StarPU Handbook - StarPU Performances

for StarPU 1.4.9

Generated by Doxygen.

This manual documents the usage of StarPU version 1.4.9. Its contents was last updated on 2025-10-24.

Chapter 1

Organization

This part shows how to measure application performances.

- Chapter Benchmarking StarPU introduces some interesting benchmarks which can be found in StarPU sources.
- Chapter Online Performance Tools gives information on online performance monitoring tools to help you analyze your program
- Chapter Offline Performance Tools gives information on offline performance tools such as a FxT library to trace execution data and tasks and a StarPU Eclipse Plugin to visualize data traces directly from the Eclipse IDE.

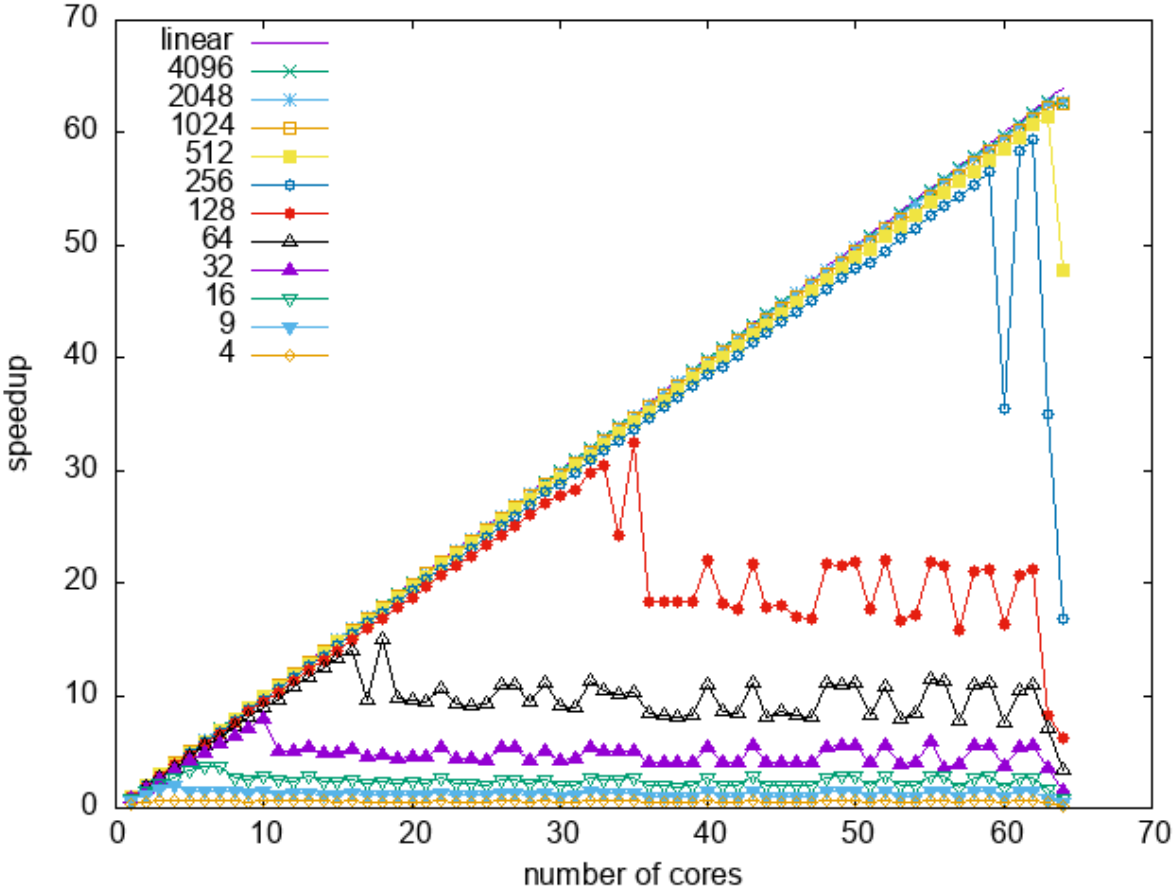
Chapter 2

Benchmarking StarPU

Some interesting benchmarks are installed among examples in `$STARPU_PATH/lib/starpu/examples/`. Make sure to try various schedulers, for instance `STARPU_SCHED=dmda`.

2.1 Task Size Overhead

This benchmark gives a glimpse into how long a task should be (in µs) for StarPU overhead to be low enough to keep efficiency. Running `tasks_size_overhead.sh` generates a plot of the speedup of tasks of various sizes, depending on the number of CPUs being used.



2.2 Data Transfer Latency

`local_pingpong` performs a ping-pong between the first two CUDA nodes, and prints the measured latency.

2.3 Matrix-Matrix Multiplication

`sgemm` and `dgemm` perform a blocked matrix-matrix multiplication using BLAS and cuBLAS. They output the obtained GFlops.

2.4 Cholesky Factorization

`cholesky_*` perform a Cholesky factorization (single precision). They use different dependency primitives.

2.5 LU Factorization

`lu_*` perform an LU factorization. They use different dependency primitives.

2.6 Simulated Benchmarks

It can also be convenient to try simulated benchmarks, if you want to give a try at CPU-GPU scheduling without actually having a GPU at hand. This can be done by using the SimGrid version of StarPU: first install the SimGrid simulator from https://simgrid.org/ (we tested with SimGrid from 3.11 to 3.16, and 3.18 to 3.30. SimGrid versions 3.25 and above need to be configured with `-Denable_msg=ON`. Other versions may have compatibility issues, 3.17 notably does not build at all. MPI simulation does not work with version 3.22). Then configure StarPU with --enable-simgrid and rebuild and install it, and then you can simulate the performance for a few virtualized systems shipped along StarPU: attila, mirage, idgraf, and sirocco.
For instance:

```
$ export STARPU_PERF_MODEL_DIR=$STARPU_PATH/share/starpu/perfmodels/sampling
$ export STARPU_HOSTNAME=attila
$ $STARPU_PATH/lib/starpu/examples/cholesky_implicit -size $((960*20)) -nblocks 20
```

Will show the performance of the cholesky factorization with the attila system. It will be interesting to try with different matrix sizes and schedulers.
Performance models are available for `cholesky_*`, `lu_*`, `*gemm`, with block sizes 320, 640, or 960 (plus 1440 for sirocco), and for `stencil` with block size 128x128x128, 192x192x192, and 256x256x256.
Read Chapter SimGridSupport for more information on the SimGrid support.

Chapter 3

Online Performance Tools

3.1 On-line Performance Feedback

Some examples which apply online performance monitoring are in the directory `tests/perfmodels/`

3.1.1 Enabling On-line Performance Monitoring

In order to enable online performance monitoring, the application can call starpu_profiling_status_set() with the parameter STARPU_PROFILING_ENABLE. It is possible to detect whether monitoring is already enabled or not by calling starpu_profiling_status_get(). Enabling monitoring also reinitialize all previously collected feedback. The environment variable STARPU_PROFILING can also be set to `1` to achieve the same effect. The function starpu_profiling_init() can also be called during the execution to reinitialize performance counters and to start the profiling if the environment variable STARPU_PROFILING is set to `1`.
Likewise, performance monitoring is stopped by calling starpu_profiling_status_set() with the parameter STARPU_PROFILING_DISABLE. Note that this does not reset the performance counters so that the application may consult them later on.
More details about the performance monitoring API are available in Profiling.

3.1.2 Per-task Feedback

If profiling is enabled, a pointer to a structure starpu_profiling_task_info is put in the field starpu_task::profiling_info when a task terminates. This structure is automatically destroyed when the task structure is destroyed, either automatically or by calling starpu_task_destroy().
The structure starpu_profiling_task_info indicates the date when the task was submitted (starpu_profiling_task_info::submit_time), started (starpu_profiling_task_info::start_time), and terminated (starpu_profiling_task_info::end_time), relative to the initialization of StarPU with starpu_init(). User can call starpu_timing_timespec_delay_us() to calculate the time elapsed between start time and end time in microseconds. It also specifies the identifier of the worker that has executed the task (starpu_profiling_task_info::workerid). These dates are stored as `timespec` structures which users may convert into micro-seconds using the helper function starpu_timing_timespec_to_us(). User can call starpu_worker_get_current_task_exp_end() to get the date when the current task is expected to be finished.
When ::STARPU_ENERGY_PROFILING is enabled, starpu_profiling_task_info::energy_consumed, provides the amount of Joules used by the task.
It is worth noting that the application may directly access this structure from the callback executed at the end of the task. The structure starpu_task associated to the callback currently being executed is indeed accessible with the function starpu_task_get_current().

3.1.3 Per-codelet Feedback

The field starpu_codelet::per_worker_stats is an array of counters. Unless the STARPU_CODELET_PROFILING environment variable was set to 0, the `i`-th entry of the array is incremented every time a task implementing the codelet is executed on the `i`-th worker. This array is not reinitialized when profiling is enabled or disabled. The function starpu_codelet_display_stats() can be used to display the execution statistics of a specific codelet.

3.1.4 Per-worker Feedback

The second argument returned by the function starpu_profiling_worker_get_info() is a structure starpu_profiling_worker_info that gives statistics about the specified worker. This structure specifies:

- In starpu_profiling_worker_info::start_time, when StarPU started collecting profiling information for that worker.
- In starpu_profiling_worker_info::total_time, the duration of the profiling measurement interval.
- In starpu_profiling_worker_info::executed_tasks, the number of tasks that were executed while profiling was enabled.

It also specifies how much time was spent in various states (executing a task, executing a callback, waiting for a data transfer to complete, etc.). Since these can happen at the same time (waiting for a data transfer while executing the previous tasks, and scheduling the next task), we provide two views. Firstly, the "all" view:

- In starpu_profiling_worker_info::all_executing_time, the time spent executing kernels, thus real useful work.
- In starpu_profiling_worker_info::all_callback_time, the time spent executing application callbacks.
- In starpu_profiling_worker_info::all_waiting_time, the time spent waiting for data transfers.
- In starpu_profiling_worker_info::all_sleeping_time, the time spent during which there was no task to be executed, i.e. lack of parallelism.
- In starpu_profiling_worker_info::all_scheduling_time, the time spent scheduling tasks.

But these times overlap, notably with GPUs the schedulers runs while tasks are getting executed. Another view is the "split" view, which eliminates the overlapping, by considering for instance that it does not matter what is happening while tasks are getting executed, that should be accounted for "executing" time, and e.g. only the scheduling periods that happen while no task is getting executed should be accounted in "scheduling" time. More precisely:

- In starpu_profiling_worker_info::executing_time, the time spent executing kernels, normally equal to starpu_profiling_worker_info::all_executing_time.
- In starpu_profiling_worker_info::callback_time, the time spent executing application callbacks while not executing a task.
- In starpu_profiling_worker_info::waiting_time, the time spent waiting for data transfers while not executing a task or a callback.
- In starpu_profiling_worker_info::sleeping_time, the time spent during which there was no task to be executed and not executing a task or a callback or waiting for a data transfer, i.e. real lack of parallelism.
- In starpu_profiling_worker_info::scheduling_time, the time spent scheduling tasks while not executing a task or a callback or waiting for a data transfer to finish, and there are tasks to be scheduled.

This thus provides a split of the starpu_profiling_worker_info::total_time into various states. The difference between starpu_profiling_worker_info::total_time and the sum of this split is the remaining uncategorized overhead of the runtime.
Calling starpu_profiling_worker_get_info() resets the profiling information associated to a worker.
To easily display all this information, the environment variable STARPU_WORKER_STATS can be set to 1 (in addition to setting STARPU_PROFILING to 1). A summary will then be displayed at program termination. To display the summary in a file instead of the standard error stream, use the environment variable STARPU_WORKER_↩ STATS_FILE.

```
Worker stats:
CUDA 0.0 (Tesla M2075 4.7 GiB 03:00.0)
        133 task(s)
        time split: total 3212.86 ms = executing: 1588.56 ms + callback: 2.95 ms + waiting: 5.34 ms + sleeping
        all time: executing: 1588.56 ms callback: 2.95 ms waiting: 22.83 ms sleeping: 1725.93 ms scheduling: 1
        286.388333 GFlop/s

CPU 0
        10 task(s)
```

```
        time split: total 3212.89 ms = executing: 2117.19 ms + callback: 0.23 ms + waiting: 0.01 ms + sleeping
        all time: executing: 2117.19 ms callback: 0.23 ms waiting: 0.01 ms sleeping: 1095.06 ms scheduling: 28
        22.029695 GFlop/s

CPU 1
        10 task(s)
        time split: total 3212.92 ms = executing: 2116.18 ms + callback: 0.17 ms + waiting: 0.01 ms + sleeping
        all time: executing: 2116.18 ms callback: 0.17 ms waiting: 0.01 ms sleeping: 1096.10 ms scheduling: 28
        22.029487 GFlop/s

CPU 2
        10 task(s)
        time split: total 3212.94 ms = executing: 2116.08 ms + callback: 0.18 ms + waiting: 0.01 ms + sleeping
        all time: executing: 2116.08 ms callback: 0.18 ms waiting: 0.01 ms sleeping: 1096.21 ms scheduling: 28
        22.029343 GFlop/s


Global time split: total 12851.60 ms = executing: 7938.01 ms (61.77%) + callback: 3.53 ms (0.03%) + waiting: 5
```

The number of GFlops/s is available because the starpu_task::flops field of the tasks were filled (or STARPU_FLOPS used in starpu_task_insert()).
When an FxT trace is generated (see Generating Traces With FxT), it is also possible to use the tool `starpu_workers_activity` (see Monitoring Activity) to generate a graphic showing the evolution of these values during the time, for the different workers.

3.1.5 Bus-related Feedback

The bus speed measured by StarPU can be displayed by using the tool `starpu_machine_display`, for instance:

```
StarPU has found:
        3 CUDA devices
                CUDA 0 (Tesla C2050 02:00.0)
                CUDA 1 (Tesla C2050 03:00.0)
                CUDA 2 (Tesla C2050 84:00.0)
from    to RAM          to CUDA 0       to CUDA 1       to CUDA 2
RAM     0.000000        5176.530428     5176.492994     5191.710722
CUDA 0  4523.732446     0.000000        2414.074751     2417.379201
CUDA 1  4523.718152     2414.078822     0.000000        2417.375119
CUDA 2  4534.229519     2417.069025     2417.060863     0.000000
```

Statistics about the data transfers which were performed and temporal average of bandwidth usage can be obtained by setting the environment variable STARPU_BUS_STATS to `1`; a summary will then be displayed at program termination. To display the summary in a file instead of the standard error stream, use the environment variable STARPU_BUS_STATS_FILE.

```
Data transfer stats:
        RAM 0 -> CUDA 0 319.92 MB       213.10 MB/s     (transfers : 91 - avg 3.52 MB)
        CUDA 0 -> RAM 0 214.45 MB       142.85 MB/s     (transfers : 61 - avg 3.52 MB)
        RAM 0 -> CUDA 1 302.34 MB       201.39 MB/s     (transfers : 86 - avg 3.52 MB)
        CUDA 1 -> RAM 0 133.59 MB       88.99 MB/s      (transfers : 38 - avg 3.52 MB)
        CUDA 0 -> CUDA 1        144.14 MB       96.01 MB/s      (transfers : 41 - avg 3.52 MB)
        CUDA 1 -> CUDA 0        130.08 MB       86.64 MB/s      (transfers : 37 - avg 3.52 MB)
        RAM 0 -> CUDA 2 312.89 MB       208.42 MB/s     (transfers : 89 - avg 3.52 MB)
        CUDA 2 -> RAM 0 133.59 MB       88.99 MB/s      (transfers : 38 - avg 3.52 MB)
        CUDA 0 -> CUDA 2        151.17 MB       100.69 MB/s     (transfers : 43 - avg 3.52 MB)
        CUDA 2 -> CUDA 0        105.47 MB       70.25 MB/s      (transfers : 30 - avg 3.52 MB)
        CUDA 1 -> CUDA 2        175.78 MB       117.09 MB/s     (transfers : 50 - avg 3.52 MB)
        CUDA 2 -> CUDA 1        203.91 MB       135.82 MB/s     (transfers : 58 - avg 3.52 MB)
Total transfers: 2.27 GB
```

3.1.6 MPI-related Feedback

Statistics about the data transfers which were performed over MPI can be obtained by setting the environment variable STARPU_MPI_STATS to `1`; a summary will then be displayed at program termination:

```
[starpu_comm_stats][1] TOTAL:   456.000000 B    0.000435 MB     0.000188 B/s    0.000000 MB/s
[starpu_comm_stats][1:0]        456.000000 B    0.000435 MB     0.000188 B/s    0.000000 MB/s
```

```
[starpu_comm_stats][0] TOTAL:   456.000000 B    0.000435 MB      0.000188 B/s    0.000000 MB/s
[starpu_comm_stats][0:1]        456.000000 B    0.000435 MB      0.000188 B/s    0.000000 MB/s
```

These statistics can be plotted as heatmaps using StarPU tool `starpu_mpi_comm_matrix.py` (see MPIDebug).

3.2 Task And Worker Profiling

A full example showing how to use the profiling API is available in the StarPU sources in the directory `examples/profiling/`.

```
struct starpu_task *task = starpu_task_create();
task->cl = &cl;
task->synchronous = 1;
/* We will destroy the task structure by hand so that we can
 * query the profiling info before the task is destroyed.  */
task->destroy = 0;
/* Submit and wait for completion (since synchronous was set to 1) */
starpu_task_submit(task);
/* The task is finished, get profiling information */
struct starpu_profiling_task_info *info = task->profiling_info;
/* How much time did it take before the task started ?  */
double delay += starpu_timing_timespec_delay_us(&info->submit_time, &info->start_time);
/* How long was the task execution ?  */
double length += starpu_timing_timespec_delay_us(&info->start_time, &info->end_time);
/* We no longer need the task structure */
starpu_task_destroy(task);
/* Display the occupancy of all workers during the test */
int worker;
for (worker = 0; worker < starpu_worker_get_count(); worker++)
{
        struct starpu_profiling_worker_info worker_info;
        int ret = starpu_profiling_worker_get_info(worker, &worker_info);
        STARPU_ASSERT(!ret);
        double total_time = starpu_timing_timespec_to_us(&worker_info.total_time);
        double executing_time = starpu_timing_timespec_to_us(&worker_info.executing_time);
        double sleeping_time = starpu_timing_timespec_to_us(&worker_info.sleeping_time);
        double overhead_time = total_time - executing_time - sleeping_time;
        float executing_ratio = 100.0*executing_time/total_time;
        float sleeping_ratio = 100.0*sleeping_time/total_time;
        float overhead_ratio = 100.0 - executing_ratio - sleeping_ratio;
        char workername[128];
        starpu_worker_get_name(worker, workername, 128);
        fprintf(stderr, "Worker %s:\n", workername);
        fprintf(stderr, "\ttotal time:  %.2lf ms\n", total_time*1e-3);
        fprintf(stderr, "\texec time:  %.2lf ms (%.2f %%)\n", executing_time*1e-3, executing_ratio);
        fprintf(stderr, "\tblocked time:  %.2lf ms (%.2f %%)\n", sleeping_time*1e-3, sleeping_ratio);
        fprintf(stderr, "\toverhead time:  %.2lf ms (%.2f %%)\n", overhead_time*1e-3, overhead_ratio);
}
```

3.3 Performance Model Example

To achieve good scheduling, StarPU scheduling policies need to be able to estimate in advance the duration of a task. This is done by giving to codelets a performance model, by defining a structure starpu_perfmodel and providing its address in the field starpu_codelet::model. The fields starpu_perfmodel::symbol and starpu_perfmodel::type are mandatory, to give a name to the model, and the type of the model, since there are several kinds of performance models. Then starpu_task_get_model_name() can be called to retrieve the name of the performance model associated with a task. For compatibility, make sure to initialize the whole structure to zero, either by using explicit memset(), or by letting the compiler implicitly do it as examplified below.

- Measured at runtime (model type STARPU_HISTORY_BASED). This assumes that for a given set of data input/output sizes, the performance will always be about the same. This is very true for regular kernels on GPUs for instance (<0.1% error), and just a bit less true on CPUs (~=1% error). This also assumes that there are few different sets of data input/output sizes. StarPU will then keep record of the average time of previous executions on the various processing units, and use it as an estimation. History is done per task size, by using a hash of the input and output sizes as an index. It will also save it in `$STARPU_HOME/.starpu/sampling/codelets` for further executions, and can be observed by using the tool `starpu_perfmodel_display`, or drawn by using the tool `starpu_perfmodel_plot` (PerformanceModelCalibration). The models are indexed by machine name. To share the models between machines (e.g. for a homogeneous cluster), use `export STARPU_HOSTNAME=some_global_name`. Measurements are only done when using a task scheduler which makes use of it,

such as dmda. Measurements can also be provided explicitly by the application, by using the function starpu_perfmodel_update_history(). An example is in the file tests/perfmodels/feed.c.

The following is a small code example.

If e.g. the code is recompiled with other compilation options, or several variants of the code are used, the symbol string should be changed to reflect that, in order to recalibrate a new model from zero. The symbol string can even be constructed dynamically at execution time, as long as this is done before submitting any task using it.

```
static struct starpu_perfmodel mult_perf_model =
{
    .type = STARPU_HISTORY_BASED,
    .symbol = "mult_perf_model"
};
struct starpu_codelet cl =
{
    .cpu_funcs = { cpu_mult },
    .cpu_funcs_name = { "cpu_mult" },
    .nbuffers = 3,
    .modes = { STARPU_R, STARPU_R, STARPU_W },
    /* for the scheduling policy to be able to use performance models */
    .model = &mult_perf_model
};
```

- Measured at runtime and refined by regression (model types STARPU_REGRESSION_BASED and STARPU_NL_REGRESSION_BASED). This still assumes performance regularity, but works with various data input sizes, by applying regression over observed execution times. STARPU_REGRESSION_BASED uses an a*n^b regression form, STARPU_NL_REGRESSION_BASED uses an a*n^b+c (more precise than STARPU_REGRESSION_BASED, but costs a lot more to compute).

  For instance, tests/perfmodels/regression_based.c uses a regression-based performance model for the function memset().

  Of course, the application has to issue tasks with varying size so that the regression can be computed. StarPU will not trust the regression unless there is at least 10% difference between the minimum and maximum observed input size. It can be useful to set the environment variable STARPU_CALIBRATE to 1 and run the application on varying input sizes with STARPU_SCHED set to dmda scheduler, to feed the performance model for a variety of inputs. The application can also provide the measurements explicitly by using the function starpu_perfmodel_update_history(). The tools starpu_perfmodel_display and starpu_perfmodel_plot can be used to observe how much the performance model is calibrated (PerformanceModelCalibration); when their output looks good, STARPU_CALIBRATE can be reset to 0 to let StarPU use the resulting performance model without recording new measures, and STARPU_SCHED can be set to dmda to benefit from the performance models. If the data input sizes vary a lot, it is really important to set STARPU_CALIBRATE to 0, otherwise StarPU will continue adding the measures, and result with a very big performance model, which will take time a lot of time to load and save.

  For non-linear regression, since computing it is quite expensive, it is only done at termination of the application. This means that the first execution of the application will use only history-based performance model to perform scheduling, without using regression.

- Another type of model is STARPU_MULTIPLE_REGRESSION_BASED, which is based on multiple linear regression. In this model, users define both the relevant parameters and the equation for computing the task duration.

$$T_{kernel} = a + b(M^{\alpha_1} * N^{\beta_1} * K^{\gamma_1}) + c(M^{\alpha_2} * N^{\beta_2} * K^{\gamma_2}) + ...$$

  M, N, K are the parameters of the task, added at the task creation. These need to be extracted by the cl_perf_func function, which should be defined by users. α, β, γ are the exponents defined by users in model->combinations table. Finally, coefficients a, b, c are computed automatically by the StarPU at the end of the execution, using least squares method of the dgels_ LAPACK function.

  examples/mlr/mlr.c example provides more details on the usage of STARPU_MULTIPLE_REGRESSION_BASED models. The --enable-mlr configure option needs to be set to calibrate the model.

  Coefficients computation is done at the end of the execution, and the results are stored in standard codelet perfmodel files. Additional files containing the duration of tasks together with the value of each parameter are stored in .starpu/sampling/codelets/tmp/ directory. These files are reused when STARPU_↩ CALIBRATE environment variable is set to 1, to recompute coefficients based on the current, but also on

the previous executions. By default, StarPU uses a lightweight dgels implementation, but the --enable-mlr-system-blas configure option can be used to make StarPU use a system-provided dgels BLAS.

Additionally, when multiple linear regression models are not enabled through --enable-mlr or when the `model->combinations` are not defined, StarPU will still write output files into `.starpu/sampling/codelets/tmp/` to allow performing an analysis. This analysis typically aims at finding the most appropriate equation for the codelet and `tools/starpu_mlr_analysis` script provides an example of how to perform such study.

- Provided as an estimation from the application itself (model type STARPU_COMMON and field starpu_perfmodel::cost_function), see for instance `examples/common/blas_model.h` and `examples/common/blas_model.c`.
- Provided explicitly by the application (model type STARPU_PER_ARCH): either field starpu_perfmodel::arch_cost_function, or the fields `.per_arch[arch][nimpl].cost_function` have to be filled with pointers to functions which return the expected duration of the task in micro-seconds, one per architecture, see for instance `tests/datawizard/locality.c`
- Provided explicitly by the application (model type STARPU_PER_WORKER) similarly with the starpu_perfmodel::worker_cost_function field.

For STARPU_HISTORY_BASED, STARPU_REGRESSION_BASED, and STARPU_NL_REGRESSION_BASED, the dimensions of task data (both input and output) are used as an index by default. STARPU_HISTORY_BASED uses a CRC hash of the dimensions as an index to distinguish histories, and STARPU_REGRESSION_BASED and STARPU_NL_REGRESSION_BASED use the total size as an index for the regression. (Data marked with STARPU_NOFOOTPRINT are not taken into account).
The starpu_perfmodel::size_base and starpu_perfmodel::footprint fields however permit the application to override that, when for instance some of the data do not matter for task cost (e.g. mere reference table), or when using sparse structures (in which case it is the number of non-zeros which matter), or when there is some hidden parameter such as the number of iterations, or when the application actually has a very good idea of the complexity of the algorithm, and just not the speed of the processor, etc. The example in the directory `examples/pi` uses this to include the number of iterations in the base size. starpu_perfmodel::size_base should be used when the variance of the actual performance is known (i.e. bigger return value is longer execution time), and thus particularly useful for STARPU_REGRESSION_BASED or STARPU_NL_REGRESSION_BASED. starpu_perfmodel::footprint can be used when the variance of the actual performance is unknown (irregular performance behavior, etc.), and thus only useful for STARPU_HISTORY_BASED. starpu_task_data_footprint() can be used as a base and combined with other parameters through starpu_hash_crc32c_be() for instance.
StarPU will automatically determine when the performance model is calibrated, or rather, it will assume the performance model is calibrated until the application submits a task for which the performance can not be predicted. For STARPU_HISTORY_BASED, StarPU will require 10 (STARPU_CALIBRATE_MINIMUM) measurements for a given size before estimating that an average can be taken as estimation for further executions with the same size. For STARPU_REGRESSION_BASED and STARPU_NL_REGRESSION_BASED, StarPU will require 10 (STARPU_CALIBRATE_MINIMUM) measurements, and that the minimum measured data size is smaller than 90% of the maximum measured data size (i.e. the measurement interval is large enough for a regression to have a meaning). Calibration can also be forced by setting the STARPU_CALIBRATE environment variable to `1`, or even reset by setting it to `2`.
How to use schedulers which can benefit from such performance model is explained in TaskSchedulingPolicy.
The same can be done for task energy consumption estimation, by setting the field starpu_codelet::energy_model the same way as the field starpu_codelet::model. Note: for now, the application has to give to the energy consumption performance model a name which is different from the execution time performance model.
The application can request time estimations from the StarPU performance models by filling a task structure as usual without actually submitting it. The data handles can be created by calling any of the functions `starpu_*_data_register` with a `NULL` pointer and `-1` node and the desired data sizes, and need to be unregistered as usual. The functions starpu_task_expected_length() and starpu_task_expected_energy() can then be called to get an estimation of the task cost on a given arch. starpu_task_footprint() can also be used to get the footprint used for indexing history-based performance models. starpu_task_destroy() needs to be called to destroy the dummy task afterwards. See `tests/perfmodels/regression_based.c` for an example.
The application can also request an on-the-fly XML report of the performance model, by calling starpu_perfmodel_dump_xml() to print the report to a `FILE*`.

3.4 Performance Monitoring Counters

This section presents the StarPU performance monitoring framework. It summarizes the objectives of the framework. It then introduces the entities involved in the framework. It presents the API of the framework, as well as some implementation details. It exposes the typical sequence of operations to plug an external tool to monitor a performance counter of StarPU.

3.4.1 Objectives

The objectives of this framework are to let external tools interface with StarPU to collect various performance metrics at runtime, in a generic, safe, extensible way. For that, it enables such tools to discover the available performance metrics in a particular StarPU build, as well as the type of each performance counter value. It lets these tools build sets of performance counters to monitor, and then register listener callbacks to collect the measurement samples of these sets of performance counters at runtime.

3.4.2 Entities

The performance monitoring framework is built on a series of concepts and items, organized consistently. The corresponding C language objects should be considered opaque by external tools, and should only be manipulated through proper function calls and accessors.

3.4.2.1 Performance Counter

The performance counter entity is the fundamental object of the framework, representing one piece of performance metrics, such as for instance the total number of tasks submitted so far, that is exported by StarPU and can be collected through the framework at runtime. A performance counter has a type and belongs to a scope. A performance counter is designated by a unique name and unique ID integer. We can start or stop collecting performance counter values by using starpu_perf_counter_collection_start() and starpu_perf_counter_collection_stop().

3.4.2.2 Performance Counter Type

A performance counter has a type. A type is designated by a unique name and unique ID number. Currently, supported types include:

Type Name	Type Definition
"int32"	32-bit signed integers
"int64"	64-bit signed integers
"float"	32-bit single-precision floating point
"double"	64-bit double-precision floating point

3.4.2.3 Performance Counter Scope

A performance counter belongs to a scope. The scope of a counter defines the context considered for computing the corresponding performance counter. A scope is designated with a unique name and unique ID number. Currently, defined scopes include:

Scope Name	Scope Definition
"global"	Counter is global to the StarPU instance
"per_worker"	Counter is within the scope of a thread worker
"per_codelet"	Counter is within the scope of a task codelet

3.4.2.4 Performance Counter Set

A performance counter set is a subset of the performance counters belonging to the same scope. Each counter of the scope can be in the enabled or disabled state in a performance counter set. A performance counter set enables a performance monitoring tool to indicate the set of counters to be collected for a particular listener callback.

3.4.2.5 Performance Counter Sample

A performance counter sample corresponds to one sample of collected measurement values of a performance counter set. Only the values corresponding to enabled counters in the sample's counter set should be observed by the listener callback. Whether the sample contains valid values for counters disabled in the set is unspecified.

3.4.2.6 Performance Counter Listener

A performance counter listener is a callback function registered by some external tool to monitor a set of performance counters in a particular scope. It is called each time a new performance counter sample is ready to be observed. The sample object should not be accessed outside the callback.

3.4.2.7 Application Programming Interface

The API of the performance monitoring framework is defined in the starpu_perf_monitoring.h public header file of StarPU. This header file is automatically included with starpu.h. An example of use of the routines is given in Sequence of operations.

3.4.3 Implementation Details

3.4.3.1 Performance Counter Registration

Each module of StarPU can export performance counters. In order to do so, modules that need to export some counters define a registration function that is called at StarPU initialization time. This function is responsible for calling the "_starpu_perf_counter_register()" function once for each counter it exports, to let the framework know about the list of counters managed by the module. It also registers performance sample updater callbacks for the module, one for each scope for which it exports counters.

3.4.3.2 Performance Sample Updaters

The updater callback for a module and scope combination is internally called every time a sample for a set of performance counter must be updated. Thus, the updated callback is responsible for filling the sample's selected counters with the counter values found at the time of the call. Global updaters are currently called at task submission time, as well as any blocking tasks management function of the StarPU API, such as starpu_task_wait_for_all(), which waits for the completion of all tasks submitted up to this point. Per-worker updaters are currently called at the level of StarPU's drivers, that is, the modules in charge of task execution of hardware-specific worker threads. The actual calls occur in-between the execution of tasks. Per-codelet updaters are currently called both at task submission time, and at the level of StarPU's drivers together with the per-worker updaters.
A performance sample object is locked during the sample collection. The locking prevents the following issues:

- The listener of sample being changed during sample collection;
- The set of counters enabled for a sample being changed;
- Conflicting concurrent updates;
- Updates while the sample is being read by the listener.

The location of the updaters' calls is chosen to minimize the sequentialization effect of the locking, in order to limit the level of interference of the monitoring process. For Global updaters, the calls are performed only on the application thread(s) in charge of submitting tasks. Since, in most cases, only a single application thread submits tasks, the sequentialization effect is moderate. Per-worker updates are local to their worker, thus here again the sample lock is un-contented, unless the external monitoring tool frequently changes the set of enabled counters in the sample.

3.4.3.3 Counter operations

In practice, the sample updaters only take snapshots of the actual performance counters. The performance counters themselves are updated with ad-hoc procedures depending on each counter. Such procedures typically involve atomic operations. While operations such as atomic increments or decrements on integer values are readily available, this is not the case for more complex operations such as min/max for computing peak value counters (for instance in the global and per-codelet counters for peak number of submitted tasks and peak number of ready

tasks waiting for execution), and this is also not the case for computations on floating point data (used for instance in computing cumulated execution time of tasks, either per worker or per codelet). The performance monitoring framework therefore supplies such missing routines, for the internal use of StarPU.

3.4.3.4 Runtime checks

The performance monitoring framework features a comprehensive set of runtime checks to verify that both StarPU and some external tool do not access a performance counter with the wrong typed routines, to quickly detect situations of mismatch that can result from the evolution of multiple pieces of software at distinct paces. Moreover, no StarPU data structure is accessed directly, either by the external code making use of the performance monitoring framework. The use of the C enum constants is optional; referring to values through constant strings is available when more robustness is desired. These runtime checks enable the framework to be extensible. Moreover, while the framework's counters currently are permanently compiled in, they could be made optional at compile time, for instance to suppress any overhead once the analysis and optimization process has been completed by the programmer. Thanks to the runtime discovery of available counters, the applicative code, or an intermediate layer such as skeleton layer acting on its behalf, would then be able to adapt to performance analysis builds versus optimized builds.

3.4.4 Exported Counters

3.4.4.1 Global Scope

Counter Name	Counter Definition
`starpu.task.g_total_submitted`	Total number of tasks submitted
`starpu.task.g_peak_submitted`	Maximum number of tasks submitted, waiting for dependencies resolution at any time
`starpu.task.g_peak_ready`	Maximum number of tasks ready for execution, waiting for an execution slot at any time

3.4.4.2 Per-worker Scope

Counter Name	Counter Definition
`starpu.task.w_total_executed`	Total number of tasks executed on a given worker
`starpu.task.w_cumul_execution_time`	Cumulated execution time of tasks executed on a given worker

3.4.4.3 Per-Codelet Scope

Counter Name	Counter Definition
`starpu.task.c_total_submitted`	Total number of submitted tasks for a given codelet
`starpu.task.c_peak_submitted`	Maximum number of submitted tasks for a given codelet waiting for dependencies resolution at any time
`starpu.task.c_peak_ready`	Maximum number of ready tasks for a given codelet waiting for an execution slot at any time
`starpu.task.c_total_executed`	Total number of executed tasks for a given codelet
`starpu.task.c_cumul_execution_time`	Cumulated execution time of tasks for a given codelet

3.4.5 Sequence of operations

This section presents a typical sequence of operations to interface an external tool with some StarPU performance counters. In this example, the counters monitored are the per-worker total number of executed tasks (`starpu.task.w_total_executed`) and the tasks' cumulated execution time (`starpu.task.w_cumul_execution_time`).

Step 0: Initialize StarPU

StarPU must first be initialized, by a call to starpu_init(), for performance counters to become available, since each module of StarPU registers the performance counters it exports during that initialization phase.

```
int ret = starpu_init(NULL);
```

Step 1: Allocate a counter set

A counter set has to be allocated on the per-worker scope. The per-worker scope id can be obtained by name, or with the pre-defined enum value starpu_perf_counter_scope_per_worker.

```
enum starpu_perf_counter_scope w_scope = starpu_perf_counter_scope_per_worker;
struct starpu_perf_counter_set *w_set = starpu_perf_counter_set_alloc(w_scope);
```

Step 2: Get the counter IDs Each performance counter has a unique ID used to refer to it in subsequent calls to the performance monitoring framework.

```
int id_w_total_executed = starpu_perf_counter_name_to_id(w_scope,
                                                         "starpu.task.w_total_executed");
int id_w_cumul_execution_time = starpu_perf_counter_name_to_id(w_scope,
                                                       "starpu.task.w_cumul_execution_time");
```

Step 3: Enable the counters in the counter set

This step indicates which counters will be collected into performance monitoring samples for the listeners referring to this counter set.

```
starpu_perf_counter_set_enable_id(w_set, id_w_total_executed);
starpu_perf_counter_set_enable_id(w_set, id_w_cumul_execution_time);
```

Step 4: Write a listener callback

This callback will be triggered when a sample becomes available. Upon execution, it reads the values for the two counters from the sample and displays these values, for the sake of the example.

```
void w_listener_cb(struct starpu_perf_counter_listener *listener,
                   struct starpu_perf_counter_sample   *sample,
                   void *context)
{
  int32_t w_total_executed =
             starpu_perf_counter_sample_get_int32_value(sample, id_w_total_executed);
  double w_cumul_execution_time =
      starpu_perf_counter_sample_get_double_value(sample, id_w_cumul_execution_time);
  printf("worker[%d]:  w_total_executed = %d, w_cumul_execution_time = %lf\n",
                                starpu_worker_get_id(),
                                w_total_executed,
                                w_cumul_execution_time);
}
```

Step 5: Initialize the listener

This step allocates the listener structure and prepares it to listen to the selected set of per-worker counters. However, it is not actually active until Step 6, once it is attached to one or more worker.

```
struct starpu_perf_counter_listener * w_listener =
                      starpu_perf_counter_listener_init(w_set, w_listener_cb, NULL);
```

Step 6: Set the listener on all workers This step actually makes the listener active, in this case on every StarPU worker thread.

```
starpu_perf_counter_set_all_per_worker_listeners(w_listener);
```

After this step, any task assigned to a worker will be counted in that worker selected performance counters, and reported to the listener.

3.5 Performance Steering Knobs

This section presents the StarPU performance steering framework. It summarizes the objectives of the framework. It introduces the entities involved in the framework, and then details the API, implementation and sequence of operations.

3.5.1 Objectives

The objectives of this framework are to let external tools interface with StarPU, observe, and act at runtime on actionable performance steering knobs exported by StarPU, in a generic, safe, extensible way. It defines an API to let such external tools discover the available performance steering knobs in a particular StarPU revision of build, as well as the type of each knob.

3.5.2 Entities

3.5.2.1 Performance Steering Knob

The performance steering knob entity designates one runtime-actionable knob exported by StarPU. It may represent some setting, or some constant used within StarPU for a given purpose. The value of the knob is typed, it can be

obtained or modified with the appropriate getter/setter routine. The knob belongs to a scope. A performance steering knob is designated with a unique name and unique ID number.

3.5.2.2 Knob Type

A performance steering knob has a type. A type is designated by a unique name and unique ID number. Currently, supported types include:

Type Name	Type Definition
"int32"	32-bit signed integers
"int64"	64-bit signed integers
"float"	32-bit single precision floating point
"double"	64-bit double precision floating point

On/Off knobs are defined as "int32" type, with value 0 for Off and value !0 for On, unless otherwise specified.

3.5.2.3 Knob Scope

A performance steering knob belongs to a scope. The scope of a knob defines the context considered for computing the corresponding knob. A scope is designated with a unique name and unique ID number. Currently, defined scopes include:

Scope Name	Scope Definition
"global"	Knob is global to the StarPU instance
"per_worker"	Knob is within the scope of a thread worker
"per_scheduler"	Knob is within the scope of a scheduling policy instance

3.5.2.4 Knob Group

The notion of Performance Steering Knob Group is currently internal to StarPU. It defines a series of knobs that are handled by the same couple of setter/getter functions internally. A knob group belongs to a knob scope.

3.5.3 Application Programming Interface

The API is defined in the starpu_perf_steering.h public header file of StarPU. This header file is automatically included with starpu.h.

3.5.4 Implementation Details

While the APIs of the monitoring and the steering frameworks share a similar design philosophy, the internals are significantly different. Since the effect of the steering knobs varies widely, there is no global locking scheme in place shared for all knobs. Instead, each knob gets its own procedures to get the value of a setting, or change it. To prevent code duplication, some related knobs may share getter/setter routines as knob groups.
The steering framework does not involve callback routines. Knob get operations proceed immediately, except for the possible delay in getting access to the knob value. Knob set operations also proceed immediately, not counting the exclusive access time, though their action result may be observed with some latency, depending on the knob and on the current workload. For instance, acting on a per-worker `starpu.worker.w_enable_worker_knob` to disable a worker thread may be observed only after the corresponding worker's assigned task queue becomes empty, since its actual effect is to prevent additional tasks to be queued to the worker, and not to migrate already queued tasks to another worker. Such design choices aim at providing a compromise between offering some steering capabilities and keeping the cost of supporting such steering capabilities to an acceptable level.
The framework is designed to be easily extensible. At StarPU initialization time, the framework calls initialization functions if StarPU modules to initialize the set of knobs they export. Knob get/set accessors can be shared among multiple knobs in a knob group. Thus, exporting a new knob is basically a matter of declaring it at initialization time, by specifying its name and value type, and either add its handling to an existing getter/setter pair of accessors in a knob group, or create a new group. As the performance monitoring framework, the performance steering

framework is currently permanently enabled, but could be made optional at compile-time to separate testing builds from production builds.

3.5.5 Exported Steering Knobs

3.5.5.1 Global Scope

Knob Name	Knob Definition
starpu.global.g_calibrate_knob	Enable/disable the calibration of performance models
starpu.global.g_enable_catch_↩ signal_knob	Enable/disable the catching of UNIX signals

3.5.5.2 Per-worker Scope

Knob Name	Knob Definition
starpu.worker.w_bind_to_pu_knob	Change the processing unit to which a worker thread is bound
starpu.worker.w_enable_worker_knob	Disable/re-enable a worker thread to be selected for task execution

3.5.5.3 Per-Scheduler Scope

Knob Name	Knob Definition
starpu.task.s_max_priority_cap_knob	Set a capping maximum priority value for subsequently submitted tasks
starpu.task.s_min_priority_cap_knob	Set a capping minimum priority value for subsequently submitted tasks
starpu.dmda.s_alpha_knob	Scaling factor for the Alpha constant for Deque Model schedulers to alter the weight of the estimated task execution time
starpu.dmda.s_beta_knob	Scaling factor for the Beta constant for Deque Model schedulers to alter the weight of the estimated data transfer time for the task's input(s)
starpu.dmda.s_gamma_knob	Scaling factor for the Gamma constant for Deque Model schedulers to alter the weight of the estimated power consumption of the task
starpu.dmda.s_idle_power_knob	Scaling factor for the baseline Idle power consumption estimation of the corresponding processing unit

3.5.6 Sequence of operations

This section presents an example of a sequence of operations representing a typical use of the performance steering knobs exported by StarPU. In this example, a worker thread is temporarily barred from executing tasks. For that, the corresponding starpu.worker.w_enable_worker_knob of the worker, initially set to 1 (= enabled) is changed to 0 (= disabled).

Step 0: Initialize StarPU

StarPU must first be initialized, by a call to starpu_init(). Performance steering knobs only become available after this step, since each module of StarPU registers the knobs it exports during that initialization phase.

```
int ret = starpu_init(NULL);
```

Step 1: Get the knob ID

Each performance steering knob has a unique ID used to refer to it in subsequent calls to the performance steering framework. The knob belongs to the "per_worker" scope.

```
int w_scope = starpu_perf_knob_scope_name_to_id("per_worker");
int w_enable_id = starpu_perf_knob_name_to_id(w_scope, "starpu.worker.w_enable_worker_knob");
```

Step 2: Get the knob current value

This knob is an On/Off knob. Its value type is therefore a 32-bit integer, with value 0 for Off and value !0 for On. The

getter functions for per-worker knobs expect the knob ID as first argument, and the worker ID as second argument. Here the getter call obtains the value of worker 5.

```
int32_t val = starpu_perf_knob_get_per_worker_int32_value(w_enable_id, 5);
```

Step 3: Set the knob current value

The setter functions for per-worker knobs expect the knob ID as first argument, the worker ID as second argument, and the new value as third argument. Here, the value for worker 5 is set to 0 to temporarily bar the worker thread from accepting new tasks for execution.

```
starpu_perf_knob_set_per_worker_int32_value(w_enable_id, 5, 0);
```

Subsequently, setting the value of the knob back to 1 enables the corresponding to accept new tasks for execution again.

```
starpu_perf_knob_set_per_worker_int32_value(w_enable_id, 5, 1);
```

Chapter 4

Offline Performance Tools

To get an idea of what is happening, a lot of performance feedback is available, detailed in this chapter. The various information should be checked for.

- What does the Gantt diagram look like? (see Creating a Gantt Diagram)
  - If it's mostly green (tasks running in the initial context) or context specific color prevailing, then the machine is properly utilized, and perhaps the codelets are just slow. Check their performance, see Performance Of Codelets.
  - If it's mostly purple (FetchingInput), tasks keep waiting for data transfers, do you perhaps have far more communication than computation? Did you properly use CUDA streams to make sure communication can be overlapped? Did you use data-locality aware schedulers to avoid transfers as much as possible?
  - If it's mostly red (Blocked), tasks keep waiting for dependencies, do you have enough parallelism? It might be a good idea to check what the DAG looks like (see Creating a DAG With Graphviz).
  - If only some workers are completely red (Blocked), for some reason the scheduler didn't assign tasks to them. Perhaps the performance model is bogus, check it (see Performance Of Codelets). Do all your codelets have a performance model? When some of them don't, the schedulers switches to a greedy algorithm which thus performs badly.

You can also use the Temanejo task debugger (see UsingTheTemanejoTaskDebugger) to visualize the task graph more easily.

4.1 Generating Traces With FxT

StarPU can use the FxT library (see `https://savannah.nongnu.org/projects/fkt/`) to generate traces with a limited runtime overhead.
You can get a tarball from `http://download.savannah.gnu.org/releases/fkt/?C=M`
Compiling and installing the FxT library in the `$FXTDIR` path is done following the standard procedure:

```
$ ./configure --prefix=$FXTDIR
$ make
$ make install
```

In order to have StarPU to generate traces, StarPU needs to be configured again after installing FxT, and configuration show:

```
FxT trace enabled: yes
```

If `configure` does not find FxT automatically, it can be specified by hand with the option --with-fxt :

```
$ ./configure --with-fxt=$FXTDIR
```

Or you can simply point the `PKG_CONFIG_PATH` environment variable to `$FXTDIR/lib/pkgconfig`
When STARPU_FXT_TRACE is set to 1, a trace is generated when StarPU is terminated by calling starpu_shutdown(). The trace is a binary file whose name has the form `prof_file_XXX_YYY` where `XXX` is the username, and `YYY` is the MPI id of the process that used StarPU (or 0 when running a sequential program).

One can change the name of the file by setting the environment variable STARPU_FXT_SUFFIX, its contents will be used instead of `prof_file_XXX`. This file is saved in the `/tmp/` directory by default, or by the directory specified by the environment variable STARPU_FXT_PREFIX.
The additional `configure` option --enable-fxt-lock can be used to generate trace events which describes the lock's behavior during the execution. It is however very heavy and should not be used unless debugging StarPU's internal locking.
When the FxT trace file `prof_file_something` has been generated, it is possible to generate different trace formats by calling:

```
$ starpu_fxt_tool -i /tmp/prof_file_something
```

Or alternatively, setting the environment variable STARPU_GENERATE_TRACE to `1` before application execution will make StarPU automatically generate all traces at application shutdown. Note that if the environment variable STARPU_FXT_PREFIX is set, files will be generated in the given directory.
One can also set the environment variable STARPU_GENERATE_TRACE_OPTIONS to specify options, see `starpu_fxt_tool -help`, for example:

```
$ export STARPU_GENERATE_TRACE=1
$ export STARPU_GENERATE_TRACE_OPTIONS="-no-acquire"
```

When running an MPI application, STARPU_GENERATE_TRACE will not work as expected (each node will try to generate trace files, thus mixing outputs...), you have to collect the trace files from the MPI nodes, and specify them all on the command `starpu_fxt_tool`, for instance:

```
$ starpu_fxt_tool -i /tmp/prof_file_something*
```

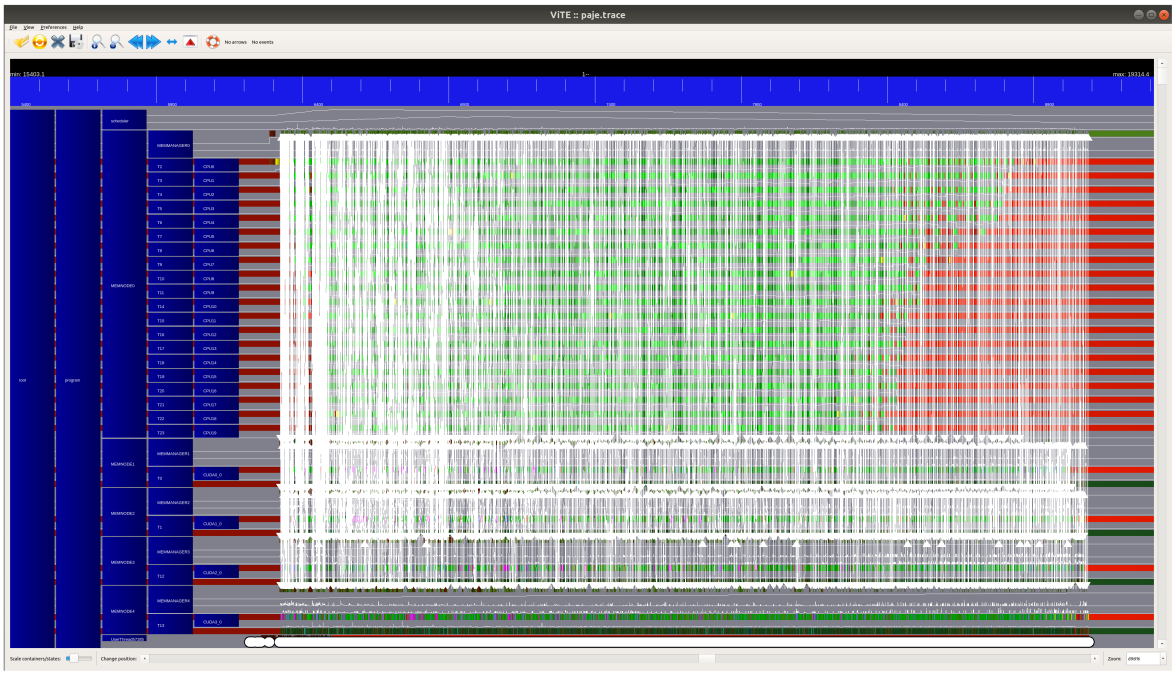
By default, the generated trace contains all information. To reduce the trace size, various `-no-foo` options can be passed to `starpu_fxt_tool`, see `starpu_fxt_tool -help`.

4.1.1 Creating a Gantt Diagram

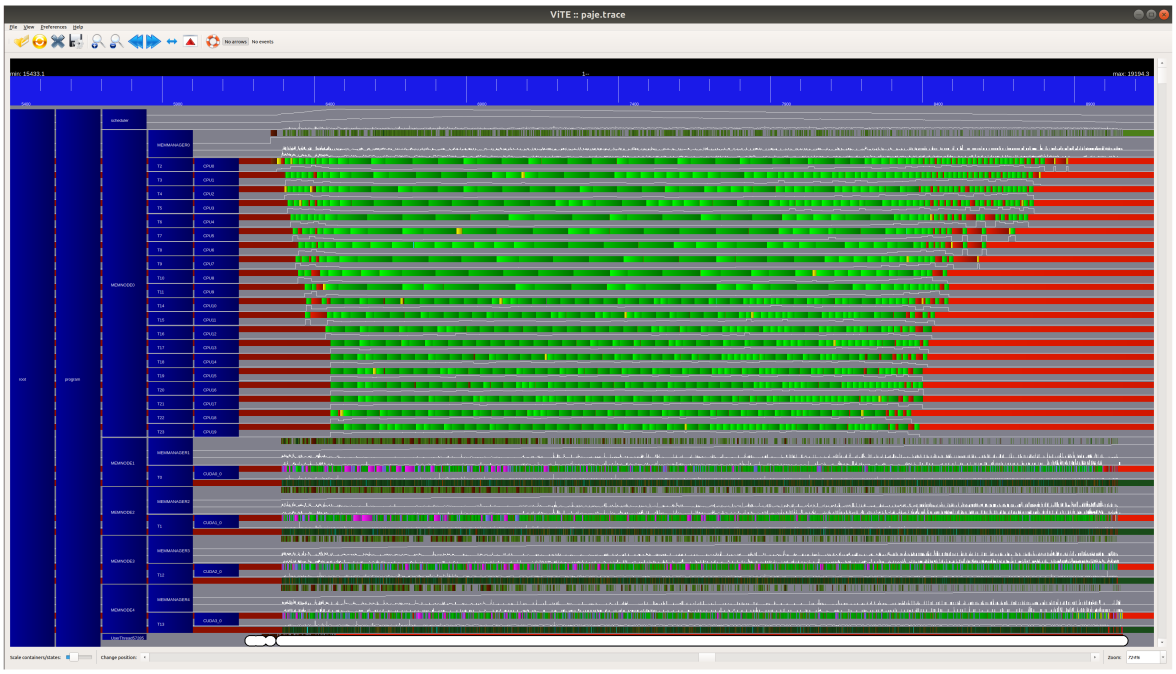
One of the generated files is a trace in the Paje format. The file, located in the current directory, is named `paje.trace`. It can be viewed with ViTE (https://solverstack.gitlabpages.inria.fr/vite/) a trace visualizing open-source tool. To open the file `paje.trace` with ViTE, use the following command:

```
$ vite paje.trace
```

Once the file is opened in ViTE interface, we will see the figure as shown below:



We can then click the "No arrows" button in task bar of ViTE interface, to better observe the Gantt diagram that illustrates the start and end dates of the different tasks or activities of a program.

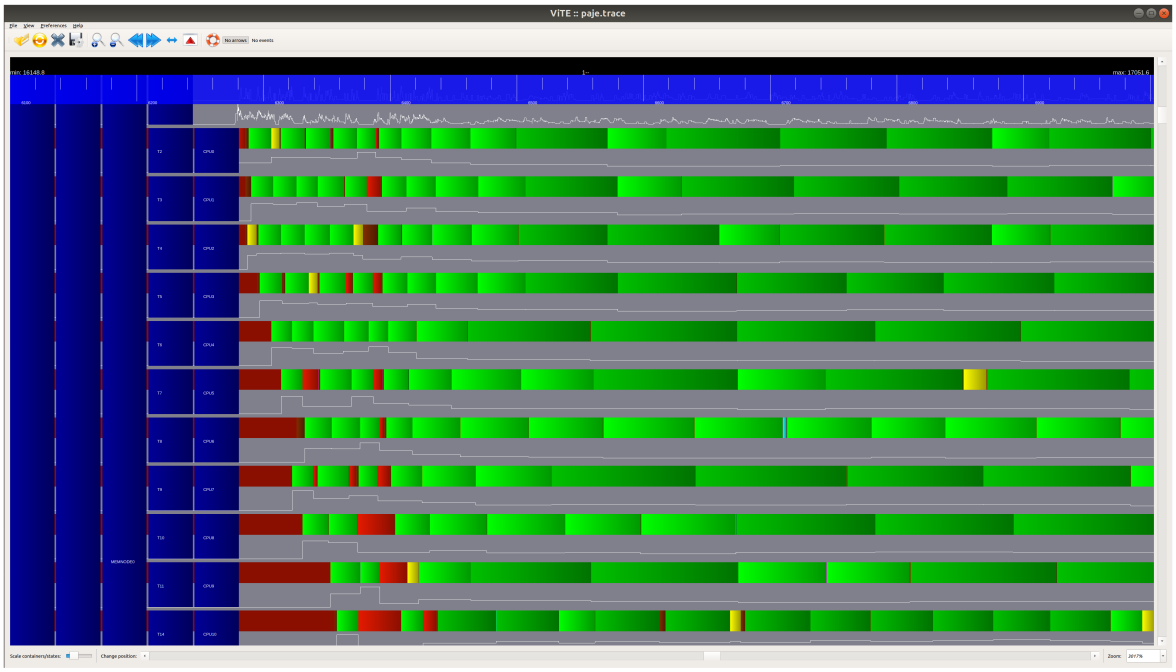


In the Gantt diagram, the bar types such as devices (CPU or GPU) are displayed on the left side. Each task is represented by a horizontal rectangle that spans the duration of the task. The rectangles are arranged along a timeline axis, which is shown at the top of the Gantt diagram and represents the overall duration of the program in milliseconds. The position of the bar along the timeline shows when the task begins and ends. We can see some long red bars at the beginning and end of the entire timeline, which represent that the unit is idle. There are no tasks at these moments, and workers are waiting or in a sleeping state.

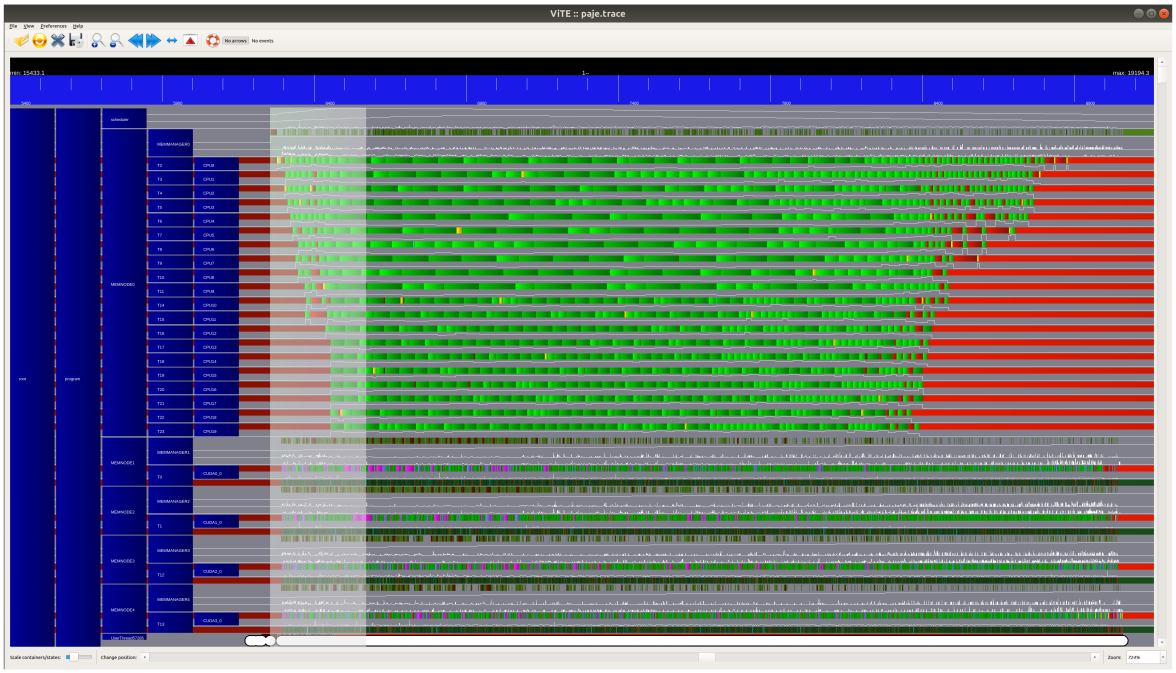
4.1.1.1 Zooming in Gantt Diagram

Then as shown in the following figure, press and hold the left mouse button to select the area you want to zoom in on. Release the button to view the selected area, and we can repeat the zoom action multiple times.

This zoom result is:



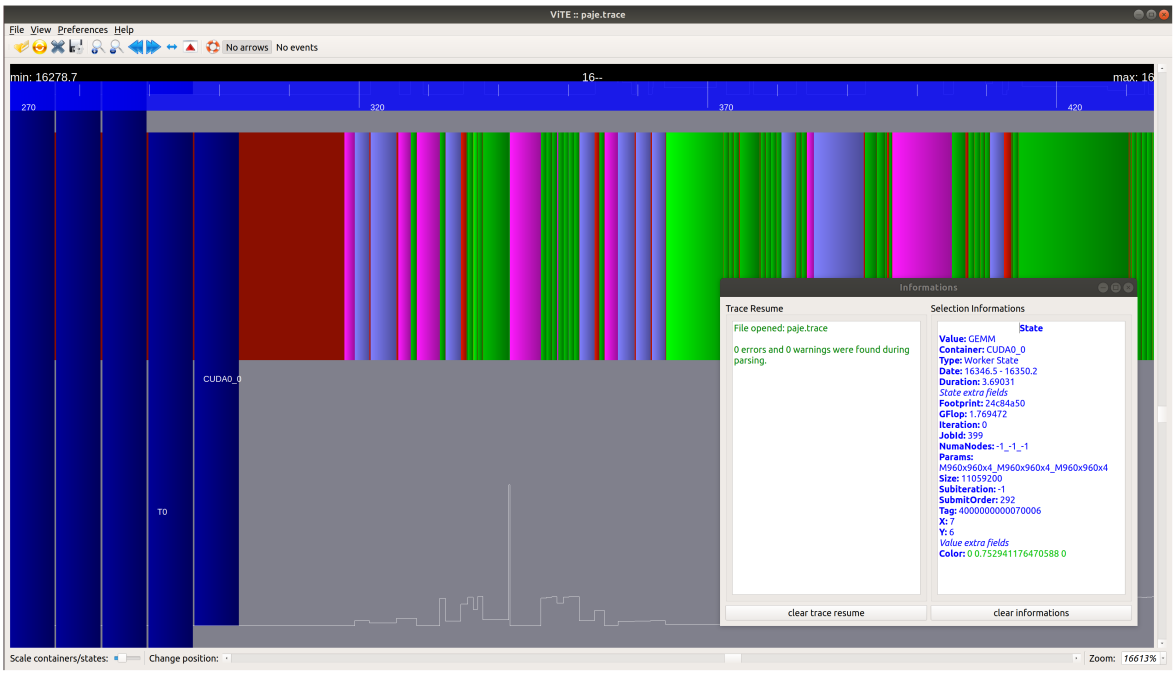
Right-clicking anywhere on the Gantt diagram restores the previous zoom view.
One can press and hold the left mouse button inside the top blue bar to select horizontally, which will horizontally zoom in on all Gantt diagrams within the selected time range.



This zoom result is:

4.1.1.2 Colors in Gantt Diagram

After zooming in, we can observe numerous blocks of varying colors, each block representing a task. Blocks of diverse colors signify different types of tasks. When we double-click on any block, a pop-up window will show related status about that task, such as its type and which worker (CPU/GPU) it belongs to, etc.



The state information displayed in the pop-up window can be:

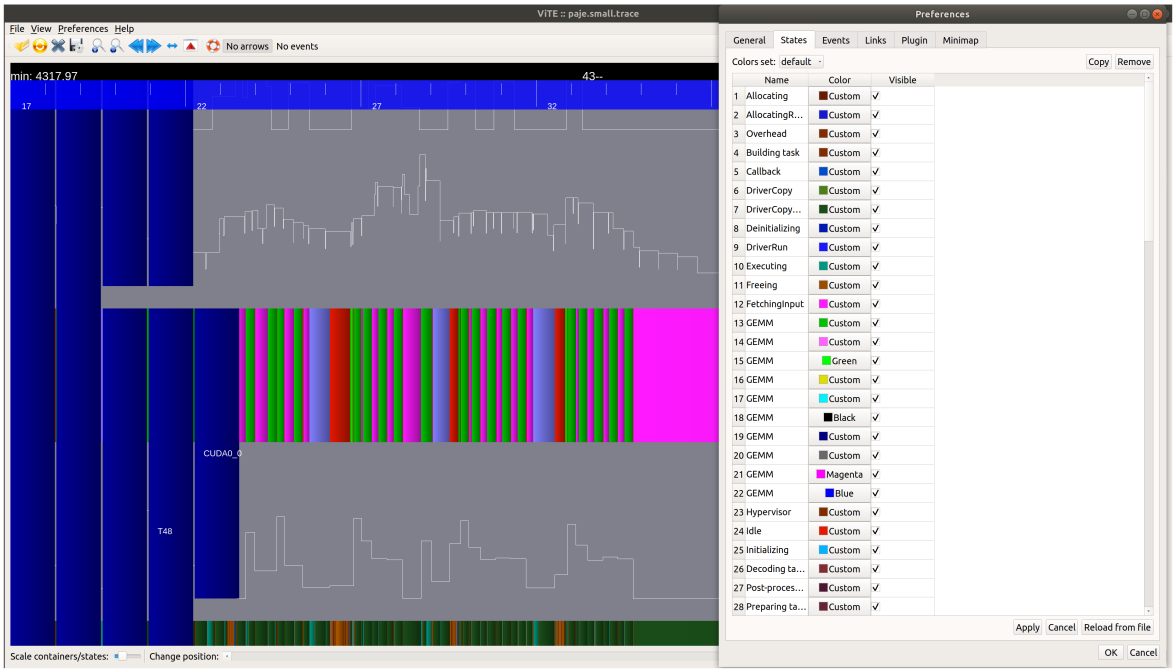
- **Value**: refers to a type of task, which can be assigned as a task name (instead of the default `unknown`) by filling the optional starpu_codelet::name, or assigning it a performance model. The name can also be set with the field starpu_task::name or by using STARPU_NAME when calling starpu_task_insert()
- **Container**: refers to a specific worker where the computation was performed, could be CPU or CUDA
- **Type**: indicates the type of this block, most often "Worker State"

- **Date**: represents a range of dates during which the computation was performed
- **Duration**: represents the duration of the computation
- **Footprint**: provides the data footprint of the task (used as indexing base for performance models)
- **GFlop**: represents the number of Gflop performed during the computation, as set in starpu_task::flops.
- **Iteration**: refers to the iteration number of the computation, as set by starpu_iteration_push() at the beginning of submission loops and starpu_iteration_pop() at the end of submission loops
- **JobId**: represents a unique identifier for the specific task, as returned by starpu_task_get_job_id()
- **NumaNodes**: refers to the NUMA node where the data is stored, the environment variable STARPU_FXT↩ _EVENTS needs to contain `TASK_VERBOSE_EXTRA`, otherwise it will be -1
- **Params**: represents parameters or input/output types and sizes, possibly indicating the dimensions of the matrices
- **Size**: represents the size of the data being operated on in bytes
- **Subiteration**: represents a sub-iteration number if the computation was part of a larger iteration or loop, as set by starpu_iteration_push()
- **SubmitOrder**: represents the order in which the task was submitted by the application
- **Tag**: represents a unique identifier for the task, which can be set either through starpu_task::tag_id or by using STARPU_TAG or STARPU_TAG_ONLY when calling starpu_task_insert()
- **X**: represents an X-coordinate index of the first data written by the task, which was set by starpu_data_set_coordinates() or starpu_data_set_coordinates_array() function. We can also get the coordinates of the data with starpu_data_get_coordinates_array() function
- **Y**: represents an Y-coordinate index of the first data written by the task, which was set by starpu_data_set_coordinates() or starpu_data_set_coordinates_array() function. We can also get the coordinates of the data with starpu_data_get_coordinates_array() function
- **Color**: represents the color RGB value associated with the task. Tasks are by default shown in green. To use a different color for every type of task, we can specify the option `-c` to `starpu_fxt_`↩ `tool` or in STARPU_GENERATE_TRACE_OPTIONS. Tasks can also be given a specific color by setting the field starpu_codelet::color or the starpu_task::color. When we call starpu_task_insert(), we can use STARPU_TASK_COLOR to set the color. Colors are expressed with the following format `0xRRGGBB` (e.g. `0xFF0000` for red). See `basic_examples/task_insert_color` for examples on how to assign colors

In the shown figure, the set of color as following:

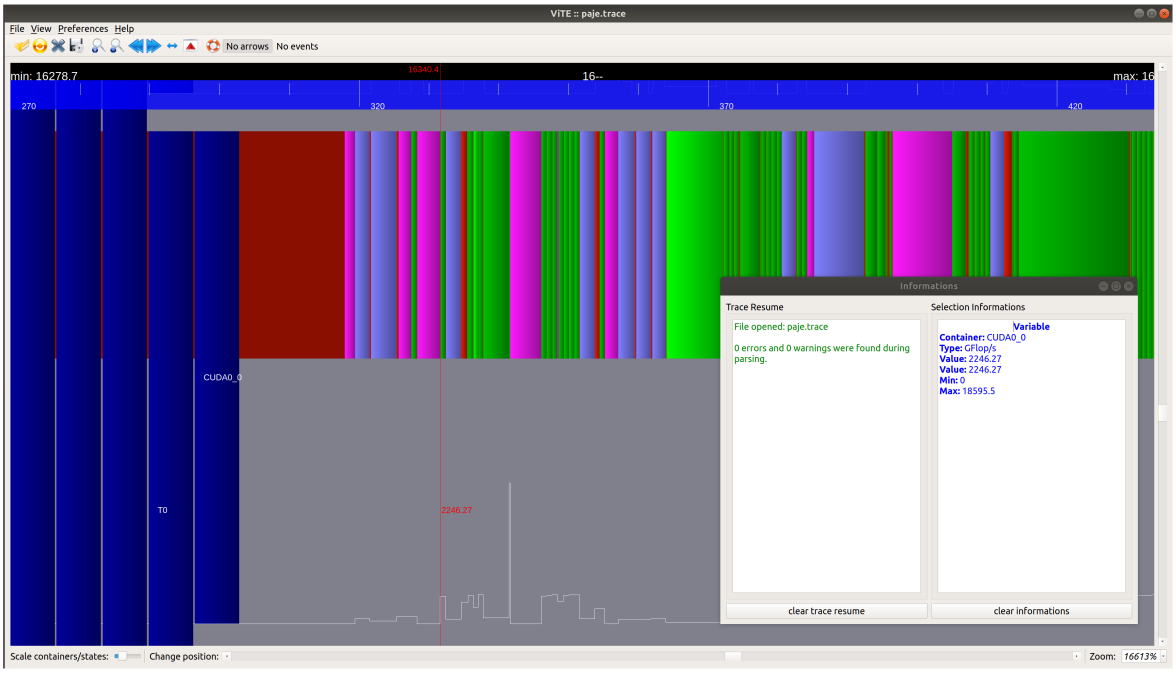
- Dark green represents GEMM
- Light green represents SYRK
- Blue represents TRSM
- Red indicates that the unit is idle, there are no tasks at the moment, it is currently waiting or in a sleeping state
- Magenta represents FetchingInput

To modify the colors in Vite interface, select "Preferences" then "Settings" in the options bar, and then choose the "States" tab in the newly opened window to select different colors for different operations, as shown in the figure below. One has to click the reload button at the top left to reload the trace with the new colors.



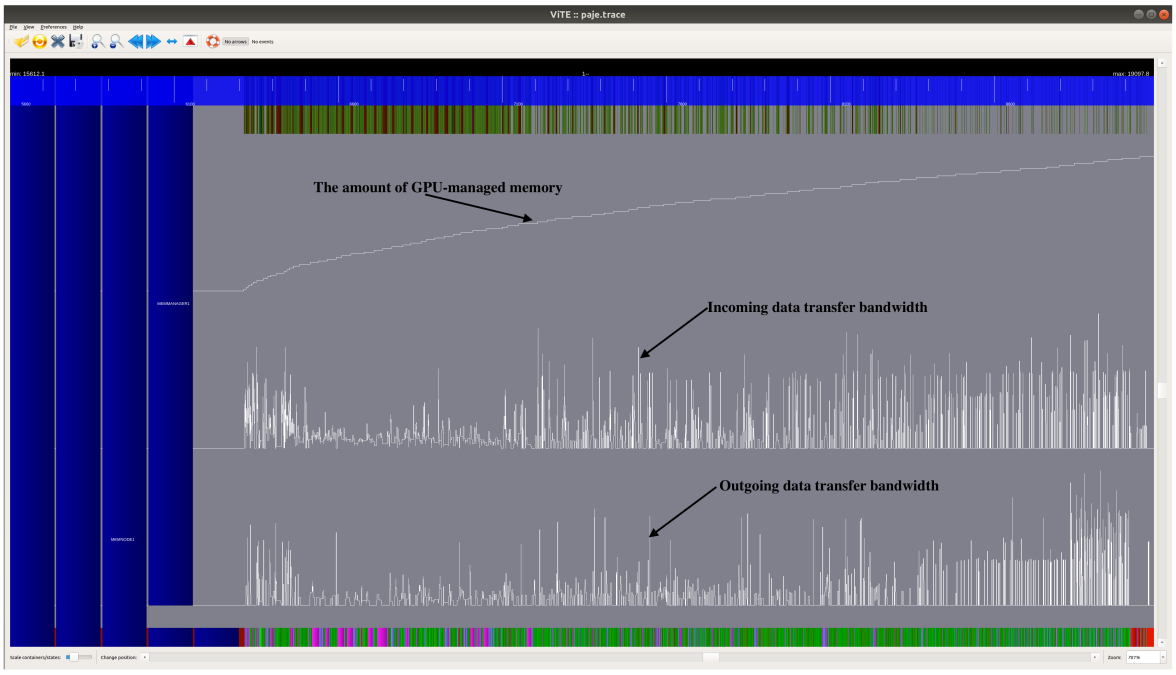
4.1.1.3 Curves in Gantt Diagram

We can see that there is a curve below task blocks, which represents the corresponding `GFlop/s`. Double-clicking near the curve will display the current `GFlop/s` information in a pop-up window (as shown in the figure). If we only click on the curve, a vertical red line shows up, and we can read on it the `GFlop/s` values of all the curves at the same time.



For GPUs, there are three additional curves above the task blocks that can be double-clicked to open a pop-up window to view information. Let's zoom in on the three curves during the entire execution process as illustrated in the figure:

As shown in the figure below, the top curve represents the amount of GPU-managed memory in MBytes, while the bottom two curves represent the data transfer between tasks on the CPU and GPU, and between tasks on different GPUs. They respectively indicate the incoming and outgoing data transfer bandwidth. By looking at the memory curve, we can observe that the memory usage kept increasing at first, but due to the reutilization of the allocations by StarPU, the curve gradually became stable later on.

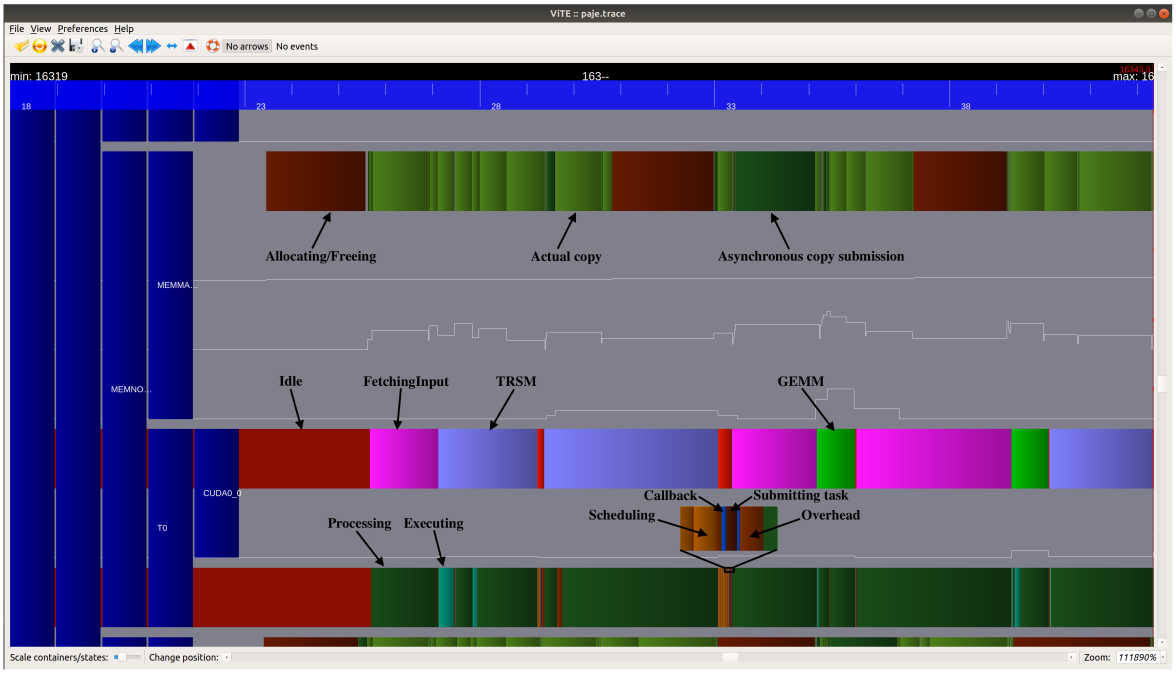


4.1.1.4 States in Gantt Diagram

Above these three curves, we can see some blocks which represent driver copy (see the top of the figure below), i.e. a memory copy. The light green blocks represent the actual copies, the dark green blocks represent asynchronous copy submissions, and the burgundy blocks represent allocating and freeing. Double-clicking on a block allows us to view relevant information in the pop-up window.
Here, a couple of issues may show up:

- If the "Allocating/Freeing" parts take a long time, it means that StarPU does not manage to re-use data buffers allocated in the GPU. If you have e.g. a lot of tiles with different sizes, it may be useful to approximate the allocation size, by using e.g. starpu_matrix_data_register_allocsize() with the proper nx / ld / ny, but an allocation size that is rounded up, so that buffers with that same rounded size can be shared.

- If the "Asynchronous copy submission" parts take a long time, it means that the CPU buffers are not pinned: you need to make sure to use starpu_malloc(), or starpu_memory_pin() (see CUDA-specificOptimizations) so that the CPU buffers are pinned so that the GPU driver can efficiently process transfers asynchronously (in the "Actual copy" part) rather than synchronously (in the "Asynchronous copy submission" part).



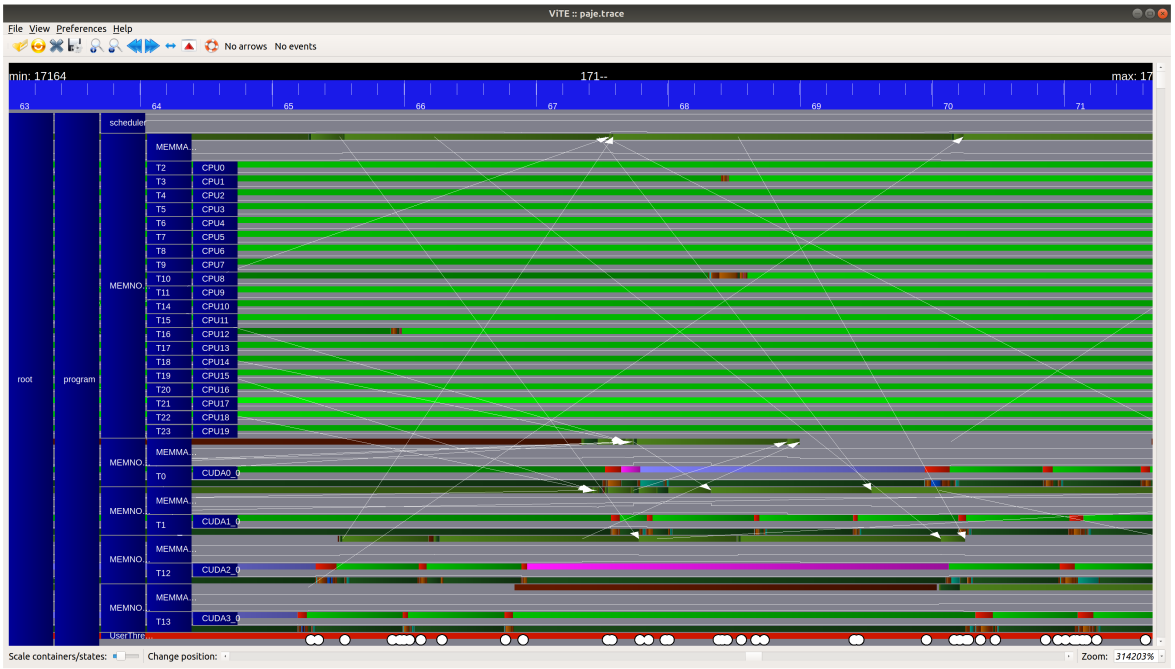
Below the GPU task blocks and `GFlops` curve (see the bottom of the figure above), we can see some other blocks that represent the CPU waiting for the GPU to complete the task. During time, CPU can do variable actions which are represented by blocks of different colors, such as:

- Dark green represents `progressing`, it keeps polling for task or data transfer completion
- Brown-yellow represents `scheduling`
- Burgundy represents `submitting task`
- Lake blue represents `executing`, it is executing the application codelet function. Here it is very short because the codelet just submits a kernel asynchronously.
- Dark blue represents `callback`
- Chestnut represents `overhead`. This state is not supposed to be long, as it represents everything that we did not classify as an operation that is supposed to be long like the operations mentioned above. If you find situations where some overhead is long, this is a bug worth reporting so we can fix it.

and we can always double-click on the block to view relevant information in the pop-up window.

4.1.1.5 Transfers in Gantt Diagram

We can horizontally zoom in on a section of the Gantt diagram, and deselect the "No arrows" option. This will allow us to see a complete process of data transfer, as shown in the following figure:

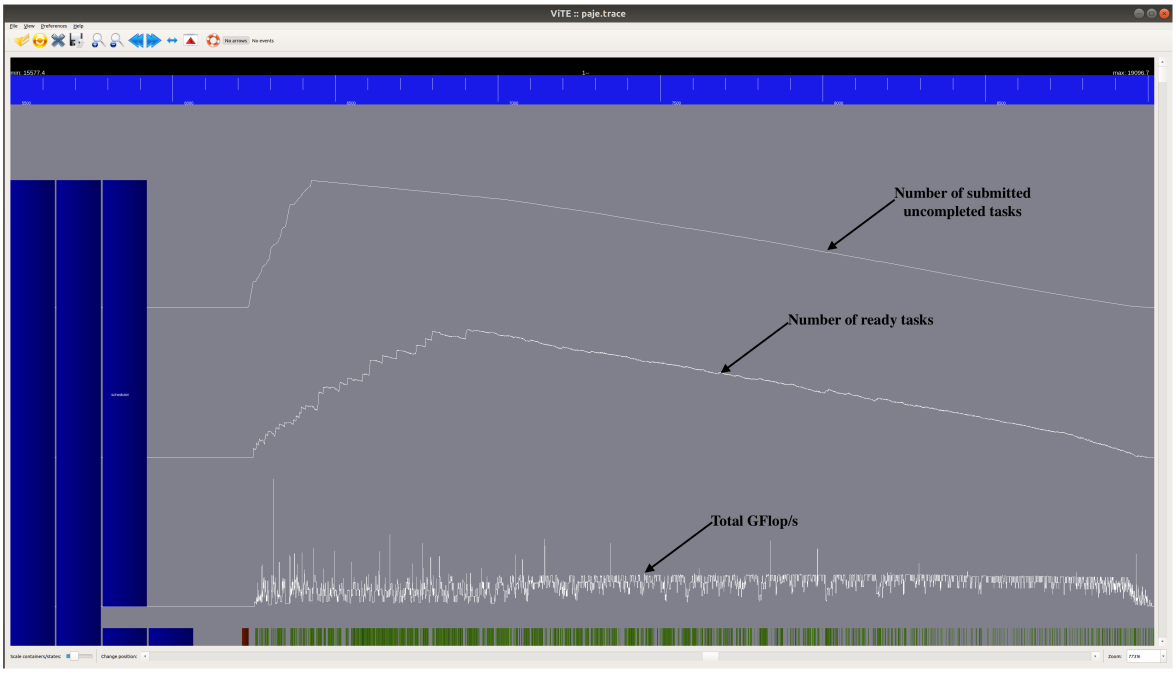


In the above figure, we can see a long segment of magenta color in CUDA2_0 task blocks. At the same time, we can see that there are numerous transfers between other workers during this time period. This indicates that CUDA2_0 is waiting for the completion of the data transfers needed by the task it wants to execute.

4.1.1.6 Scheduler in Gantt Diagram

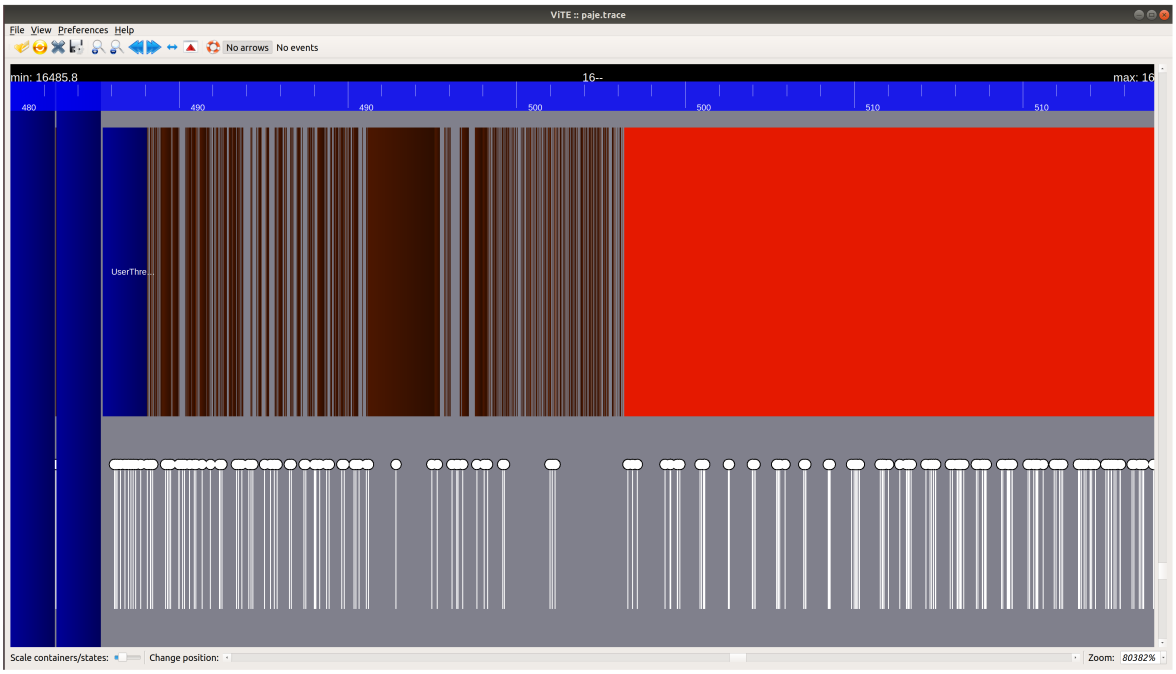
At the top of the entire Gantt diagram, there are three curves that represent the information of the scheduler. Let's zoom in on the three curves during the entire execution process as illustrated in the figure below:

As shown in the figure below, from top to bottom, they respectively indicate the number of submitted uncompleted tasks, the number of ready tasks, and the total `GFlop/s` for this moment. By double-clicking on the curves, we can view relevant information in the pop-up window.



4.1.1.7 Main Thread in Gantt Diagram

At the very bottom of the entire Gantt diagram, we will see a red bar, which represents the main thread waiting for tasks. In front of the red bar (see the figure below), there are some dark red bars, which represent the main thread submitting tasks.

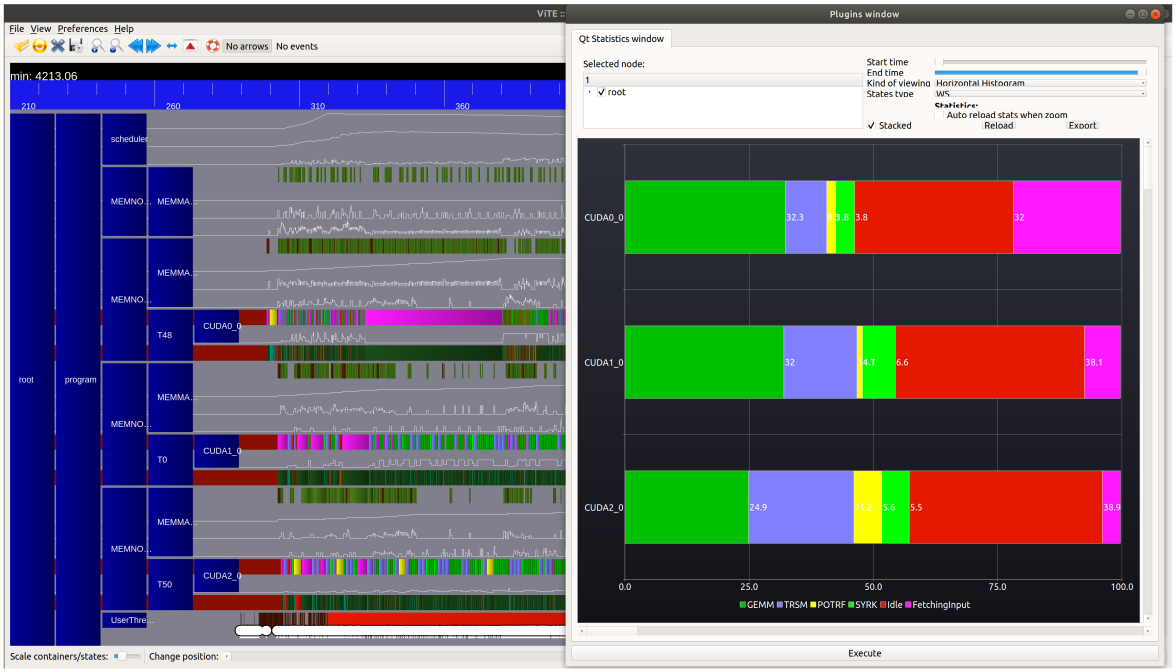


Below these red bars, we can see some white vertical lines with small circles on top, which represent events. The default events can be either task push or task pop or task wait for all. The application can inject its own events at any desired moment with the function starpu_fxt_trace_user_event() or starpu_fxt_trace_user_event_string(). Similarly, double-clicking on the white bars allows you to see relevant information in the pop-up window.

4.1.1.8 Statistics in Gantt Diagram

To get statistics on the time spent in runtime overhead, we can use the statistics plugin of ViTE. In the Preferences menu, select Plugins. In "States Type", select "Worker State". Then click on "Reload" to update the histogram. The

red "Idle" percentages are due to lack of parallelism, the "FetchingInput" percentages are due to waiting for data transfers. The brown "Overhead" and "Scheduling" percentages are due to the overhead of the runtime and of the scheduler.



4.1.2 Creating a DAG With Graphviz

Another generated trace file is a task graph described using the DOT language. The file, created in the current directory, is named `dag.dot` file in the current directory. It is possible to get a graphical output of the graph by using the `graphviz` library:

```
$ dot -Tpdf dag.dot -o output.pdf
```

4.1.3 Getting Task Details

Another generated trace file gives details on the executed tasks. The file, created in the current directory, is named `tasks.rec`. This file is in the `recutils` format, i.e. `Field: value` lines, and empty lines are used to separate each task. This can be used as a convenient input for various ad-hoc analysis tools. By default, it only contains information about the actual execution. Performance models can be obtained by running `starpu_tasks_rec_complete` on it:

```
$ starpu_tasks_rec_complete tasks.rec tasks2.rec
```

which will add `EstimatedTime` lines which contain the performance model-estimated time (in µs) for each worker starting from 0. Since it needs the performance models, it needs to be run the same way as the application execution, or at least with `STARPU_HOSTNAME` set to the hostname of the machine used for execution, to get the performance models of that machine.
Another possibility is to obtain the performance models as an auxiliary `perfmodel.rec` file, by using the `starpu_perfmodel_recdump` utility:

```
$ starpu_perfmodel_recdump tasks.rec -o perfmodel.rec
```

One can also simply call starpu_task_get_name() to get the name of a task.

4.1.4 Getting Scheduling Task Details

The file, `sched_tasks.rec`, created in the current directory, in the `recutils` format, gives information about the tasks scheduling, and lists the push and pop actions of the scheduler. For each action, it gives the timestamp, the job priority and the job id. Each action is separated from the next one by empty lines. The job id associated with the task can be retrieved by calling starpu_task_get_job_id().

4.1.5 Monitoring Activity

Another generated trace file is an activity trace. The file, created in the current directory, is named `activity.data`. A profile of the application showing the activity of StarPU during the execution of the program can be generated:

```
$ starpu_workers_activity activity.data
```

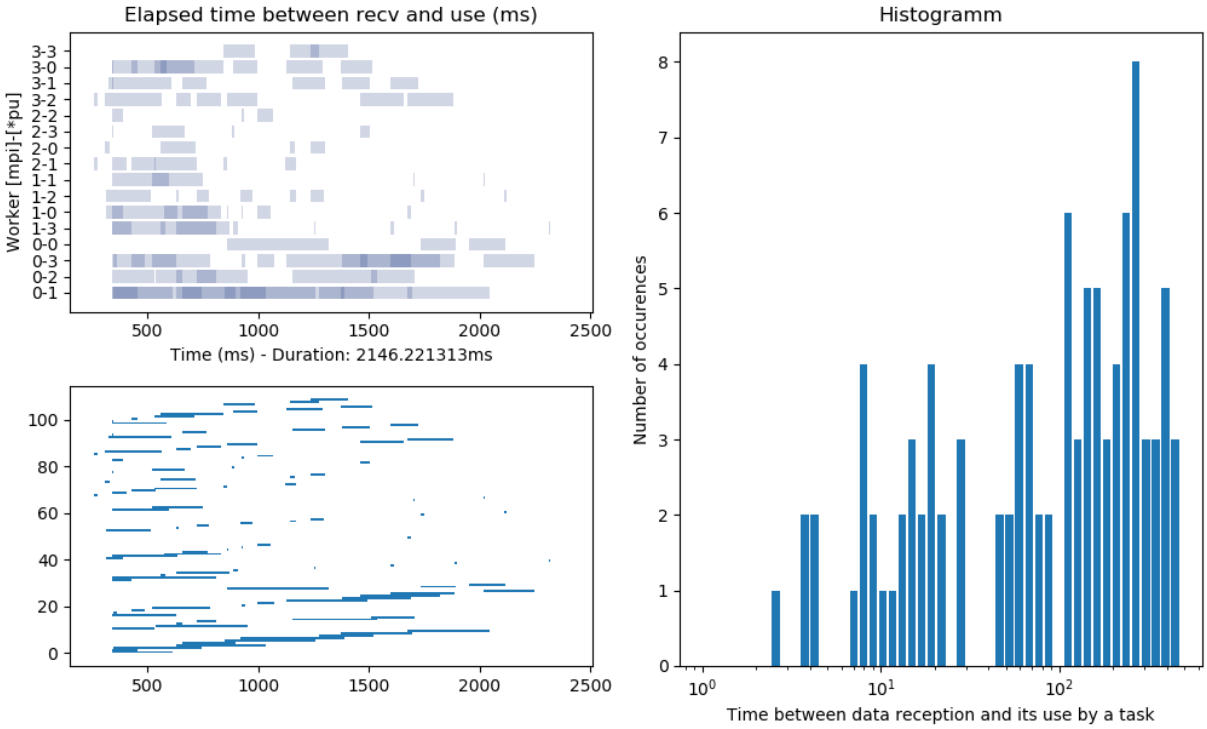
This will create a file named `activity.eps` in the current directory. This picture is composed of two parts. The first part shows the activity of the different workers. The green sections indicate which proportion of the time was spent executed kernels on the processing unit. The red sections indicate the proportion of time spent in StarPU: an important overhead may indicate that the granularity may be too low, and that bigger tasks may be appropriate to use the processing unit more efficiently. The black sections indicate that the processing unit was blocked because there was no task to process: this may indicate a lack of parallelism, which may be alleviated by creating more tasks when it is possible.
The second part of the picture `activity.eps` is a graph showing the evolution of the number of tasks available in the system during the execution. Ready tasks are shown in black, and tasks that are submitted but not schedulable yet are shown in grey.

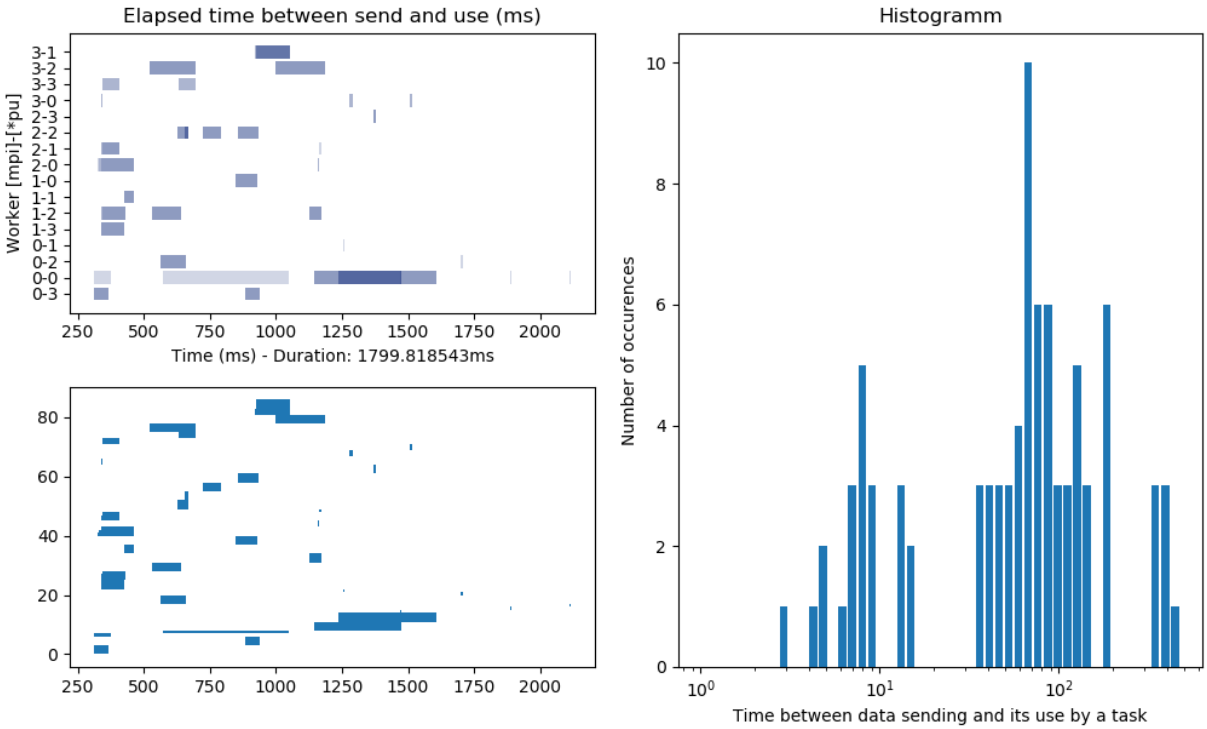
4.1.6 Getting Modular Schedular Animation

When using modular schedulers (i.e. schedulers which use a modular architecture, and whose name start with "modular-"), the call to `starpu_fxt_tool` will also produce a `trace.html` file which can be viewed in a javascript-enabled web browser. It shows the flow of tasks between the components of the modular scheduler.

4.1.7 Analyzing Time Between MPI Data Transfer and Use by Tasks

`starpu_fxt_tool` produces a file called `comms.rec` which describes all MPI communications. The script `starpu_send_recv_data_use.py` uses this file and `tasks.rec` in order to produce two graphs: the first one shows durations between the reception of data and their usage by a task and the second one plots the same graph but with elapsed time between send and usage of a data by the sender.





4.1.8 Number of events in trace files

When launched with the option `-number-events`, `starpu_fxt_tool` will produce a file named `number_events.data`. This file contains the number of events for each event type. Events are represented with their key. To convert event keys to event names, you can use the `starpu_fxt_number_events_to_names.py` script:

```
$ starpu_fxt_number_events_to_names.py number_events.data
```

The number of recorded events (and thus the performance overhead introduced by tracing) can be reduced by setting which categories of events to record with the environment variable STARPU_FXT_EVENTS.

4.1.9 Limiting The Scope Of The Trace

For computing statistics, it is useful to limit the trace to a given portion of the time of the whole execution. This can be achieved by calling

```
starpu_fxt_autostart_profiling(0)
```

before calling starpu_init(), to prevent tracing from starting immediately. Then

```
starpu_fxt_start_profiling();
```

and

```
starpu_fxt_stop_profiling();
```

can be used around the portion of code to be traced. This will show up as marks in the trace, and states of workers will only show up for that portion.

4.2 Performance Of Codelets

After calibrating performance models of codelets (see Performance Model Example and PerformanceModel↩ Calibration), they can be examined by using the tool `starpu_perfmodel_display`:

```
$ starpu_perfmodel_display -l
file: <malloc_pinned.hannibal>
file: <starpu_slu_lu_model_trsm_ru.hannibal>
file: <starpu_slu_lu_model_getrf.hannibal>
file: <starpu_slu_lu_model_gemm.hannibal>
file: <starpu_slu_lu_model_trsm_ll.hannibal>
```

Here, the codelets of the example `lu` are available. We can examine the performance of the kernel `22` (in micro-seconds), which is history-based:

```
$ starpu_perfmodel_display -s starpu_slu_lu_model_gemm
performance model for cpu
# hash      size       mean          dev           n
57618ab0    19660800   2.851069e+05  1.829369e+04  109
performance model for cuda_0
# hash      size       mean          dev           n
57618ab0    19660800   1.164144e+04  1.556094e+01  315
performance model for cuda_1
# hash      size       mean          dev           n
57618ab0    19660800   1.164271e+04  1.330628e+01  360
performance model for cuda_2
# hash      size       mean          dev           n
57618ab0    19660800   1.166730e+04  3.390395e+02  456
```

We can see that for the given size, over a sample of a few hundreds of execution, the GPUs are about 20 times faster than the CPUs (numbers are in us). The standard deviation is extremely low for the GPUs, and less than 10% for CPUs.
This tool can also be used for regression-based performance models. It will then display the regression formula, and in the case of non-linear regression, the same performance log as for history-based performance models:

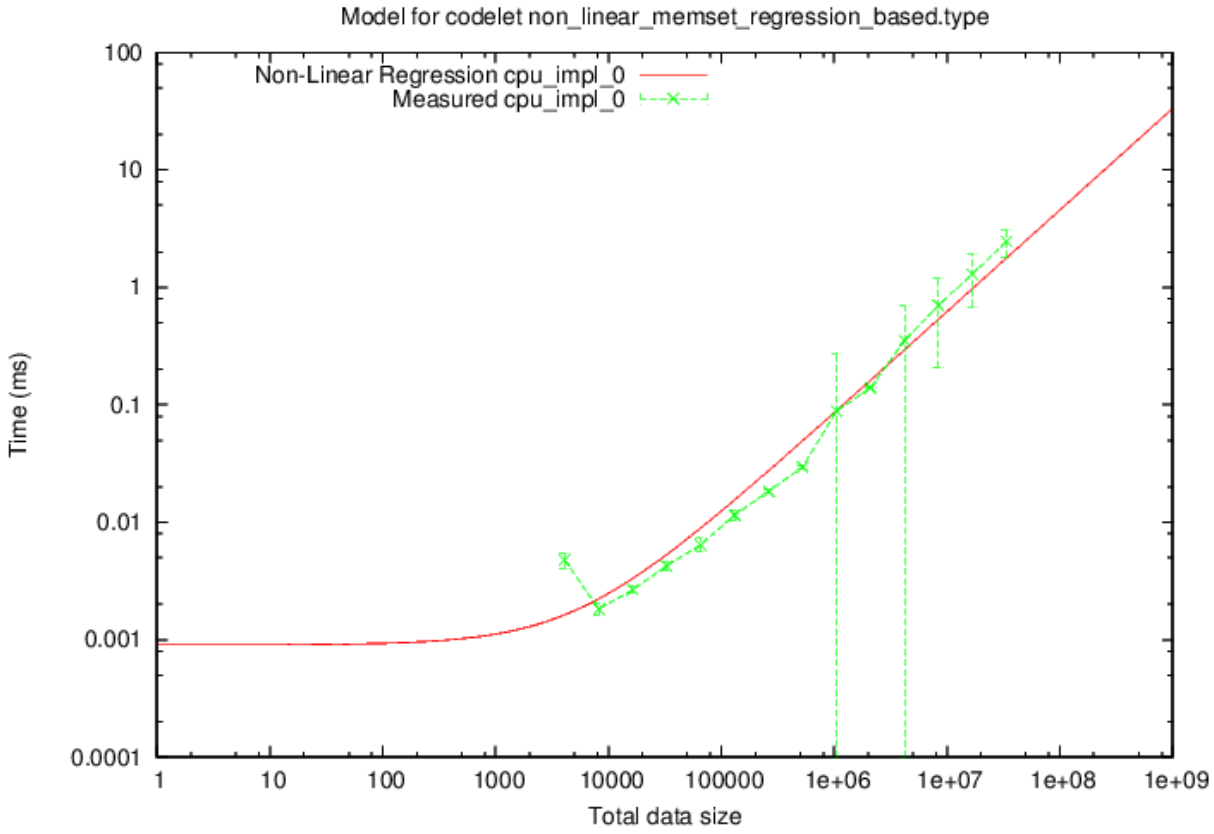
```
$ starpu_perfmodel_display -s non_linear_memset_regression_based
performance model for cpu_impl_0
        Regression : #sample = 1400
        Linear: y = alpha size ^ beta
                alpha = 1.335973e-03
                beta = 8.024020e-01
        Non-Linear: y = a size ^b + c
                a = 5.429195e-04
                b = 8.654899e-01
                c = 9.009313e-01
# hash          size            mean            stddev          n
a3d3725e        4096            4.763200e+00    7.650928e-01    100
870a30aa        8192            1.827970e+00    2.037181e-01    100
48e988e9        16384           2.652800e+00    1.876459e-01    100
961e65d2        32768           4.255530e+00    3.518025e-01    100
...
```

The same can also be achieved by using StarPU's library API, see Performance Model and notably the function starpu_perfmodel_load_symbol(). The source code of the tool `starpu_perfmodel_display` can be a useful example.
An XML output can also be printed by using the `-x` option:

```
$ tools/starpu_perfmodel_display -x -s non_linear_memset_regression_based
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE StarPUPerfmodel SYSTEM "starpu-perfmodel.dtd">
<!-- symbol non_linear_memset_regression_based -->
<!-- All times in us -->
<perfmodel version="45">
  <combination>
    <device type="CPU" id="0" ncores="1"/>
    <implementation id="0">
      <!-- cpu0_impl0 (Comb0) -->
      <!-- time = a size ^b + c -->
      <nl_regression a="5.429195e-04" b="8.654899e-01" c="9.009313e-01"/>
      <entry footprint="a3d3725e" size="4096" flops="0.000000e+00" mean="4.763200e+00" deviation="7.650928e-01
      <entry footprint="870a30aa" size="8192" flops="0.000000e+00" mean="1.827970e+00" deviation="2.037181e-01
      <entry footprint="48e988e9" size="16384" flops="0.000000e+00" mean="2.652800e+00" deviation="1.876459e-0
      <entry footprint="961e65d2" size="32768" flops="0.000000e+00" mean="4.255530e+00" deviation="3.518025e-0
    </implementation>
  </combination>
</perfmodel>
```

The tool `starpu_perfmodel_plot` can be used to draw performance models. It writes a `.gp` file in the current directory, to be run with the tool `gnuplot`, which generates the corresponding curve both in postscript and png format.

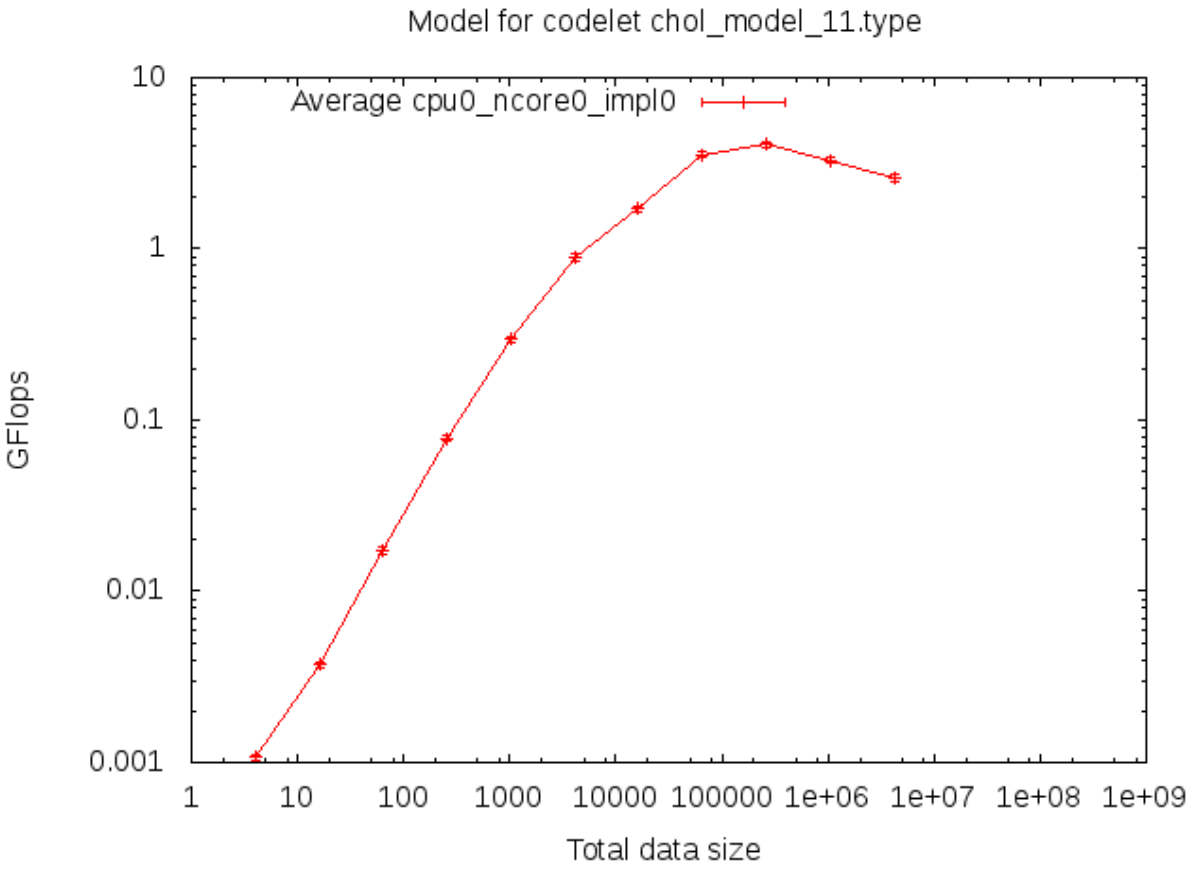
```
$ tools/starpu_perfmodel_plot -s non_linear_memset_regression_based
$ gnuplot starpu_non_linear_memset_regression_based.gp
$ gv starpu_non_linear_memset_regression_based.eps
$ geeqie starpu_non_linear_memset_regression_based.png
```



When the field starpu_task::flops is set (or STARPU_FLOPS is passed to starpu_task_insert()), `starpu_perfmodel_plot` can directly draw a GFlops/s curve, by simply adding the `-f` option:

```
$ starpu_perfmodel_plot -f -s chol_model_potrf
```

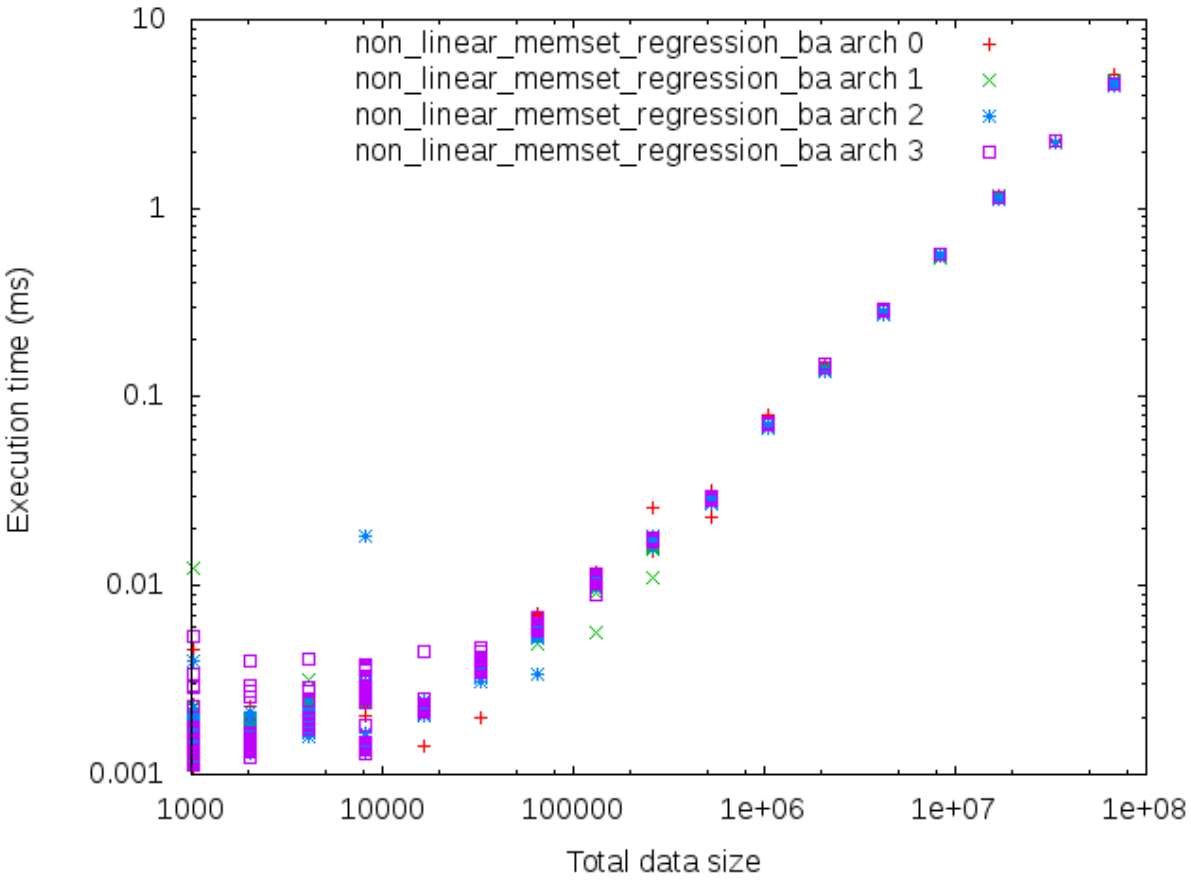
This will however disable displaying the regression model, for which we can not compute GFlops/s.



When the FxT trace file `prof_file_something` has been generated, it is possible to get a profiling of each codelet by calling:

```
$ starpu_fxt_tool -i /tmp/prof_file_something
$ starpu_codelet_profile distrib.data codelet_name
```

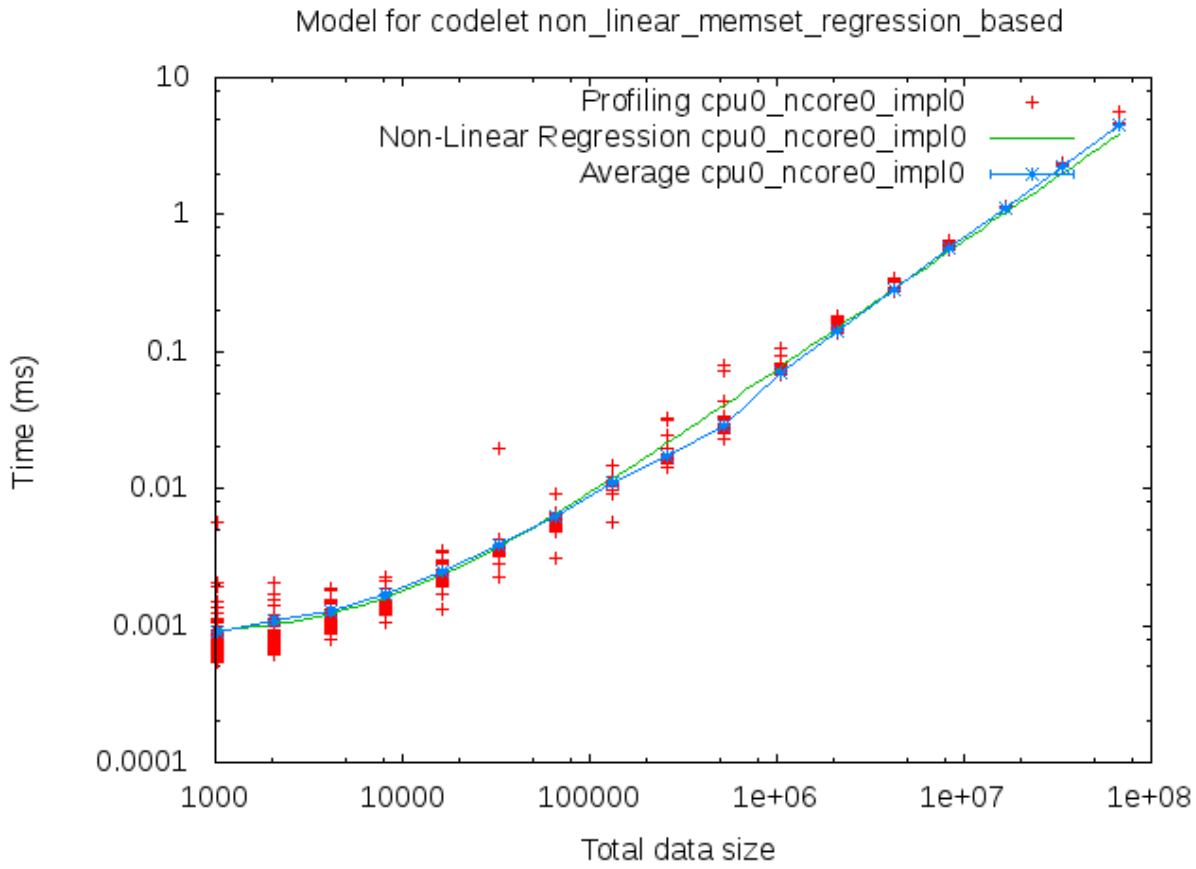
This will create profiling data files, and a `distrib.data.gp` file in the current directory, which draws the distribution of codelet time over the application execution, according to data input size.



This is also available in the tool `starpu_perfmodel_plot`, by passing it the fxt trace:

```
$ starpu_perfmodel_plot -s non_linear_memset_regression_based -i /tmp/prof_file_foo_0
```

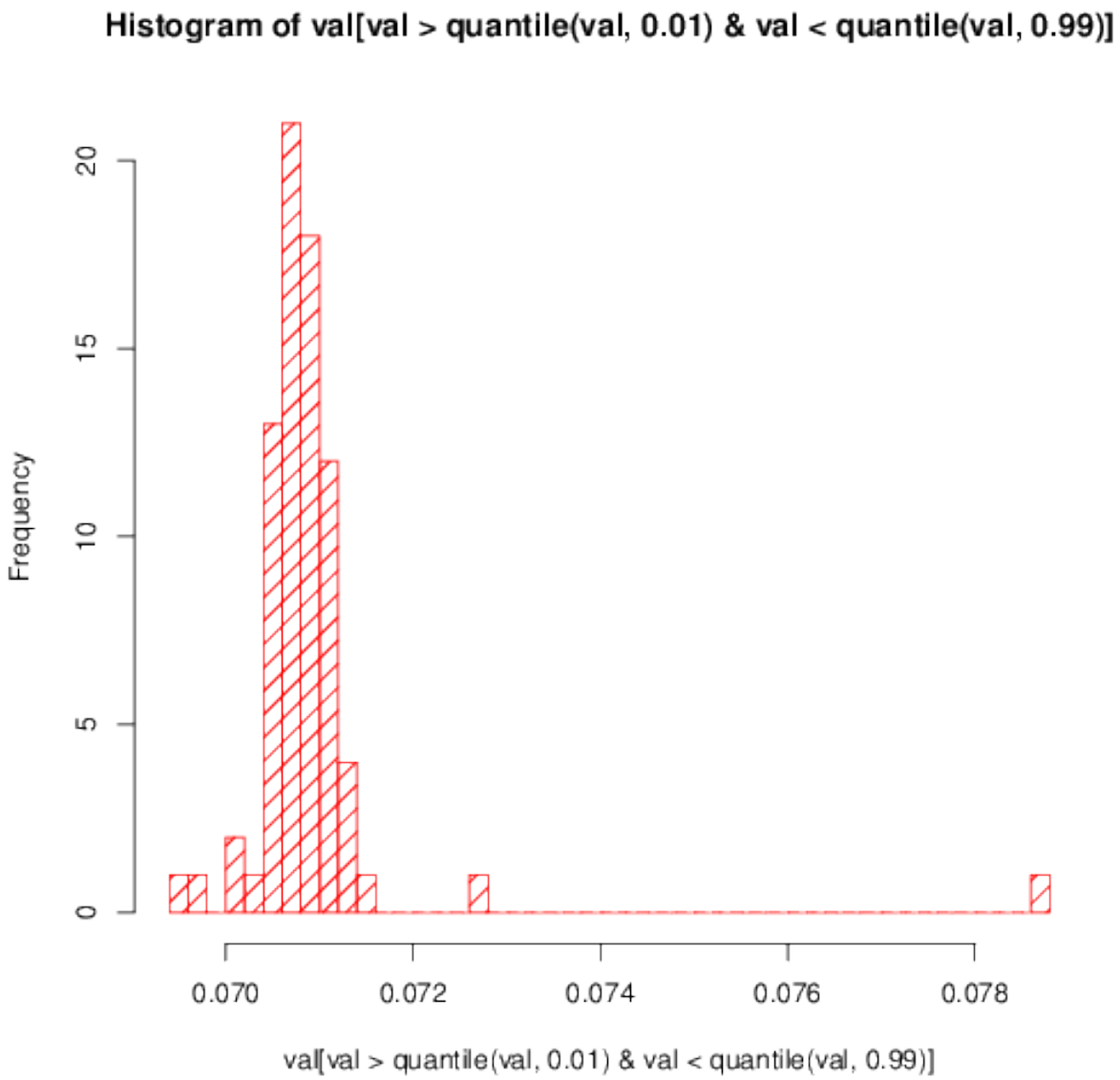
It will produce a `.gp` file which contains both the performance model curves, and the profiling measurements.



If you have the statistical tool `R` installed, you can additionally use

```
$ starpu_codelet_histo_profile distrib.data
```

Which will create one `.pdf` file per codelet and per input size, showing a histogram of the codelet execution time distribution.

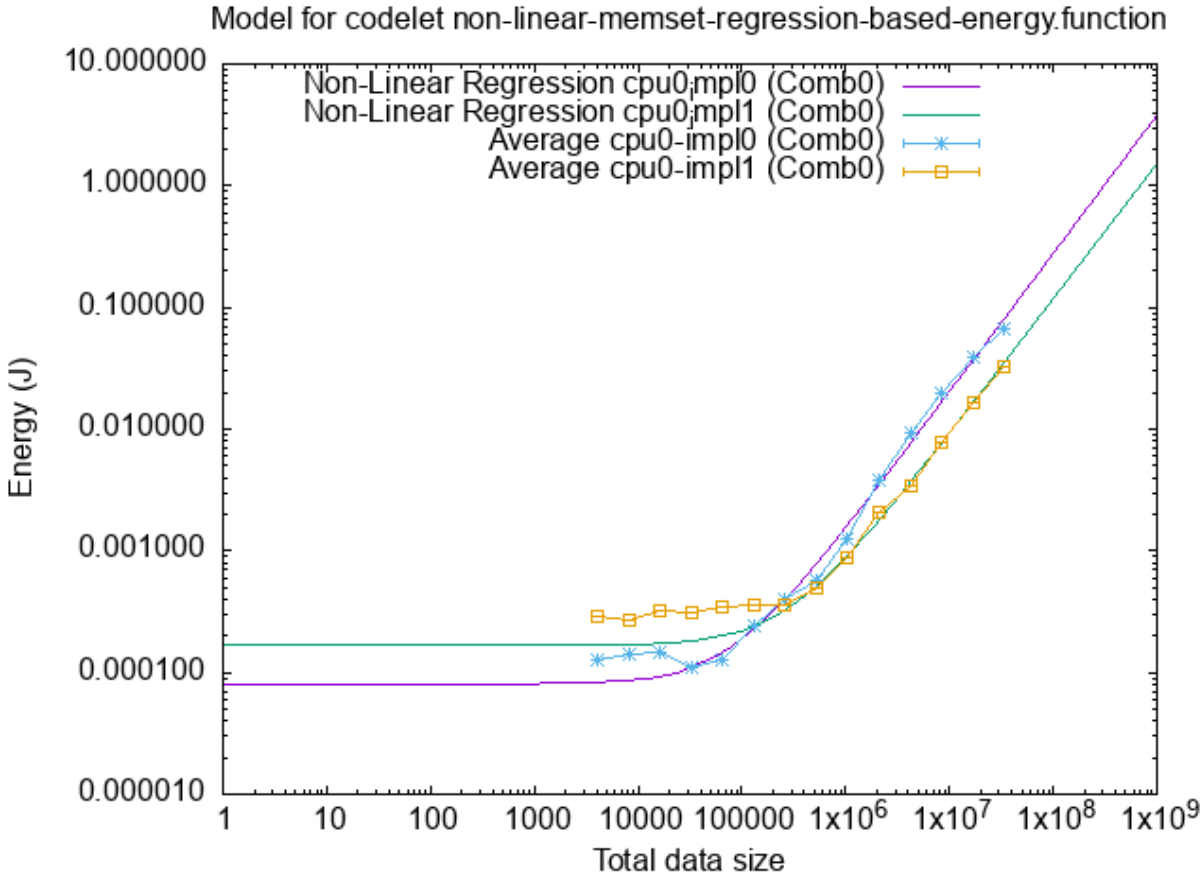


4.3 Energy Of Codelets

A performance model of the energy of codelets can also be recorded thanks to the starpu_codelet::energy_model field of the starpu_codelet structure. StarPU usually cannot record this automatically, since the energy measurement probes are usually not fine-grain enough. It is however possible to measure it by writing a program that submits batches of tasks, let StarPU measure the energy requirement of the batch, and compute an average, see MeasuringEnergyandPower .

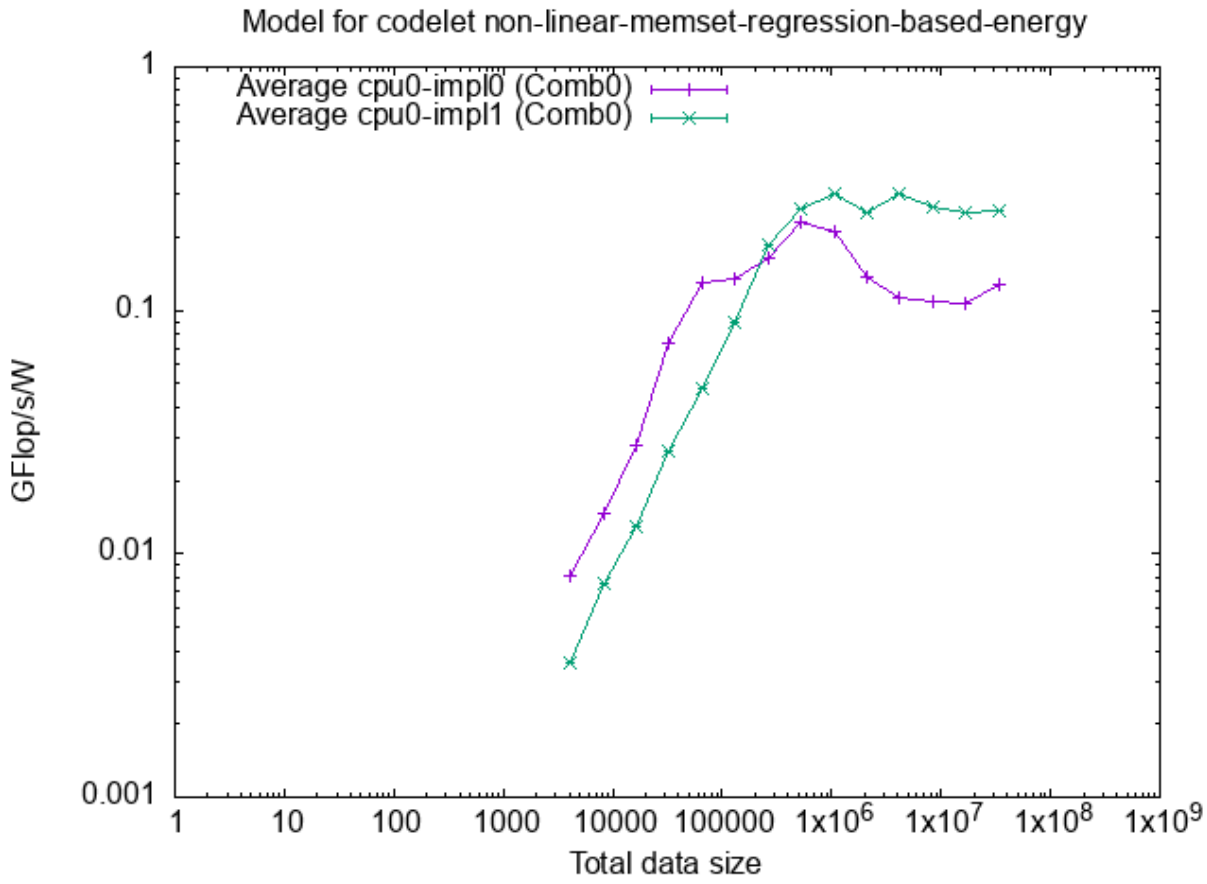
The energy performance model can then be displayed in Joules with `starpu_perfmodel_display` just like the time performance model. The `starpu_perfmodel_plot` needs an extra `-e` option to display the proper unit in the graph:

```
$ tools/starpu_perfmodel_plot -e -s non_linear_memset_regression_based_energy
$ gnuplot starpu_non_linear_memset_regression_based_energy.gp
$ gv starpu_non_linear_memset_regression_based_energy.eps
```



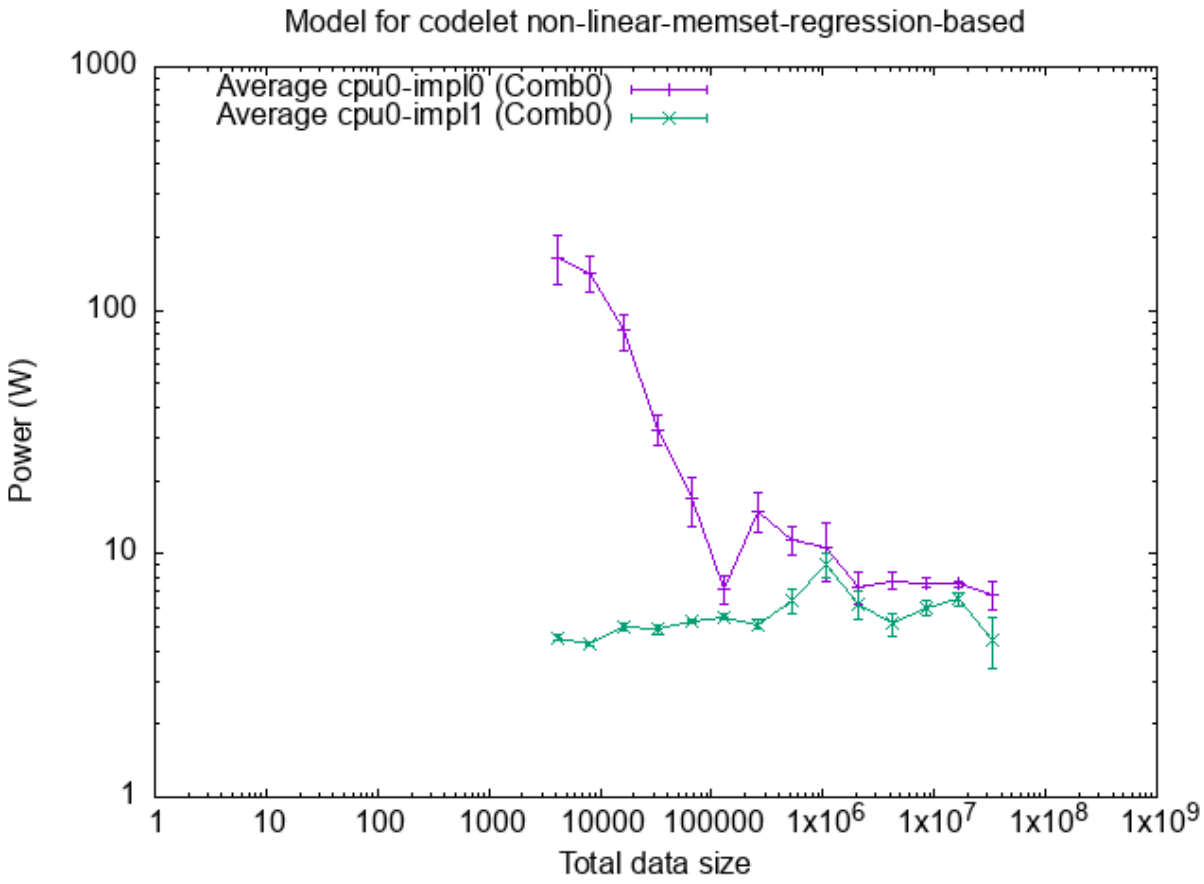
The `-f` option can also be used to display the performance in terms of GFlops/s/W, i.e. the efficiency:

```
$ tools/starpu_perfmodel_plot -f -e -s non_linear_memset_regression_based_energy
$ gnuplot starpu_gflops_non_linear_memset_regression_based_energy.gp
$ gv starpu_gflops_non_linear_memset_regression_based_energy.eps
```



We clearly see here that it is much more energy-efficient to stay in the L3 cache.
One can combine the two time and energy performance models to draw Watts:

```
$ tools/starpu_perfmodel_plot -se non_linear_memset_regression_based non_linear_memset_regression_based_energy
$ gnuplot starpu_power_non_linear_memset_regression_based.gp
$ gv starpu_power_non_linear_memset_regression_based.eps
```

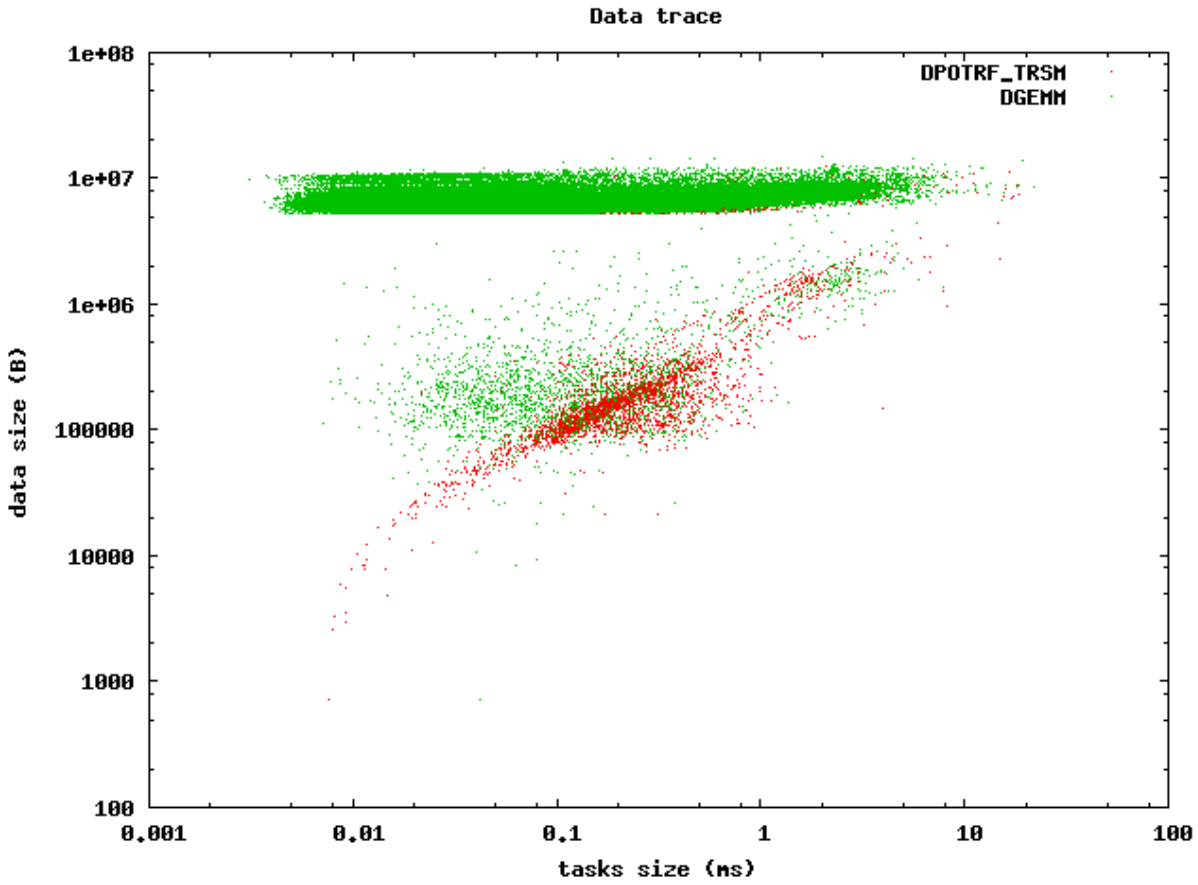


4.4 Data trace and tasks length

It is possible to get statistics about tasks length and data size by using :

```
$ starpu_fxt_data_trace filename [codelet1 codelet2 ... codeletn]
```

Where filename is the FxT trace file and codeletX the names of the codelets you want to profile (if no names are specified, `starpu_fxt_data_trace` will profile them all). This will create a file, `data_trace.gp` which can be executed to get a `.eps` image of these results. On the image, each point represents a task, and each color corresponds to a codelet.



4.5 Trace Statistics

More than just codelet performance, it is interesting to get statistics over all kinds of StarPU states (allocations, data transfers, etc.). This is particularly useful to check what may have gone wrong in the accuracy of the SimGrid simulation.
This requires the `R` statistical tool, with the `plyr`, `ggplot2` and `data.table` packages. If your system distribution does not have packages for these, one can fetch them from `CRAN`:

```
$ R
> install.packages("plyr")
> install.packages("ggplot2")
> install.packages("data.table")
> install.packages("knitr")
```

The `pj_dump` tool from `pajeng` is also needed (see https://github.com/schnorr/pajeng)
One can then get textual or `.csv` statistics over the trace states:

```
$ starpu_paje_state_stats -v native.trace simgrid.trace
"Value"            "Events_native.csv" "Duration_native.csv" "Events_simgrid.csv" "Duration_simgrid.csv"
"Callback"         220                 0.075978              220                 0
"chol_model_potrf" 10                  565.176               10                  572.8695
"chol_model_trsm"  45                  9184.828              45                  9170.719
"chol_model_gemm"  165                 64712.07              165                 64299.203
$ starpu_paje_state_stats native.trace simgrid.trace
```

An other way to get statistics of StarPU states (without installing `R` and `pj_dump`) is to use the `starpu_trace_state_stats.py` script, which parses the generated `trace.rec` file instead of the `paje.trace` file. The output is similar to the previous script, but it doesn't need any dependencies.
The different prefixes used in `trace.rec` are:

```
E: Event type
N: Event name
```

```
C: Event category
W: Worker ID
T: Thread ID
S: Start time
```

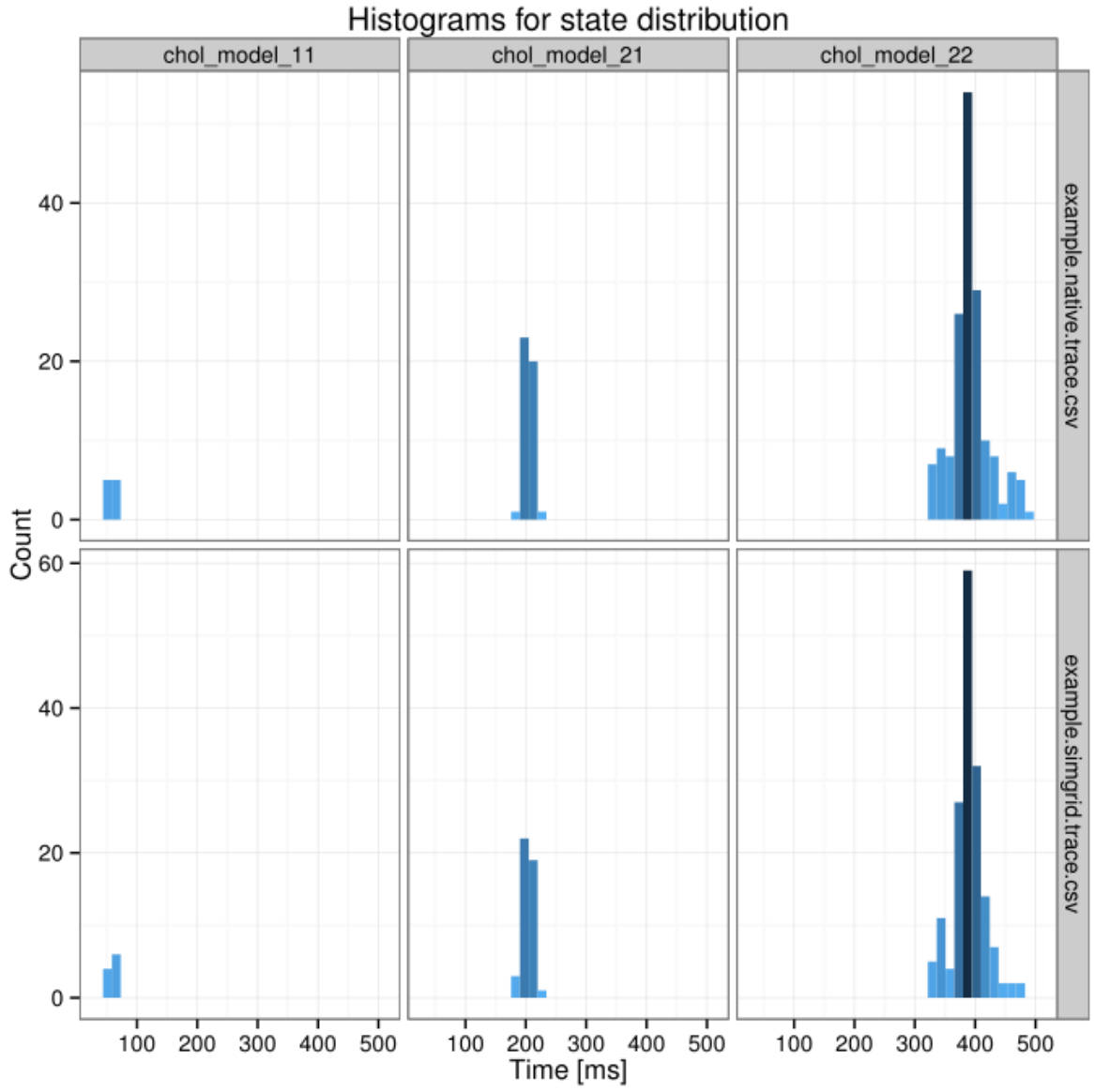
Here's an example on how to use it:

```
$ starpu_trace_state_stats.py trace.rec | column -t -s ","
"Name"             "Count" "Type"       "Duration"
"Callback"          220 Runtime 0.075978
"chol_model_potrf"  10  Task    565.176
"chol_model_trsm"   45  Task    9184.828
"chol_model_gemm"   165 Task    64712.07
```

`starpu_trace_state_stats.py` can also be used to compute the different efficiencies. Refer to the usage description to show some examples.
And one can plot histograms of execution times, of several states, for instance:

```
$ starpu_paje_draw_histogram -n chol_model_potrf,chol_model_trsm,chol_model_gemm native.trace simgrid.trace
```

and see the resulting pdf file:



A quick statistical report can be generated by using:

```
$ starpu_paje_summary native.trace simgrid.trace
```

it includes gantt charts, execution summaries, as well as state duration charts and time distribution histograms.
Other external Paje analysis tools can be used on these traces, one just needs to sort the traces by timestamp order (which not guaranteed to make recording more efficient):

```
$ starpu_paje_sort paje.trace
```

4.6 PAPI counters

Performance counter values could be obtained from the PAPI framework if `./configure` detected the libpapi.
In Debian, the `libpapi-dev` package provides the required files. Additionally, the `papi-tools` package contains a set of useful tools, for example `papi_avail` to see which counters are available.
To be able to use Papi counters, one may need to reduce the level of the kernel parameter `kernel.perf_event_paranoid` to 2 or below. See https://www.kernel.org/doc/html/latest/admin-guide/perf-secu
html for the security impact of this parameter.
Then one has to set the STARPU_PROFILING environment variable to 1 and specify which events to record with the STARPU_PROF_PAPI_EVENTS environment variable. For instance:

```
export STARPU_PROFILING=1 STARPU_PROF_PAPI_EVENTS="PAPI_TOT_INS PAPI_TOT_CYC"
```

The comma can also be used to separate events to monitor.
In the current simple implementation, only CPU tasks have their events measured and require CPUs that support the PAPI events. It is important to note that not all events are available on all systems, and general PAPI recommendations should be followed.
The counter values can be accessed using the profiling interface:
`task->profiling_info->papi_values`
Also, it can be accessed and/or saved with tracing when using STARPU_FXT_TRACE. With the use of `starpu_fxt_tool` the file `papi.rec` is generated containing the following triple:

```
Task Id
Event Id
Value
```

External tools like `rec2csv` can be used to convert this rec file to a `csv` file, where each line represents a value for an event for a task.

4.7 Theoretical Lower Bound On Execution Time

StarPU can record a trace of what tasks are needed to complete the application, and then, by using a linear system, provide a theoretical lower bound of the execution time (i.e. with an ideal scheduling).
The computed bound is not really correct when not taking into account dependencies, but for an application which have enough parallelism, it is very near to the bound computed with dependencies enabled (which takes a huge lot more time to compute), and thus provides a good-enough estimation of the ideal execution time.
Then there is an example to show how to use this.
For kernels with history-based performance models (and provided that they are completely calibrated), StarPU can very easily provide a theoretical lower bound for the execution time of a whole set of tasks. See for instance `examples/lu/lu_example.c`: before submitting tasks, call the function starpu_bound_start(), and after complete execution, call starpu_bound_stop(). starpu_bound_print_lp() or starpu_bound_print_mps() can then be used to output a Linear Programming problem corresponding to the schedule of your tasks. Or starpu_bound_print_dot() can be used to print a task dependency graph in the DOT format. Run it through `lp_solve` or any other linear programming solver, and that will give you a lower bound for the total execution time of your tasks. If StarPU was compiled with the library `glpk` installed, starpu_bound_compute() can be used to solve it immediately and get the optimized minimum, in ms. Its parameter `integer` allows deciding whether integer resolution should be computed and returned. Besides to solve it immediately and get the optimized minimum starpu_bound_print() can also print the statistics of actual execution and theoretical upper bound.
The `deps` parameter tells StarPU whether to take tasks, implicit data, and tag dependencies into account. Tags released in a callback or similar are not taken into account, only tags associated with a task are. It must be understood that the linear programming problem size is quadratic with the number of tasks and thus the time to solve it will be very long, it could be minutes for just a few dozen tasks. You should probably use `lp_solve`

`-timeout 1 test.pl -wmps test.mps` to convert the problem to MPS format and then use a better solver, `glpsol` might be better than `lp_solve` for instance (the `-pcost` option may be useful), but sometimes doesn't manage to converge. `cbc` might look slower, but it is parallel. For `lp_solve`, be sure to try at least all the `-B` options. For instance, we often just use `lp_solve -cc -B1 -Bb -Bg -Bp -Bf -Br -BG -Bd -Bs -BB -Bo -Bc -Bi` , and the `-gr` option can also be quite useful. The resulting schedule can be observed by using the tool `starpu_lp2paje`, which converts it into the Paje format.
Data transfer time can only be taken into account when `deps` is set. Only data transfers inferred from implicit data dependencies between tasks are taken into account. Other data transfers are assumed to be completely overlapped.
Setting `deps` to 0 will only take into account the actual computations on processing units. However, it still properly takes into account the varying performances of kernels and processing units, which is quite more accurate than just comparing StarPU performances with the fastest of the kernels being used.
The `prio` parameter tells StarPU whether to simulate taking into account the priorities as the StarPU scheduler would, i.e. schedule prioritized tasks before less prioritized tasks, to check to which extend this results to a less optimal solution. This increases even more computation time.

4.8 Trace visualization with StarVZ

Creating views with StarVZ (see: https://github.com/schnorr/starvz) is made up of two steps. The initial stage consists of a pre-processing of the traces generated by the application, while the second one consists of the analysis itself and is carried out with R packages' aid. StarVZ is available at CRAN (https://cran.r-project.org/package=starvz) and depends on `pj_dump` (from `pajeng`) and `rec2csv` (from `recutils`).
To download and install StarVZ, it is necessary to have `R`, `pajeng`, and `recutils`:

```
# For pj_dump and rec2csv
apt install -y pajeng recutils

# For R
apt install -y r-base libxml2-dev libssl-dev libcurl4-openssl-dev libgit2-dev libboost-dev
```

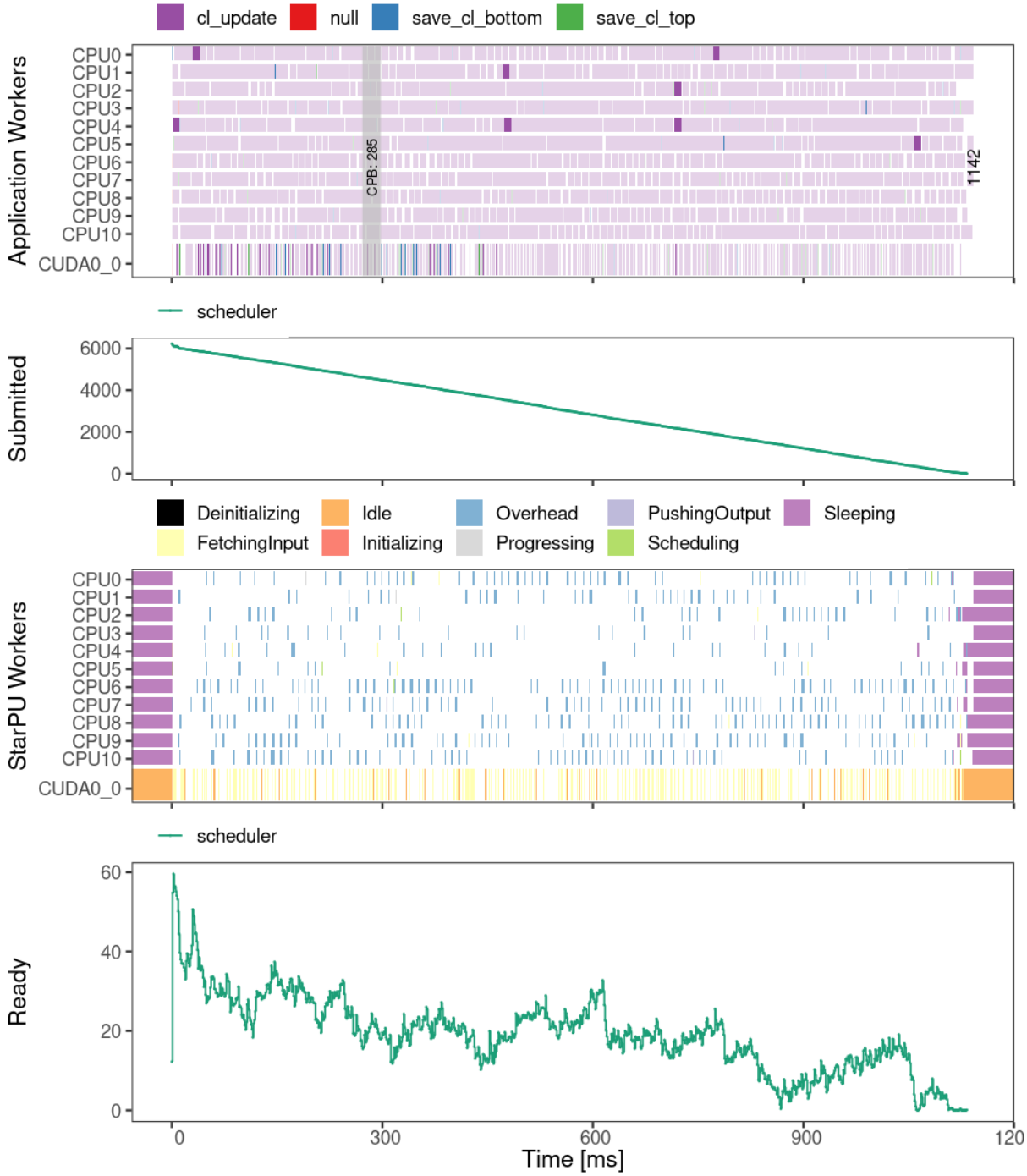
To install the StarVZ, the following command can be used:

```
echo "install.packages('starvz', repos = 'https://cloud.r-project.org')" | R --vanilla
```

To generate traces from an application, it is necessary to set STARPU_GENERATE_TRACE and build StarPU with FxT. Then, StarVZ can be used on a folder with StarPU FxT traces to produce a default view:

```
export PATH=$(Rscript -e 'cat(system.file("tools/", package = "starvz"), sep="\n")'):$PATH

starvz /foo/path-to-fxt-files
```

An example of default view:



One can also use existing trace files (`paje.trace`, `tasks.rec`, `data.rec`, `papi.rec` and `dag.dot`) skipping the StarVZ internal call to starpu_fxt_tool with:

```
starvz --use-paje-trace /foo/path-to-trace-files
```

Alternatively, each StarVZ step can be executed separately. Step 1 can be used on a folder with:

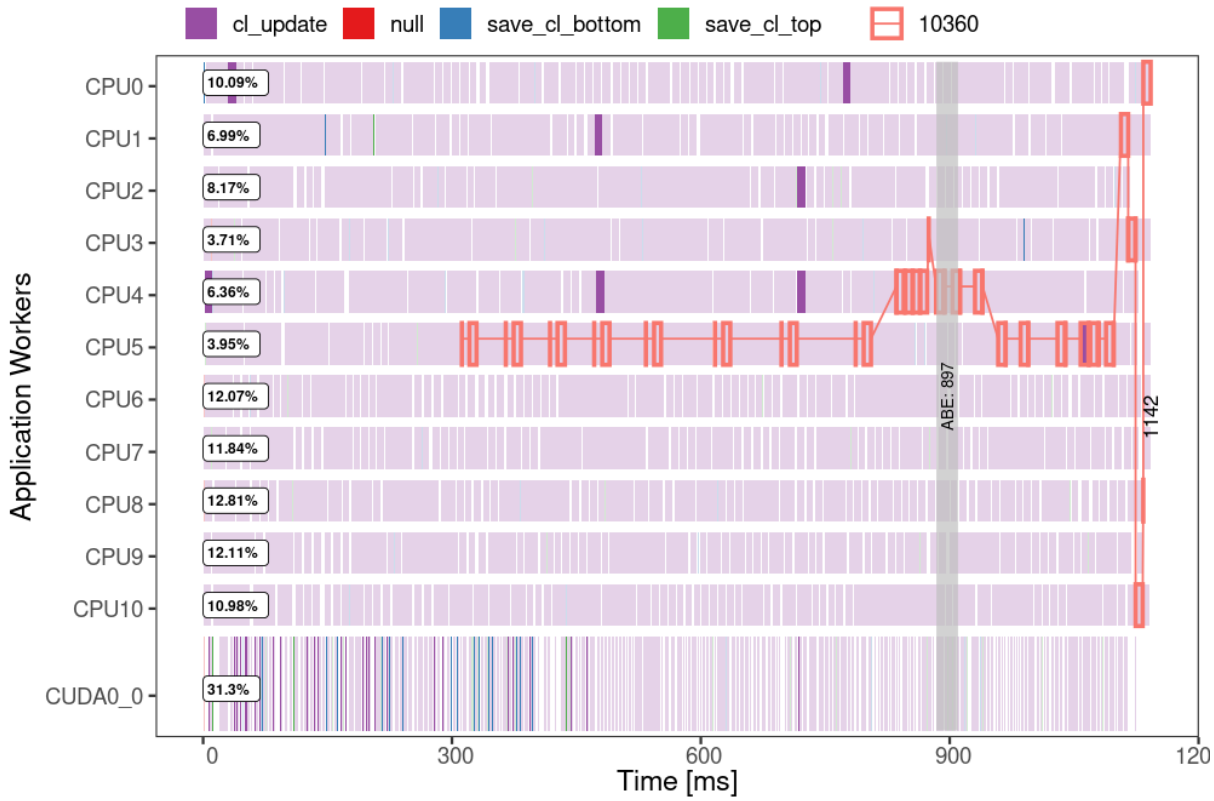
```
starvz -1 /foo/path-to-fxt-files
```

Then the second step can be executed directly in R. StarVZ enables a set of different plots that can be configured on a .yaml file. A default file is provided (`default.yaml`); also, the options can be changed directly in R.

```
library(starvz)
library(dplyr)
```

```
dtrace <- starvz_read("./", selective = FALSE)

# show idleness ratio
dtrace$config$st$idleness = TRUE

# show ABE bound
dtrace$config$st$abe$active = TRUE

# find the last task with dplyr
dtrace$config$st$tasks$list = dtrace$Application %>% filter(End == max(End)) %>% .$JobId
# show last task dependencies
dtrace$config$st$tasks$active = TRUE
dtrace$config$st$tasks$levels = 50

plot <- starvz_plot(dtrace)
```

An example of visualization follows:

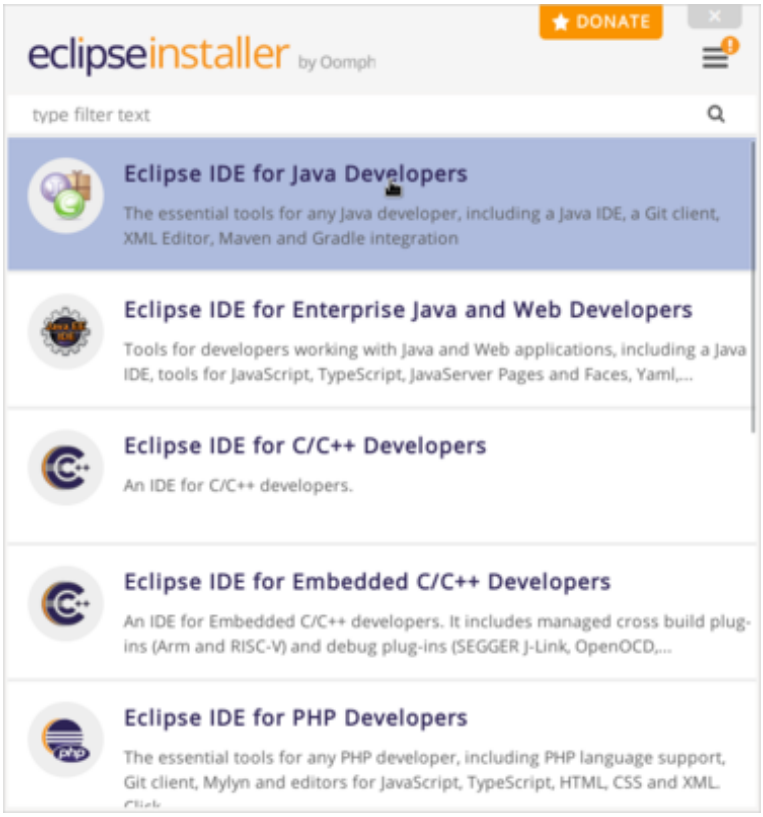


4.9 StarPU Eclipse Plugin

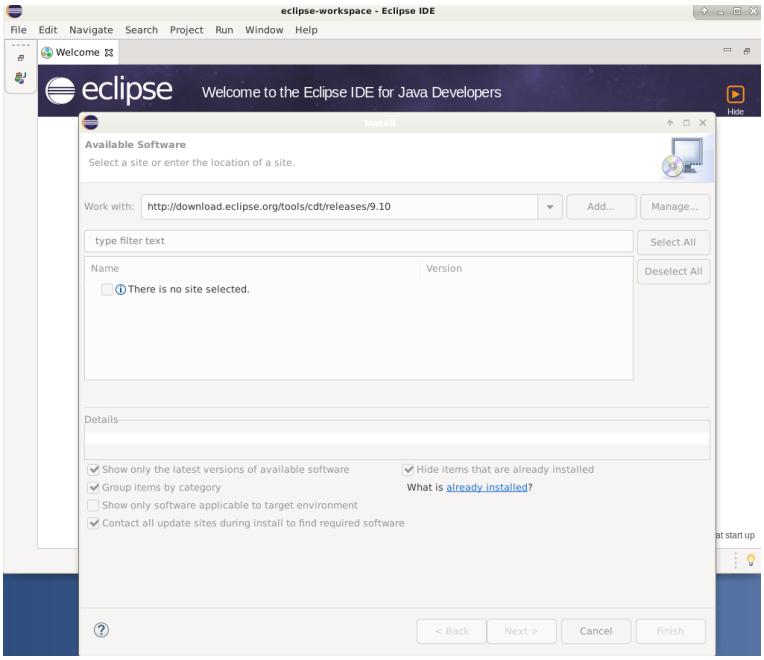
The StarPU Eclipse Plugin provides the ability to generate the different traces directly from the Eclipse IDE.

4.9.1 Eclipse Installation

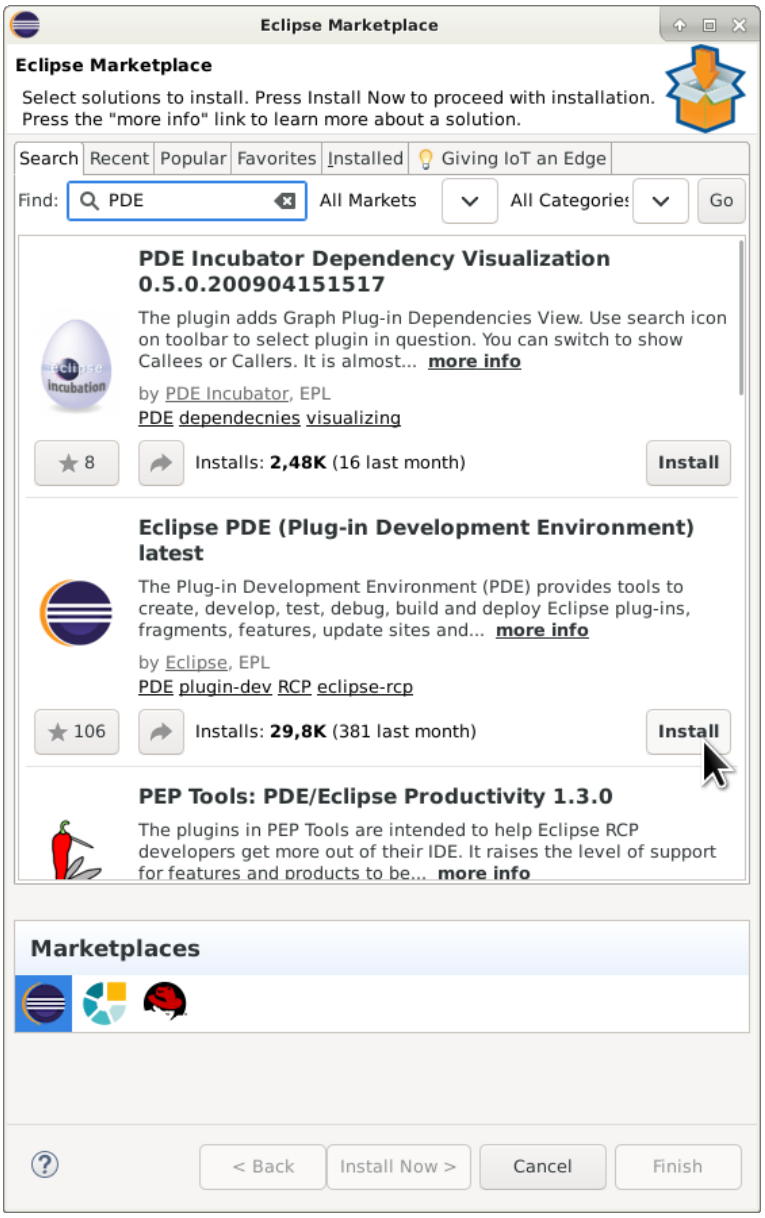
Download the Eclipse installer from https://www.eclipse.org/downloads/packages/installer. When you run the installer, click on *Eclipse IDE for Java Developers* to start the installation process.



To be able to develop C/C++ applications, you need to install the CDT plugin. To do so, go to the *Help* dropdown menu at the top of the Eclipse window, choose *Install New Software ...*. In the new window, enter the URL `http://download.eclipse.org/tools/cdt/releases/9.10` into the box *Work with* and press the return key.



You need then to select *CDT Main Features*, then click the button *Next* twice, accept the terms of the license, and click the button *Finish*. Eclipse will ask you to restart.
To be able to compile the plugin, you need to install the plugin development environment (PDE). To do so, go to the menu *Help*, choose *Eclipse Marketplace...*. In the new window, enter *PDE* into the box *Find* and press the return key.



You can then click on the button *Install* of the *Eclipse PDE latest*. You may need to confirm the installation, then accept the terms of the license, and finally restart the Eclipse IDE.
The installation is now done.

4.9.2 StarPU Eclipse Plugin Compilation and Installation

StarPU can now be compiled and installed with its Eclipse plugin. To do so, you first need to configure StarPU with the option --enable-eclipse-plugin. The Eclipse IDE executable `eclipse` must be in your `PATH`.

```
export PATH=$HOME/usr/local/eclipse/java-2021-03/eclipse:$PATH
mkdir build
cd build
../configure --prefix=$HOME/usr/local/starpu --enable-eclipse-plugin
make
make install
```

The StarPU Eclipse plugin is installed in the directory `dropins`.

```
$ ls $HOME/usr/local/eclipse/java-2021-03/eclipse/dropins
StarPU_1.0.0.202105272056.jar
```

In the next section, we will show you how to use the plugin.

4.9.3 StarPU Eclipse Plugin Instruction

Once StarPU has been configured and installed with its Eclipse plugin, you first need to set up your environment for StarPU.

```
cd $HOME/usr/local/starpu
source ./bin/starpu_env
```

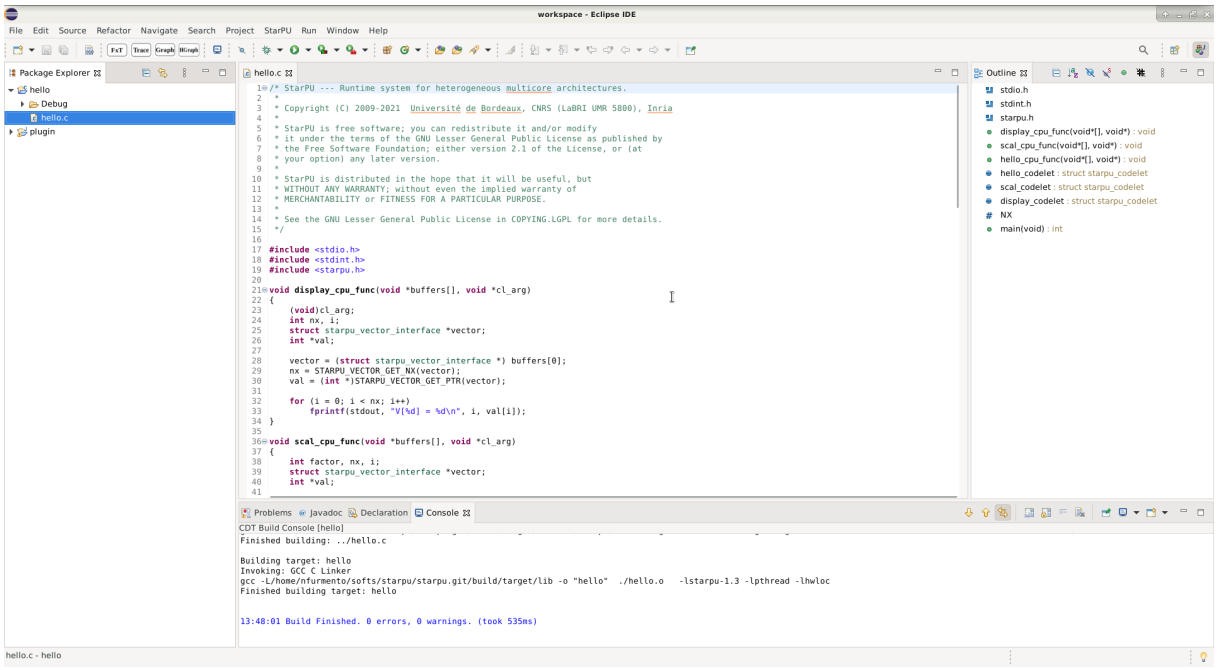
To generate traces from the application, it is necessary to set STARPU_FXT_TRACE to 1.

```
export STARPU_FXT_TRACE=1
```

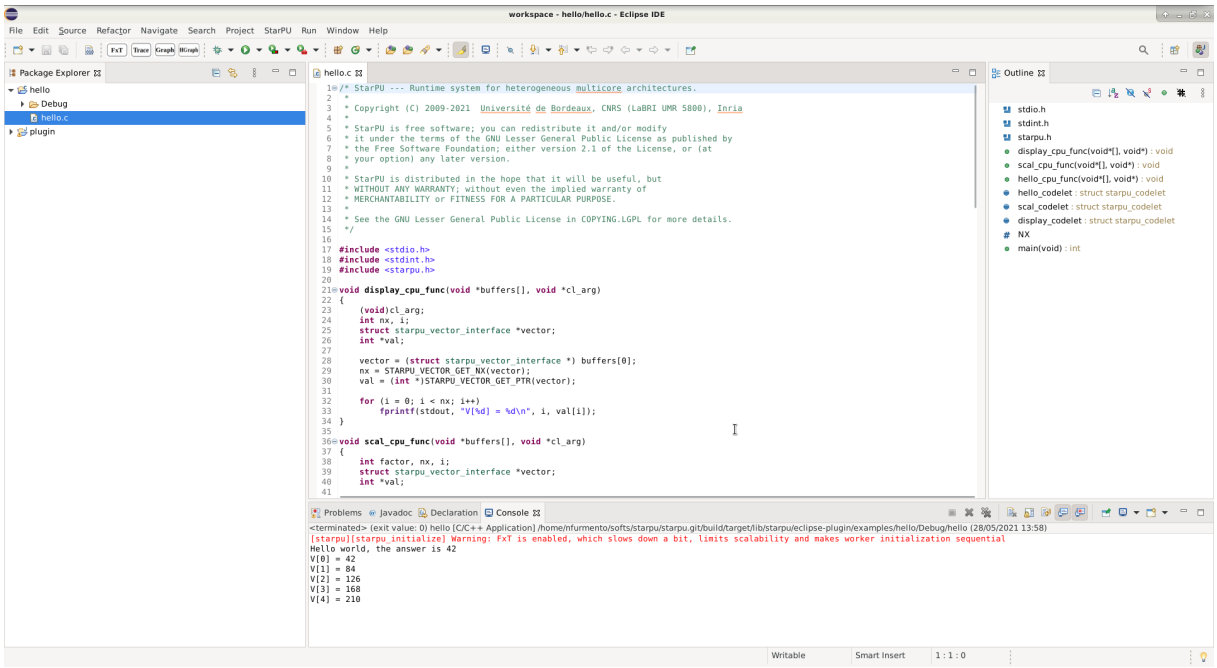
The eclipse workspace together with an example is available in `lib/starpu/eclipse-plugin`.

```
cd ./lib/starpu/eclipse-plugin
eclipse -data workspace
```

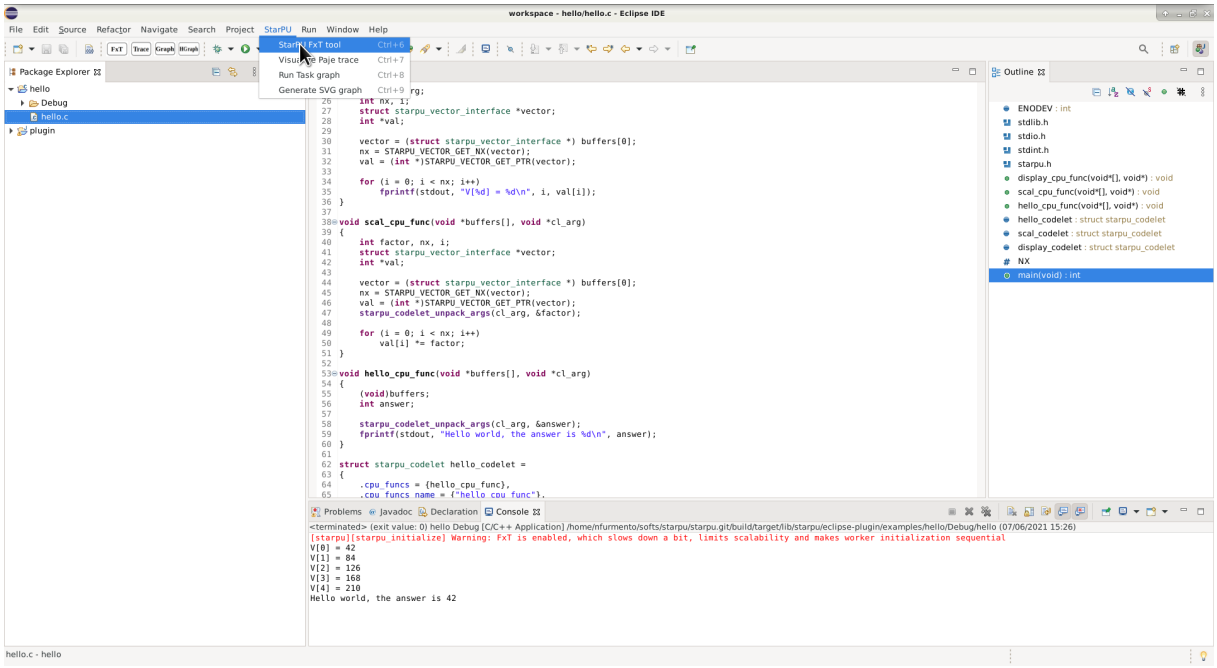
You can then open the file `hello/hello.c`, and build the application by pressing `Ctrl-B`.



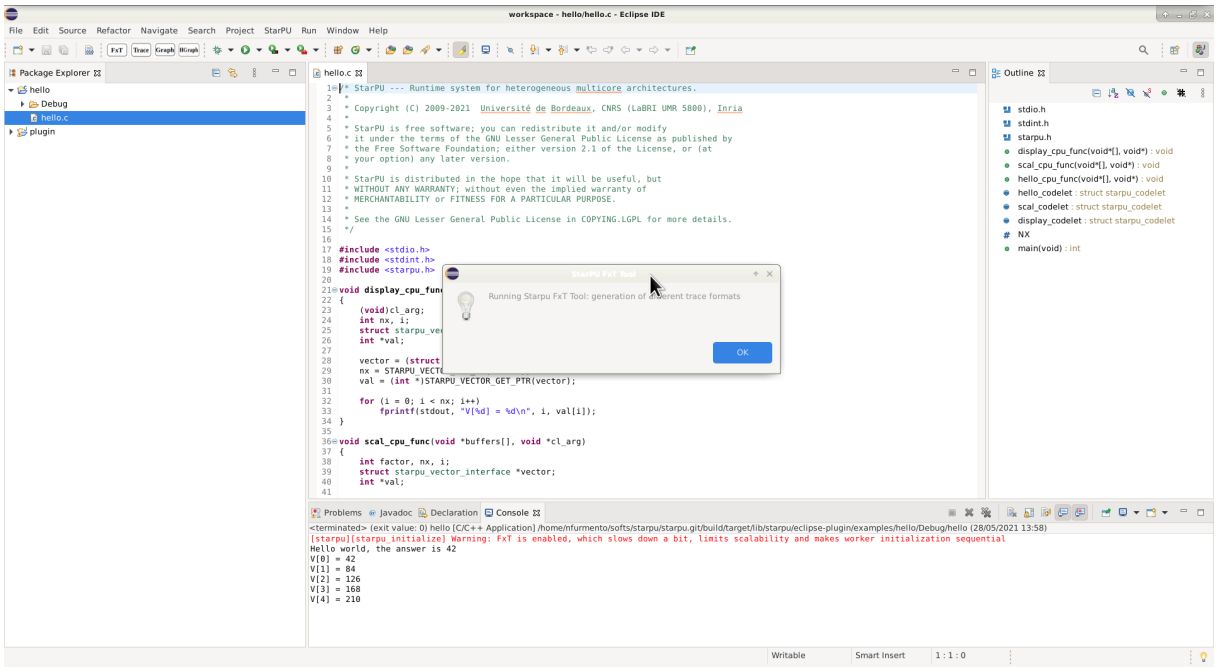
The application can now be executed.



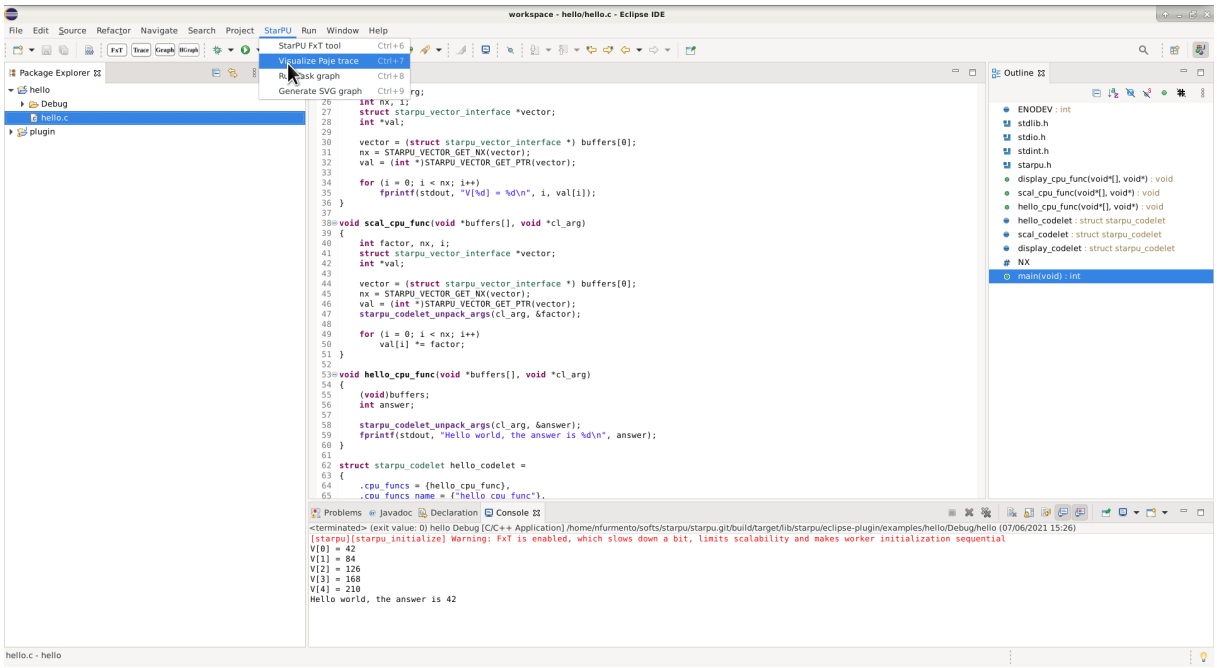
After executing the C/C++ StarPU application, one can use the StarPU plugin to generate and visualise the task graph of the application. The StarPU plugin eclipse is either available through the icons in the upper toolbar, or from the dropdown menu `StarPU`.

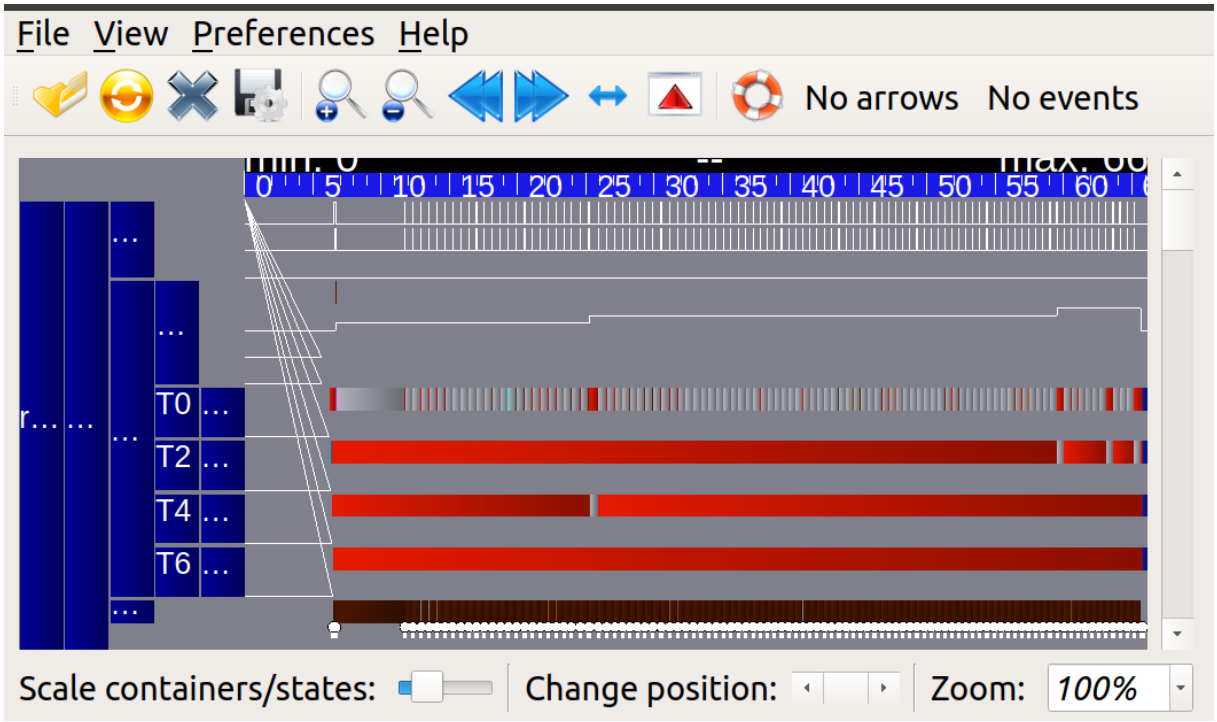


To start, one first need to run the StarPU FxT tool, either through the `FxT` icon of the toolbar, or from the menu `StarPU` / `StarPU FxT Tool`. This will call the tool `starpu_fxt_tool` to generate traces for your application execution.
A message dialog box is displayed to confirm the generation of the different traces.

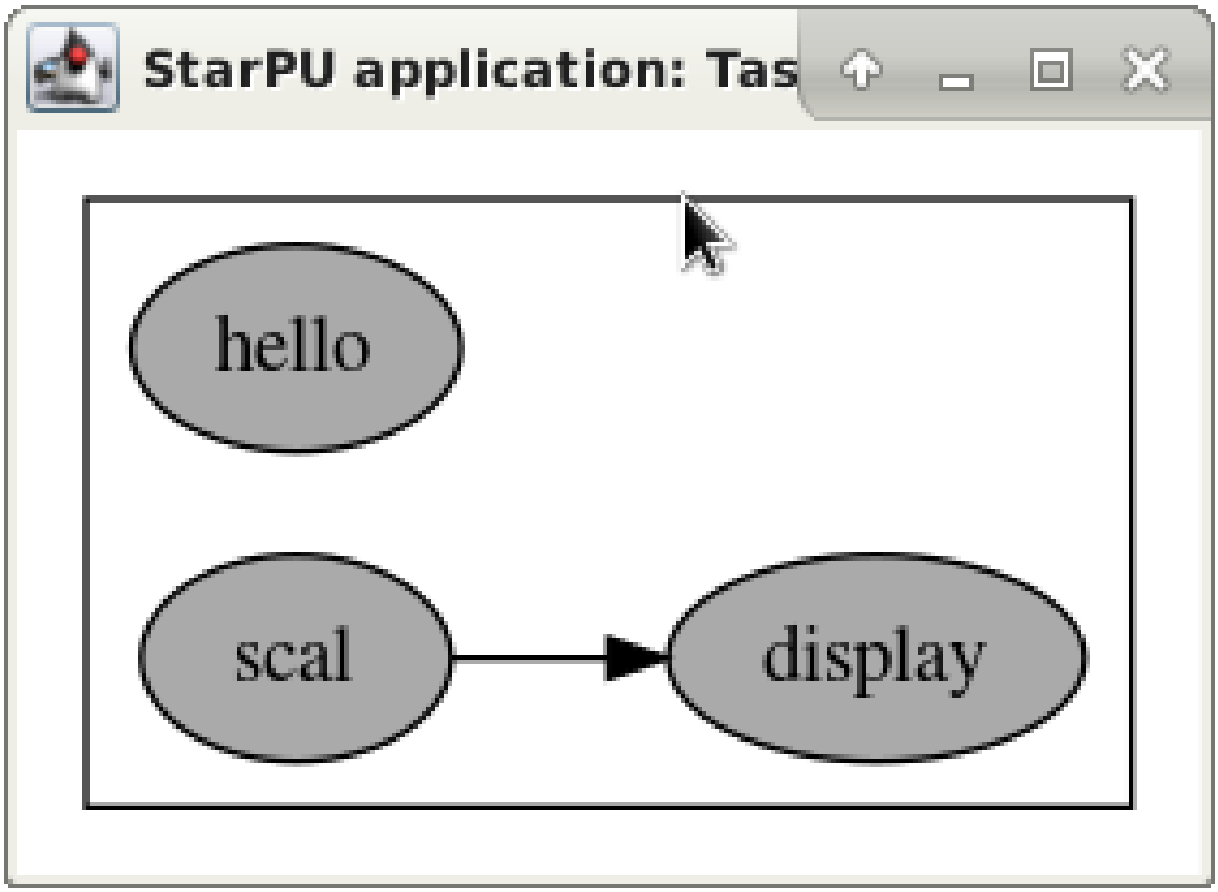


One of the generated files is a Paje trace which can be viewed with ViTE, a trace explorer. To open and visualise the file `paje.trace` with ViTE, one can select the second command of the StarPU menu, which is named `Generate Paje Trace`, or click on the second icon named `Trace` in the toolbar.

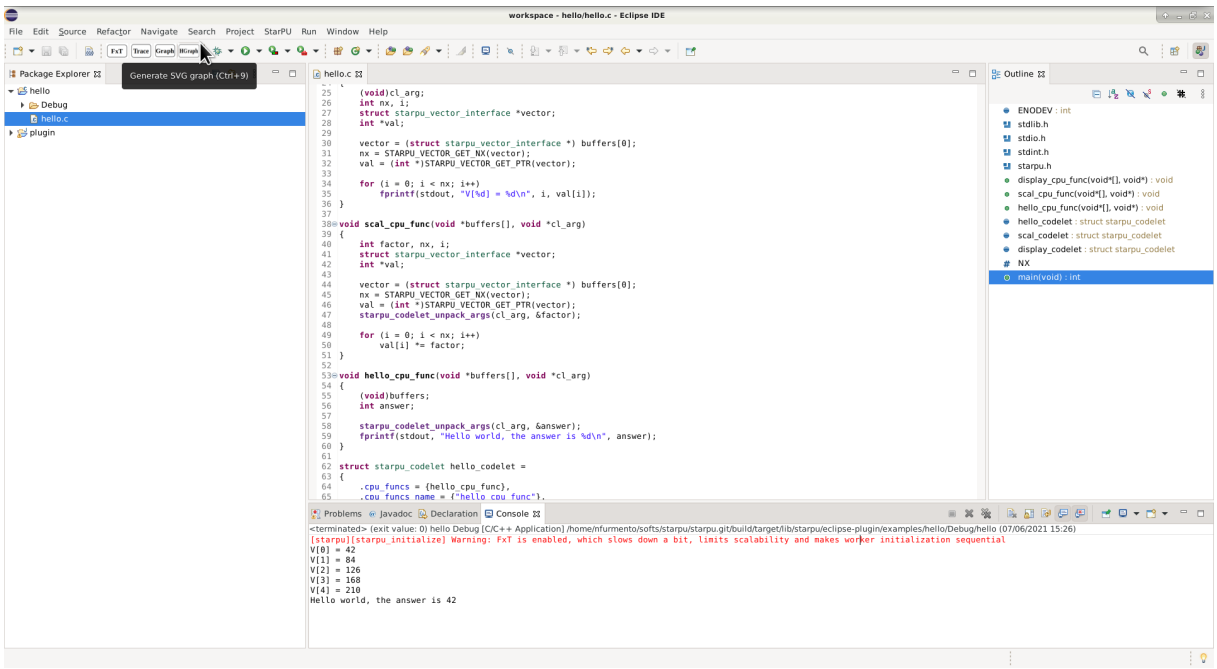


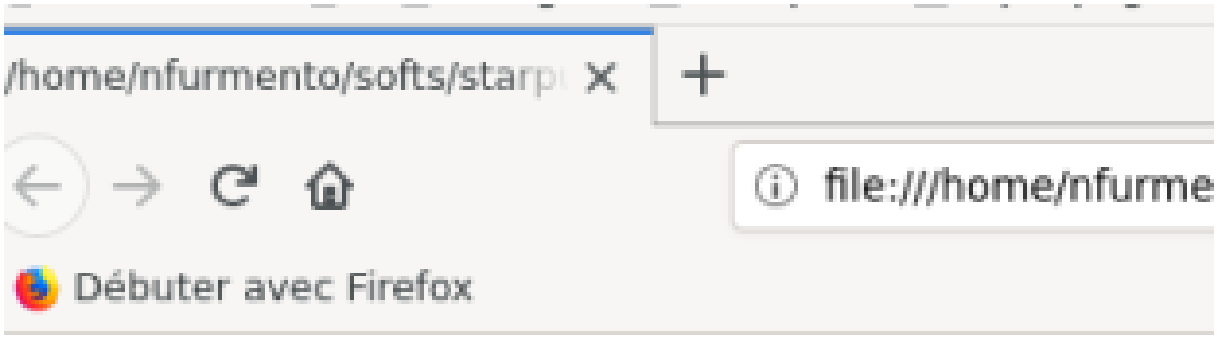


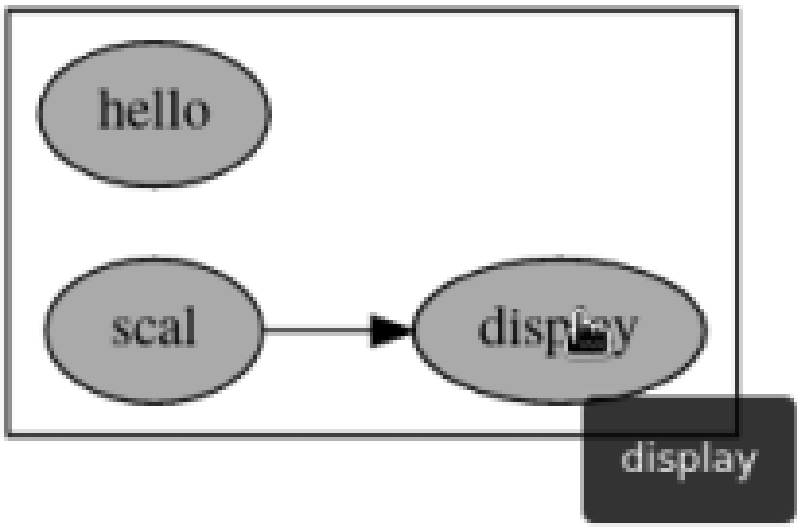
Another generated trace file is a task graph described using the DOT language. It is possible to get a graphical output of the graph by calling the `graphviz library`. To do this, one can click on the third command of StarPU menu. A task graph of the application in the `png` format is then generated.



In StarPU eclipse plugin, one can display the graph task directly from eclipse, or through a web browser. To do this, there is another command named `Generate SVG graph` in the StarPU menu or HGraph in the toolbar of eclipse.
From the HTML file, you can see the graph task, and by clicking on a task name, it will open the C file in which the task submission was called (if you have an editor which understands the syntax `href="file.c#123"`).







4.10 Memory Feedback

It is possible to enable memory statistics. To do so, you need to pass the option --enable-memory-stats when running `configure`. It is then possible to call the function starpu_data_display_memory_stats() to display statistics about the current data handles registered within StarPU.
Moreover, statistics will be displayed at the end of the execution on data handles which have not been cleared out. This can be disabled by setting the environment variable STARPU_MEMORY_STATS to `0`.
For example, by adding a call to the function starpu_data_display_memory_stats() in the fblock example before unpartitioning the data, one will get something similar to:

```
$ STARPU_MEMORY_STATS=1 ./examples/filters/fblock
...
#---------------------
Memory stats :
#-------
Data on Node #2
#-----
Data : 0x5562074e8670
Size : 144

#--
Data access stats
```

```
/!\ Work Underway
Node #0
        Direct access : 0
        Loaded (Owner) : 0
        Loaded (Shared) : 0
        Invalidated (was Owner) : 1

Node #2
        Direct access : 0
        Loaded (Owner) : 1
        Loaded (Shared) : 0
        Invalidated (was Owner) : 0

#-------
Data on Node #3
#-----
Data : 0x5562074e9338
Size : 96

#--
Data access stats
/!\ Work Underway
Node #0
        Direct access : 0
        Loaded (Owner) : 0
        Loaded (Shared) : 0
        Invalidated (was Owner) : 1

Node #3
        Direct access : 0
        Loaded (Owner) : 1
        Loaded (Shared) : 0
        Invalidated (was Owner) : 0


#---------------------
...
```

4.11 Data Statistics

Different data statistics can be displayed at the end of the execution of the application. To enable them, you need to define the environment variable STARPU_ENABLE_STATS. When calling starpu_shutdown() various statistics will be displayed, execution, MSI cache statistics, allocation cache statistics, and data transfer statistics. The display can be disabled by setting the environment variable STARPU_STATS to `0`. If the environment variable STARPU_BUS_STATS is defined, you can call starpu_profiling_bus_helper_display_summary() to display statistics about the bus. If the environment variable STARPU_WORKER_STATS is defined, you can call starpu_profiling_worker_helper_display_summary() to display statistics about the workers. You can also call starpu_display_stats() which call both starpu_profiling_bus_helper_display_summary() and starpu_profiling_worker_helper_display_summary() at the same time.

```
$ ./examples/cholesky/cholesky_tag
Computation took (in ms)
518.16
Synthetic GFlops : 44.21
#---------------------
MSI cache stats :
TOTAL MSI stats hit 1622 (66.23 %)      miss 827 (33.77 %)
...
```

```
$ STARPU_STATS=0 ./examples/cholesky/cholesky_tag
Computation took (in ms)
518.16
Synthetic GFlop/s : 44.21
```

4.12 Tracing MPI applications

When an MPI execution is traced, especially if the execution is on several nodes, clock synchronization issues can appear. One may notice them mainly on communications (they are received before they are sent, for instance).

Each processor can call the function starpu_profiling_set_id() to set the ID used for the profiling trace filename. This function can be useful when executing an MPI program on several nodes, as it enables each processor to set a unique ID that helps to differentiate its trace file from the files generated by other processors. By doing this, it becomes easier to analyze and compare the profiling results of each processor separately, which is particularly helpful for large-scale parallel applications.
By default, StarPU does two MPI barriers with all MPI processes: one at the beginning of the application execution and one at the end. Then, `starpu_fxt_tool` considers all processes leave the barriers at the exact same time, which makes two points for time synchronization between MPI processes.
However, a simple MPI barrier can be not precise enough, because the assumption *all processes leave the barriers at the exact same time* is in reality false. To have a more precise barrier, one may use the `mpi_sync_clocks library` (automatically provided when StarPU is built with NewMadeleine, but it can also be used with other MPI libraries). It provides a *synchronized* barrier, which aims at actually releasing all processes at the exact same time. Unfortunately, the gained precision costs some time (several seconds per barrier), that is why one can disable this precise synchronization with the environment variable STARPU_MPI_TRACE_SYNC_CLOCKS set to `0`, and use the faster MPI barrier instead.

4.13 Verbose Traces

Traces can also be inspected by hand by using the tool `fxt_print`, for instance:

```
$ fxt_print -o -f /tmp/prof_file_something
```

Timings are in nanoseconds (while timings as seen in ViTE are in milliseconds).

Part I

Appendix

Chapter 5

The GNU Free Documentation License

Version 1.3, 3 November 2008

1. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of ``copyleft'', which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

2. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The ``Document'', below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as ``you''. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

   A ``Modified Version'' of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A ``Secondary Section'' is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The ``Invariant Sections'' are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The ``Cover Texts'' are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A ``Transparent'' copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not `Transparent'' is called`Opaque''.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The ``Title Page'' means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, ``Title Page'' means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The ``publisher'' means any person or entity that distributes copies of the Document to the public.

A section ``Entitled XYZ'' means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as `Acknowledgements'', Dedications'', `Endorsements'', or`History''.) To ``Preserve the Title'' of such a section when you modify the Document means that it remains a section ``Entitled XYZ'' according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

3. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

4. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a

computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

(a) Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

(b) List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

(c) State on the Title page the name of the publisher of the Modified Version, as the publisher.

(d) Preserve all the copyright notices of the Document.

(e) Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

(f) Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

(g) Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

(h) Include an unaltered copy of this License.

(i) Preserve the section Entitled ``History'', Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled ``History'' in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

(j) Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the ``History'' section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

(k) For any section Entitled `Acknowledgements'' or`Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

(l) Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

(m) Delete any section Entitled ``Endorsements''. Such a section may not be included in the Modified Version.

(n) Do not retitle any existing section to be Entitled ``Endorsements'' or to conflict in title with any Invariant Section.

(o) Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled ``Endorsements'', provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-↩ Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

6. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled ``History'' in the various original documents, forming one section Entitled `History''; likewise combine any sections EntitledAcknowledgements", and any sections Entitled `Dedications''. You must delete all sections EntitledEndorsements."

7. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

8. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an ``aggregate'' if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled `Acknowledgements'', Dedications", or ``History'', the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

10. TERMINATION

    You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

    However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

    Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

    Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

11. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License `‘or any later version’’ applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

12. RELICENSING

    `Massive Multiauthor Collaboration Site'' (orMMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A `Massive Multiauthor Collaboration'' (orMMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    `‘CC-BY-SA’’ means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    `‘Incorporate’’ means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is `‘eligible for relicensing’’ if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

5.1 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (C) *year your name*. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled `‘GNU Free Documentation License’’.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the `‘with...Texts.’’ line with this:

with the Invariant Sections being *list their titles*, with the Front-Cover Texts being *list*, and with the Back-Cover Texts being *list*.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.
If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.