, linear algebra solvers represent a matrix into 2D organized tiles. Each tile has a coordinate correlated to the application algorithm. Sometimes, it is useful to view the memory using this application information correlating it with its algorithm. **StarVZ** has three visualizations that follow these characteristics for matrix-oriented applications. These views present the data in a 2D grid using the blocks coordinates. **(i)** `panel_memory_snap()` presents the current state of the memories on a specific moment defined by the argument **selected_time**, each major tile is the MSI state of that block, and the inner minor tile is tasks that are using it at that moment. The shape of the inner tile is the type of access (write or read). This view also has a **step** argument to show all tasks that started after $selected\_time - step$ and finished before $selected\_time$ to give a progression perspective. **(ii)** `panel_memory_heatmap()` shows how much time each block was present on each manager. The intensity of the tile's color represents the presence. **(iii)** `panel_dist2d()` presents the static initial distribution of the data across multiple MPI nodes, where the tiles' color is a different MPI node that owns that data. Figure 17 presents these three views.

```
R> library("ggplot2")
R> library("patchwork")
R> snap <- panel_memory_snap(data = starvz_sample_lu, step = 100,
                              selected_time = 200, base_size = 26) +
           ggtitle("Snap")
R> heatmap <- panel_memory_heatmap(starvz_sample_lu, base_size = 26) +
           ggtitle("Heatmap")
R> dist2D <- panel_dist2d(starvz_sample_lu, legend = FALSE, base_size = 26) +
           ggtitle("2D Dist")
R> snap + plot_spacer() + heatmap + plot_spacer() +
          dist2D + plot_layout(widths = c(2, 0.05, 2, 0.05, 1))
```

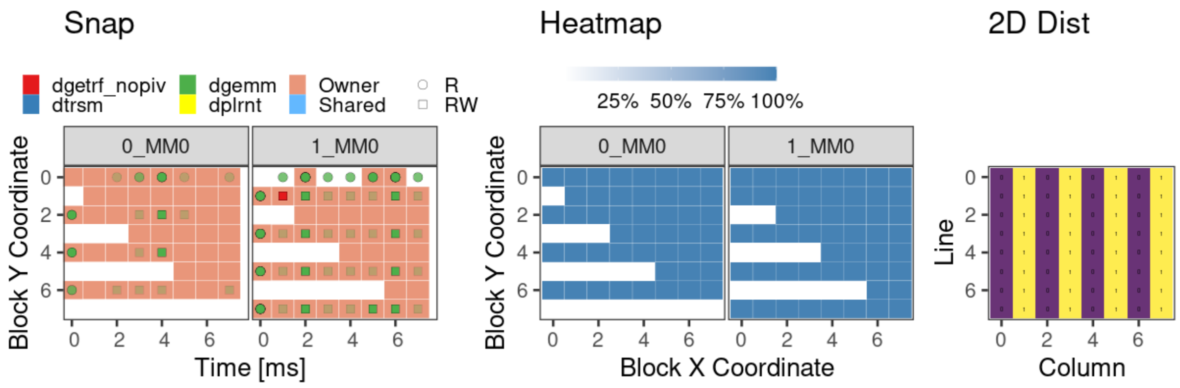


Figure 17: The memory snap (left), memory heatmap (center), and distribution (right) plots.

The ABE provides an estimation to the makespan (see Section 4.1) computed with a linear program. While the estimation helps assess whether there is room for improvements, the linear program's solution can provide hints to achieve this goal. The `panel_abe_solution()` function plots bullets with the ideal partition of each task per resource type and per node; and bars representing the observed partition in the real execution. Figure 18 depicts such a view obtained with the code:

```
R> panel_abe_solution(data = jss_sample_lu)
```

As we use a regression model to classify tasks with anomalous duration in the cases where they have irregular costs, like in sparse matrix factorization, it is essential to have a diagnostic plot to observe if the model fits well in the data. The `panel_model_gflops()` represents the relationship between the theoretical GFlops and the duration of computing tasks. Each panel depicts such a relationship for each task and resource type. Figure 19 depicts the view obtained with the code:

```
R> qr_model_data <- starvz_read(directory = "data/qr-model")

R> panel_model_gflops(data = qr_model_data, freeScales = FALSE)
```

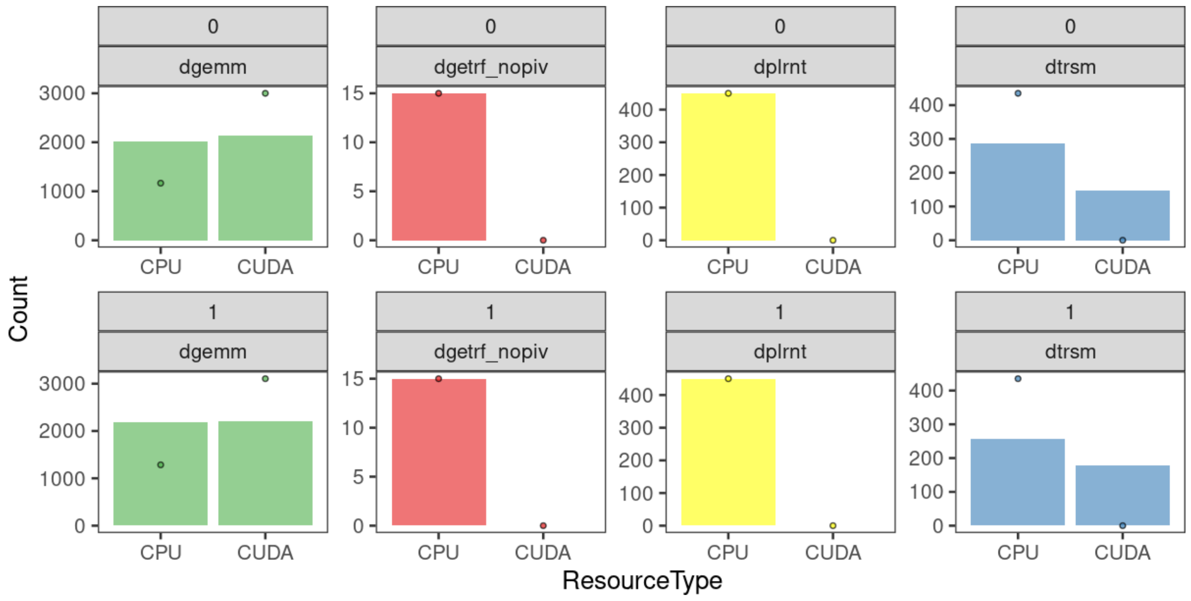


Figure 18: Illustrating the solution for the area bound estimation.

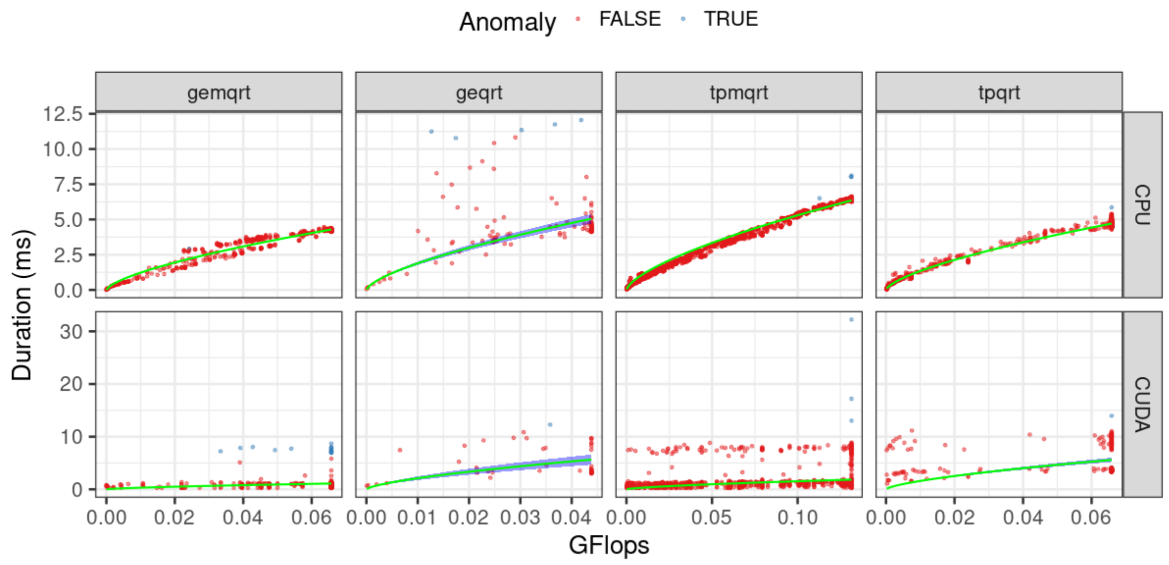


Figure 19: GFlop and task duration diagnostic plot.

4.7. Extensibility

Although **StarVZ** provides the function `starvz_plot()` that generates and assembles all active plots in the configuration, it may be of user interest to reassemble plots using a different layout or even combine multiple **StarVZ** plots. **StarVZ** provides two functions (used internally in `starvz_plot()`) for manual assembly purposes. The function `starvz_plot_list()` returns a list of **ggplots** of all enabled panels. This function is useful if the user wants to customize the final plot layout (using, for example, **patchwork** with the individual **ggplot2** elements), or individual **ggplot** characteristics (like colors, spacing, and guides). For example, if the user wants to show the `state` view and `summary_nodes` view horizontally, as depicted in Figure 20:

```
R> library("patchwork")
R> jss_sample_lu$config$summary_nodes$active <- TRUE
R> plot_list <- starvz_plot_list(data = jss_sample_lu)
R> plot_list$st + plot_spacer() +
   plot_list$summary_nodes + plot_layout(widths = c(1, 0.01, 1))
```

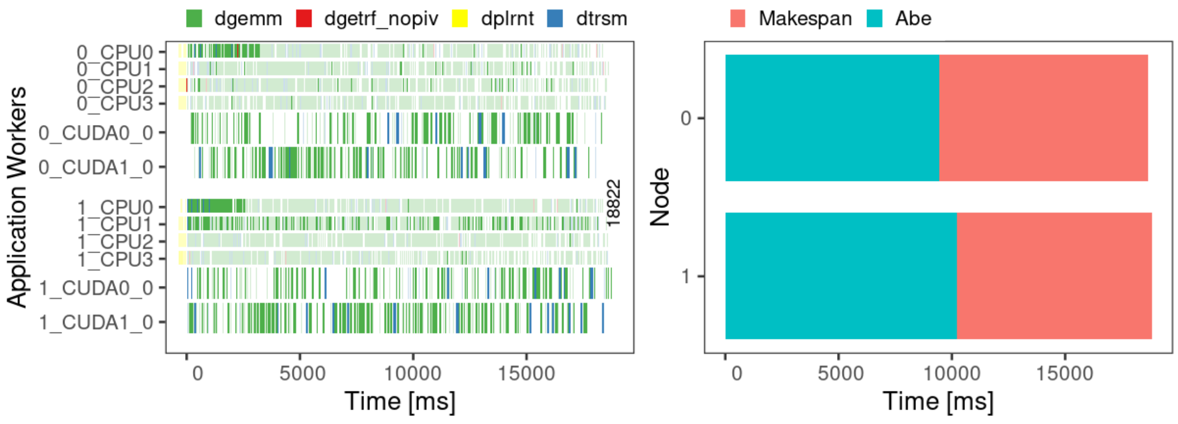


Figure 20: A custom layout with the space/time view (left) and summary nodes (right).

Moreover, if the goal is to present multiple **StarVZ** plots (of different executions) side-by-side for comparing purposes, the function `starvz_assemble()` can be used. The argument of `starvz_assemble()` is a list of views (for one plot only) or a list of views‘ lists (for multiple trace visualization). For coherence, only one argument `config` configuration should be use. The function automatically removes Y-axis titles (argument `remove_Y_info`) of the second and following plots and avoid legend duplication (argument `remove_legends`). An example of assembling two different trace visualizations is shown in Figure 21. The `starvz_plot_list()` function generates each plot, we combined them with a call to the `starvz_assemble()` function, as follows:

```
R> library("ggplot2")
R> jss_sample_lu_2 <- starvz_read(directory = "data/compare-usage",
                                  config_file = "data/compare-usage/config.yaml",
                                  selective = FALSE)
R> jss_sample_lu_2$config$limits$end <- 18900
R> jss_sample_lu$config <- jss_sample_lu_2$config
R> jss_sample_lu$config$title$text <- "LWS"
R> trace1_plots <- starvz_plot_list(jss_sample_lu)
R> trace2_plots <- starvz_plot_list(jss_sample_lu_2)
R> starvz_assemble(trace1_plots, trace2_plots,
                   config = jss_sample_lu_2$config)
```

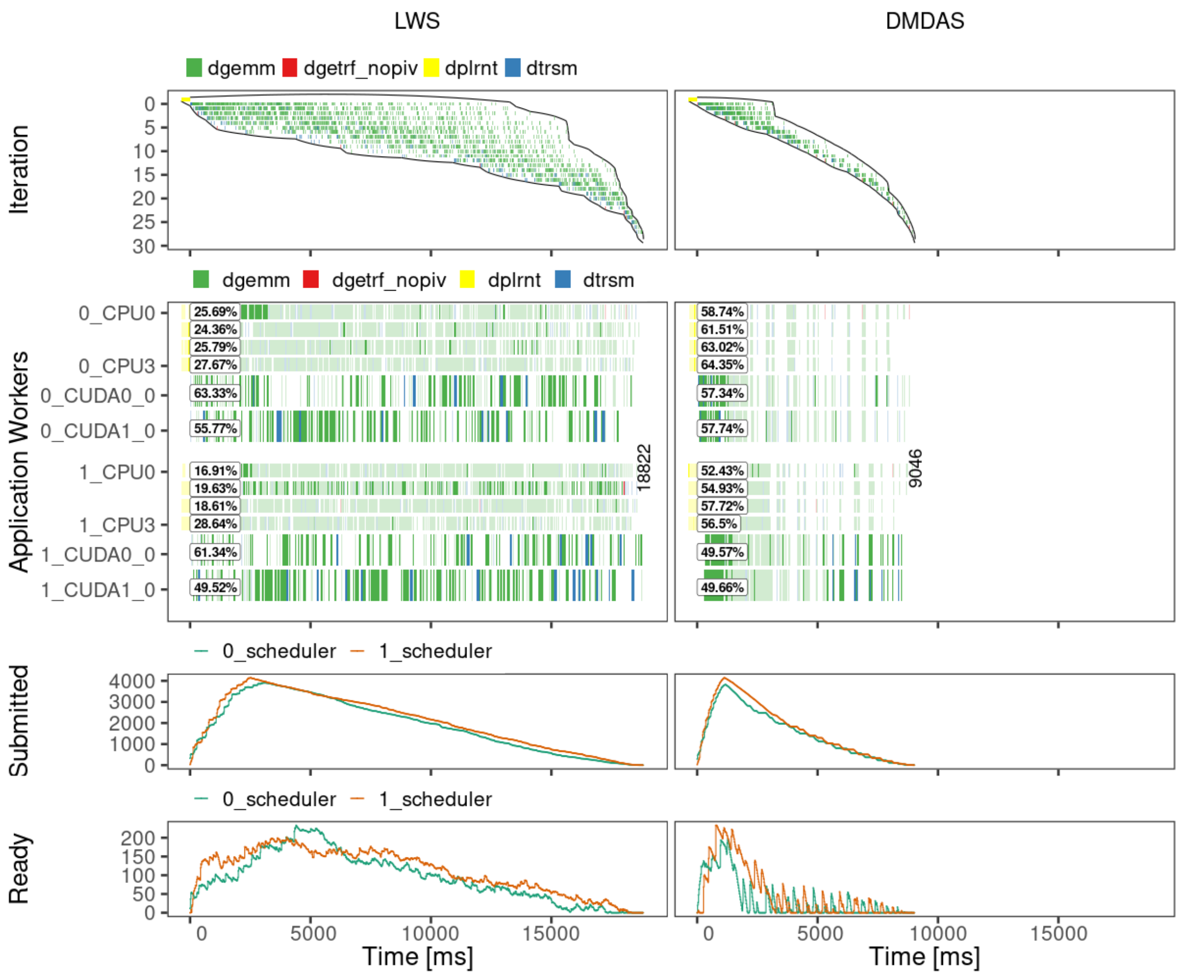


Figure 21: Comparing two traces using the assemble function.

5. Case studies

We present here case studies that illustrate the use of **StarVZ** in the performance analysis process. These case studies come from common linear algebra operations: the dense tiled LU factorization of **Chameleon** (Agullo *et al.* 2010), and the sparse QR factorization available in **QRMumps** (Agullo *et al.* 2013). We also present a proof-of-concept to show task-based OpenMP applications analyzed with our framework.

5.1. LU

Most of the prior visualizations used an execution of the LU factorization provided by **Chameleon** (Agullo *et al.* 2010). Figure 21 is a great example of a use-case utilization of **StarVZ**. The figure presents identical executions except for one parameter, the **StarPU** scheduler. The left plot utilizes the `lws` scheduler while the right one utilize the `dmdas` scheduler. The main difference is the total makespan from 18822 to 9046. The plots have the same horizontal time limit (manually configured), according to the worst execution. The explanation of these time differences is on the **StarVZ** plots. The idle time values per worker show that the GPUs with the `dmdas` scheduler have lower idle times than the `lws`. Because the GPUs are more computationally powerful, such utilization difference reflects on the final makespan a lot. Moreover, checking the runtime states, the number of idle states on the first half of the execution using the `dmdas` is lower than the `lws`, that has idle states on GPUs during all the execution. The final explanation relies on scheduler algorithms. The `dmdas` scheduler derives from the `heft` algorithm (Topcuoglu *et al.* 2002), it will greedily place a task on the worker with the lower time ending estimation considering data transfers. This behavior enables the `dmdas` algorithm to know when a worker will need some data and make more prefetches. On the other hand, the `lws` algorithm makes workers steal tasks from others when they do not have tasks to execute. Because they can steal a task that they do not have all the data, it will have to do a fetch, leading to lack of the positive effects of communication and computation overlap. Using the **StarVZ** workflow, we can also verify the raw values for fetches and prefetches for both cases using the following code. The `jss_sample_lu` is the `lws` case while the `jss_sample_lu_2` is the `dmdas` case. The `dmdas` case makes much more prefetches (4122) than fetches (677) while `lws` does fetches (4982) most of the time.

```
R> library("tidyverse")
R> jss_sample_lu$Link %>%
   filter(Type == "Intra-node data Fetch" |
          Type == "Intra-node data TaskPreFetch" |
          Type == "Intra-node data PreFetch ") %>%
   group_by(Type) %>%
   summarize(N = n(), .groups = "drop") %>% data.frame()

R> jss_sample_lu_2$Link %>%
   filter(Type == "Intra-node data Fetch" |
          Type == "Intra-node data TaskPreFetch" |
          Type == "Intra-node data PreFetch ") %>%
   group_by(Type) %>%
   summarize(N = n(), .groups ="drop")  %>% data.frame()
```

```
                          Type    N
1        Intra-node data Fetch 4982
2 Intra-node data TaskPreFetch   10

                          Type    N
1        Intra-node data Fetch  677
2 Intra-node data TaskPreFetch 4122
```

5.2. QRMumps

QRMumps (Agullo *et al.* 2013) is a task-based parallel sparse matrix factorization solver, built on top of **StarPU**. This solver implements the Multifrontal method (Duff and Reid 1983), a common approach to sparse solver parallelization. It breaks the factorization steps in smaller and denser sub-problems called frontal matrices, organized with an Elimination Tree structure. **QRMumps** dilutes the tree nodes in a fine-grained DAG of tasks, but some structure characteristics still influence the execution. For example, the communication between nodes and the available tree-level parallelism. We present in Figure 22 two executions where we only change the **StarPU** scheduler from `lws` (left) to `prio` (right). The scheduler affects the tree traversal and the moment where communication happens. For instance, when we analyze the `prio` scheduler, where the computational tasks have higher priority than the communication and initialization tasks, we can observe that the communication tasks are often executed just in the nodes' end. This effect limits the tree parallelism because the parent nodes have to wait for these communications to occur. In contrast, in the `lws` scheduler execution, we observe that the communication tasks are performed along with the computational tasks, enabling more tree parallelism, as we can observe in the Figure. Hence, such visualizations are interesting to verify the effects of the application and runtime configuration for a given workload, helping to understand and improve the application. They can be created as follows:

```
R> qr_lws <- starvz_read(directory = "data/qr-sched/lws/")
R> qr_prio <- starvz_read(directory = "data/qr-sched/prio/")
R> endTime <- max(c(qr_lws$Application$End, qr_prio$Application$End))
R> qr_lws$config$limits$end <- endTime
R> qr_lws$config$title$text <- "LWS"
R> qr_prio$config$title$text <- "PRIO"
R> starvz_assemble(starvz_plot_list(qr_lws),
   starvz_plot_list(qr_prio), config = qr_prio$config)
```

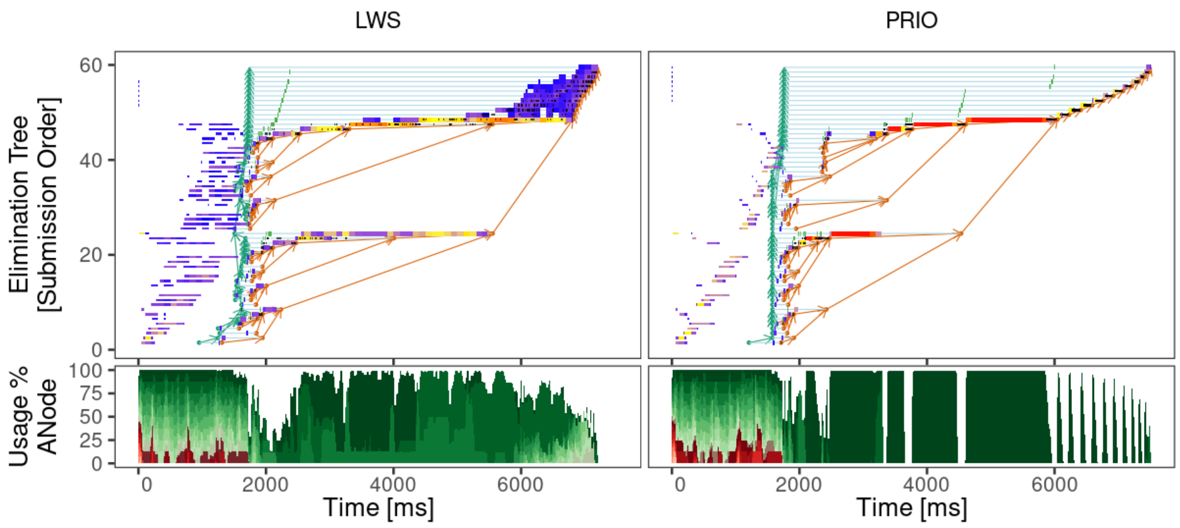


Figure 22: Execution comparison between lws (left) and prio (right) schedulers.

5.3. OpenMP SparseLU via KStar

StarPU can execute OpenMP programs with the support of the **KStar** (Agullo *et al.* 2017) source-to-source compiler. In this use-case, we select the **SparseLU** benchmark, which computes the LU factorization over a sparse matrix. This benchmark is part of the **KASTORS**

suite (Virouleau *et al.* 2014), which provides benchmarks with the recent OpenMP `task depend` construction. This primitive enables data-flow dependencies in OpenMP codes translating to fine-grained task-synchronizations as those obtained with the native **StarPU** API. The **SparseLU** benchmark stresses the runtime system scheduling capabilities since it presents intrinsic unbalanced loads due to the matrix's sparse structure. One can see in Figure 23 that all workers receive a similar workload. The ready tasks curve, although unstable, shows that the amount of ready tasks remains high enough to keep computing resources busy, resulting in low resource idleness.

```
R> starvz_phase1(directory = "data/kstar-sparselu")
R> jss_sample_kstar <- starvz_read(directory = "data/kstar-sparselu",
                                   selective = FALSE)
R> starvz_plot(jss_sample_kstar)
```

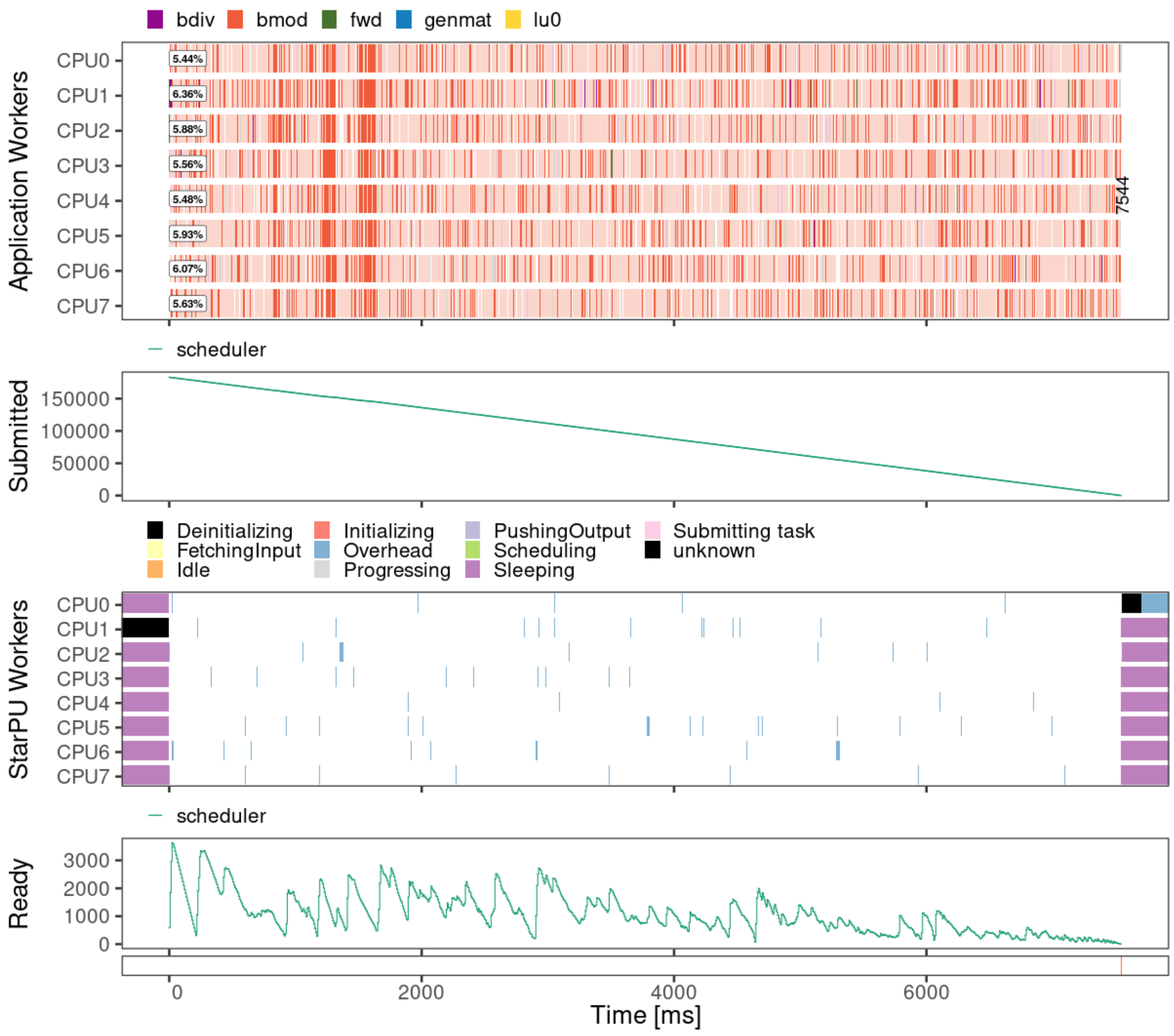


Figure 23: The visualization for the KStar/OpenMP SparseLU application.

6. Final remarks

The performance analysis is an essential step in the development and optimization of parallel applications. However, the complexity of applications and computing platforms makes the analysis process laborious and difficult. The **StarVZ** package helps to tackle such complexity in the context of task-based parallel applications. The framework relies on modern data science tools to provide a configurable analysis workflow, combining proposed visualizations panels and scripting capabilities. In this paper, we present the **StarVZ** R-package available on CRAN, its design and philosophy, including methodological aspects, and most of the functions that are part of the package. We demonstrate the usefulness of the **StarVZ** package by giving examples of its functionalities in three real task-based applications comparing runtimes settings, application configurations, and appropriateness for generic OpenMP parallel applications. An analyst can use the views of **StarVZ** and quickly adapt it for their needs, as it uses standard and modern R-packages, providing the data and panels in familiar structures, such as **tibble** and **ggplot** objects. Moreover, **StarVZ** provides means of data aggregation for the analysis scalability.

Although **StarVZ** already provides handy functionalities, it is possible to expand further to improve the methodology and workflow. Future work includes additional support for other runtimes, including native support for the OpenMP task dependencies without relying on the **KStar** for execution thought **StarPU**. We also intend to include support for nested tasks with preemption. While the visualization may be enough for experienced analysts, automatic reports may be useful for beginners in the field. As a consequence, we intend to include textual suggestions to cover known performance problems, such as wrong worker pinning, resource performance verification (NUMA, contention, variability), possible causes for idle times (late execution of tasks with satisfied dependencies), and lack of ready tasks by problems in the application DAG, granularity or communication.

7. Acknowledgments

We would like to thank our colleagues Arnaud Legrand, Luka Stanisic, Guilherme Alles and Rémy Drouilhet for all the discussion and help during the development of **StarVZ**. We also would like to thank the **StarPU** team, especially Samuel Thibault for his insights. This study was financed by the "Coordenação de Aperfeiçoamento de Pessoal de Nível Superior" (CAPES) - Finance Code 001, the CNPq - Brazil scholarship for the 1º (141971/2020-7) and 3º (131347/2019-5) author, and the projects: FAPERGS MultiGPU (16/354-8) and GreenCloud (16/488-9), the CNPq project 447311/2014-0, the CAPES/Brafitec EcoSud 182/15, the CAPES/Cofecub 899/18, Petrobras (2016/00133-9, 2018/00263-5). Some experiments are carried out in the PCAD infrastructure, http://gppd-hpc.inf.ufrgs.br, at INF/UFRGS.

References

Agullo E, Augonnet C, Dongarra J, Ltaief H, Namyst R, Thibault S, Tomov S (2010). "Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs." In W mei W Hwu (ed.), *GPU Computing Gems*, volume 2. Morgan Kaufmann.

Agullo E, Aumage O, Bramas B, Coulaud O, Pitoiset S (2017). "Bridging the Gap between OpenMP and Task-Based Runtime Systems for the Fast Multipole Method." *IEEE Transactions on Parallel and Distributed Systems*, **28**(10), 2794–2807. ISSN 10459219. doi:10.1109/TPDS.2017.2697857.

Agullo E, Buttari A, Guermouche A, Lopez F (2013). "Multifrontal QR Factorization for Multicore Architectures over Runtime Systems." In F Wolf, B Mohr, D an Mey (eds.), *Euro-Par 2013 Parallel Processing*, pp. 521–532. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-40047-6. doi:10.1007/978-3-642-40047-6_53.

Augonnet C, Thibault S, Namyst R, Wacrenier PA (2011). "**StarPU**: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures." *Concurrency and Computation: Practice and Experience*, **23**(2), 187–198. ISSN 1532-0626. doi:10.1002/cpe.1631.

Berkelaar M, *et al.* (2020). ***lpSolve**: Interface to 'lp_solve' v. 5.5 to Solve Linear/Integer Programs.* R package version 5.6.15, URL https://CRAN.R-project.org/package=lpSolve.

Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y (1996). "**Cilk**: An Efficient Multithreaded Runtime System." *Journal of Parallel and Distributed Computing*, **37**(1), 55–69. ISSN 0743-7315. doi:10.1006/jpdc.1996.0107.

Brinkmann S, Gracia J, Niethammer C (2013). "Task Debugging with **Temanejo**." In A Cheptsov, S Brinkmann, J Gracia, MM Resch, WE Nagel (eds.), *Tools for High Performance Computing 2012*, pp. 13–21. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-37349-7. doi:10.1007/978-3-642-37349-7_2.

Ceballos G, Grass T, Hugo A, Black-Schaffer D (2018). "Analyzing Performance Variation of Task Schedulers with **TaskInsight**." *Parallel Computing*, **75**, 11–27. ISSN 0167-8191. doi:10.1016/j.parco.2018.02.003.

Chapman B, Jost G, van der Pas R, Kuck D (2008). *Using OpenMP: Portable Shared Memory Parallel Programming.* Number vol. 10 in Scientific Computation Series, 3rd edition. Books24x7.com. ISBN 9780262533027.

Coulomb K, Degomme A, Faverge M, Trahay F (2012). "An Open-Source Tool-Chain for Performance Analysis." In H Brunst, MS Müller, WE Nagel, MM Resch (eds.), *Tools for High Performance Computing 2011*, pp. 37–48. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-31476-6.

de Kergommeaux JC, Stein B, Bernard P (2000). "Pajé, an Interactive Visualization Tool for Tuning Multi-Threaded Parallel Applications." *Parallel Computing*, **26**(10), 1253–1274. ISSN 0167-8191. doi:10.1016/S0167-8191(00)00010-7.

Duff IS, Reid JK (1983). "The Multifrontal Solution of Indefinite Sparse Symmetric Linear." *ACM Transactions on Mathematical Software (TOMS)*, **9**(3), 302–325. doi:10.1145/356044.356047.

Duran A, Ayguadé E, Badia RM, Labarta J, Martinell L, Martorell X, Planas J (2011). "**OmpSs**: a Proposal for Programming Heterogeneous Multi-Core Architectures." *Parallel Processing Letters*, **21**(2), 173–193. ISSN 1793-642X. doi:10.1142/s0129626411000151.

Eyraud-Dubois L (2019). ***pmtool**: Post-mortem Analysis Tool for starpu Scheduling Studies.* URL https://gitlab.inria.fr/eyrauddu/pmtool.

François R, Ooms J, Richardson N, Apache Arrow (2020). ***arrow**: Integration to 'Apache' 'Arrow'.* R package version 0.17.1, URL https://CRAN.R-project.org/package=arrow.

Garcia Pinto V, Mello Schnorr L, Stanisic L, Legrand A, Thibault S, Danjean V (2018). "A Visual Performance Analysis Framework for Task-Based Parallel Applications Running on Hybrid Clusters." *Concurrency and Computation: Practice and Experience*, **30**(18), e4472. doi:10.1002/cpe.4472.

Gautier T, Lima JV, Maillard N, Raffin B (2013). "**XKaapi**: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures." *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. doi:10.1109/ipdps.2013.66.

Glur C (2019). ***data.tree**: General Purpose Hierarchical Data Structure.* R package version 0.7.11, URL https://CRAN.R-project.org/package=data.tree.

Gropp W, Lusk E, Skjellum A (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* Scientific and Engineering Computation, 2nd edition. MIT Press. ISBN 9780262527392.

Hennessy J, Patterson D (2017). *Computer Architecture: A Quantitative Approach.* 6th edition. Elsevier Science. ISBN 978-0-128-11906-8.

Huynh A, Thain D, Pericàs M, Taura K (2015). "**DAGViz**: A DAG Visualization Tool for Analyzing Task-Parallel Program Traces." In *Proceedings of the 2nd Workshop on Visual Performance Analysis (VPA)*. ACM, ACM. ISBN 9781450340137. doi:10.1145/2835238.2835241.

Knüpfer A, Brunst H, Doleschal J, Jurenz M, Lieber M, Mickler H, Müller MS, Nagel WE (2008). "The **Vampir** Performance Analysis Tool-Set." In M Resch, R Keller, V Himmler, B Krammer, A Schulz (eds.), *Tools for High Performance Computing*, pp. 139–155. Springer, Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-68564-7.

Leandro Nesi L, Thibault S, Stanisic L, Mello Schnorr L (2019). "Visual Performance Analysis of Memory Behavior in a Task-Based Runtime on Hybrid Platforms." In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 142–151. IEEE/ACM. doi:10.1109/CCGRID.2019.00025.

Muddukrishna A, Jonsson PA, Podobas A, Brorsson M (2016). "**Grain Graphs**: OpenMP Performance Analysis Made Easy." In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, ACM. ISBN 9781450340922. doi:10.1145/2851141.2851156.

NVIDIA (2020). "CUDA C++ Programming Guide v11.0." *Technical report*, NVIDIA. URL https://docs.nvidia.com/cuda/pdf/CUDA%5FC%5FProgramming%5FGuide.pdf.

Pedersen TL (2020). ***patchwork**: The Composer of Plots.* R package version 1.0.1, URL https://CRAN.R-project.org/package=patchwork.

Pillet V, Labarta J, Cortes T, Girona S (1995). "**Paraver**: A Tool to Visualize and Analyze Parallel Code." In *Proceedings of WoTUG-18: transputer and occam developments*, volume 44, pp. 17–31. Citeseer.

Pinto VG, Stanisic L, Legrand A, Schnorr LM, Thibault S, Danjean V (2016). "Analyzing Dynamic Task-Based Applications on Hybrid Platforms: An Agile Scripting Approach." In *Third Workshop on Visual Performance Analysis, VPA@SC, Salt Lake, UT, USA*, pp. 17–24. IEEE, IEEE. doi:10.1109/VPA.2016.008.

OpenCL Working Group K (2020). "The OpenCL C 3.0 Specification." *Technical report*, Khronos Group Inc. URL https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL%5FC.pdf.

OpenMP Architecture Review Board (2018). "OpenMP API Specification: Version 5.0." *Technical report*, OpenMP ARB. URL https://www.openmp.org/spec-html/5.0/openmp.html.

R Core Team (2017). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Schnorr LM, Legrand A (2013). "Visualizing More Performance Data Than What Fits on Your Screen." In *Tools for High Performance Computing 2012*, pp. 149–162. Springer. doi:10.1007/978-3-642-37349-7_10.

Strohmaier E, Dongarra J, Simon H, Meuer M (2020). "Development over Time | TOP500 Supercomputer Sites." URL https://www.top500.org/statistics/overtime/.

Topcuoglu H, Hariri S, Min-You Wu (2002). "Performance-effective and low-complexity task scheduling for heterogeneous computing." *IEEE Transactions on Parallel and Distributed Systems*, **13**(3), 260–274. doi:10.1109/71.993206.

Urbanek S, Puchert A (2020). *OpenCL: Interface Allowing R to Use OpenCL*. R package version 0.2-1, URL https://CRAN.R-project.org/package=OpenCL.

Virouleau P, Brunet P, Broquedis F, Furmento N, Thibault S, Aumage O, Gautier T (2014). "Evaluation of OpenMP Dependent Tasks with the **KASTORS** Benchmark Suite." In L DeRose, B de Supinski, S Olivier, B Chapman, M Müller (eds.), *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pp. 16–29. Springer International Publishing. ISBN 978-3-319-11453-8. doi:10.1007/978-3-319-11454-5_2.

Wickham H (2016). ***ggplot2**: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-3-319-24277-4. URL https://ggplot2.tidyverse.org.

Wickham H, Averick M, Bryan J, Chang W, McGowan LD, François R, Grolemund G, Hayes A, Henry L, Hester J, Kuhn M, Pedersen TL, Miller E, Bache SM, Müller K, Ooms J, Robinson D, Seidel DP, Spinu V, Takahashi K, Vaughan D, Wilke C, Woo K, Yutani H (2019). "Welcome to the **tidyverse**." *Journal of Open Source Software*, **4**(43), 1686. doi:10.21105/joss.01686.

Wickham H, François R, Henry L, Müller K (2020). ***dplyr**: A Grammar of Data Manipulation*. R package version 1.0.0, URL https://CRAN.R-project.org/package=dplyr.

Wilson JM (2003). "Gantt Charts: A Centenary Appreciation." In *European Journal of Operational Research*, volume 149, pp. 430–437. Elsevier, North-Holland. ISSN 3772217. doi:10.1016/S0377-2217(02)00769-5.

Yu H (2002). "**Rmpi**: Parallel Statistical Computing in R." *R News*, **2**(2), 10–14. URL https://cran.r-project.org/doc/Rnews/Rnews%5F2002-2.pdf.

Affiliation:

Lucas Leandro Nesi, Vinicius Garcia Pinto
1 Institute of Informatics – Federal University of Rio Grande do Sul – UFRGS
Av. Bento Gonçalves, 9500, Porto Alegre, Brazil
2 Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG
38000 Grenoble, France
E-mail: {llnesi, vgpinto}@inf.ufrgs.br

Marcelo Cogo Miletto, Lucas Mello Schnorr
1 Institute of Informatics – Federal University of Rio Grande do Sul – UFRGS
Av. Bento Gonçalves, 9500, Porto Alegre, Brazil
E-mail: {mmiletto, schnorr}@inf.ufrgs.br

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd