Decentralized Online Scheduling Of Malleable NP-hard Jobs

Euro-Par 2022 Software Artifact Overview Document

Peter Sanders and Dominik Schreiber

This document serves as a documentation on the software artifact associated with our Euro-Par 2022 publication (named as above). In particular, we describe how to build and run our scheduling platform Mallob and how to reproduce our experiments. As our research was conducted on a considerable portion (1536–6144 cores) of a supercomputer with a total run time of around 14 h, we describe both our original setup as well as a small setup which only uses a single machine with 64 hardware threads and which runs within a reduced time frame (< 2.5 h in total). As such, much simpler and less costly experiments can be run while reproducing several of our results on a qualitative level.

Commands and Scripts

For your convenience, all commands featured in this document can be found in raw text form in `doc/document-commands.txt`. Where appropriate, we provide fully automated scripts which are then highlighted in boxes like this. In particular, the scripts which run individual experiments are meant to be well comprehensible and, if desired, provide additional insights.

1 Getting Started

We now describe how to to build and initialize our scheduling platform named Mallob. Throughout the document, we assume basic familiarity with the command-line interface of Linux (specifically `bash`).

1.1 Prerequisites

On an Ubuntu 20.04 system, the following packages with their dependencies are sufficient to build and execute Mallob and to run our scripts (except for producing plots, see Section 1.5).

```
sudo apt install git cmake zlib1g-dev libopenmpi-dev unzip xz-utils ↩
  build-essential cmake wget gdb gawk
```

1.2 Building Mallob

Mallob is a C++17 application linked with MPI. Only Linux on `x86_64` architectures is supported. To build Mallob, follow these steps from the base directory of this artifact:

1. Build `jemalloc` (scalable memory allocator):
   `( cd lib && bash fetch_and_build_jemalloc.sh )`
2. Build the external SAT solvers Mallob uses in our experiments:
   `( cd lib && bash fetch_and_build_sat_solvers.sh )`
3. Create a build directory: `mkdir -p build; cd build`
4. Generate build files with CMake:

```
CC=$(which mpicc) CXX=$(which mpicxx) cmake -DCMAKE_BUILD_TYPE=RELEASE ↩
  -DMALLOB_USE_JEMALLOC=1 -DMALLOB_LOG_VERBOSITY=4 -DMALLOB_ASSERT=1 ↩
  -DMALLOB_JEMALLOC_DIR=lib/jemalloc-5.2.1/lib ..
```

5. Build Mallob: `make; cd ..`

1.3 Fetching Benchmarks

In our experiments, we use benchmarks from the International SAT Competition 2020. You can fetch them with the following command:

```
( cd instances && bash fetch_sat2020_instances.sh )
```

Please consider that all instances together take about 33 GiB of disk space.
If, for some reason, you are not able to download these benchmarks, we included the 80 smallest instances among them in our artifact. To use them instead of the full set of benchmarks, replace the content of the files `templates/selection_isc2020.txt` and `templates/selection_isc2020_384.txt` with the content of the file `templates/selection_isc2020_smallest_included.txt`. This modification **will** change the outcome of our experiments, as this renders jobs simpler to solve on average.

1.4 Test Run

All Mallob runs are run from the base directory of Mallob (i.e., the executable is at `./build/mallob`). Make sure that `./reports/` and `./templates/` are valid paths as well. To test that everything functions correctly, perform this basic "sanity check" for your Mallob environment:

```
PATH="build:$PATH RDMAV_FORK_SAFE=1" mpirun -np 4 build/mallob -T=60 ↩
  -mono=instances/r3unknown_100k.cnf
```

For all runs of Mallob we set two environment variables: `PATH="build:$PATH"`, which assures that Mallob can find the executable for the solver sub-process, and `RDMAV_FORK_SAFE=1`, which can be necessary for some MPI setups to allow spawning (non-MPI) sub-processes without memory corruption[1]. The above command should process a single SAT formula with four MPI processes (i.e., four workers) for 60 seconds and then exit unsuccessfully. (Exiting with a solution would be a very lucky outcome.) You can compare the output with the sample output we provide in `sample-output/testrun.txt`.

1.5 On Producing Plots

We use PyPlot / Matplotlib for outputting plots from experiments. In order to produce plots from our experiments via the scripts we provide, you require Python 3 and a functional Matplotlib installation. It should suffice to install the following package with its required dependencies:

```
sudo apt install python3-matplotlib
```

As compute servers oftentimes do not have access to the graphical packages which Matplotlib requires, we facilitate the process of extracting data from the large run logs on the compute server and then transferring the data to a computer with a graphical environment in order to plot them. For a log directory d in `logs/`, we provide scripts (`reports/report-*.sh`) which create files in `logs/`d`/data` and then attempt to plot the data. On a server without graphical capabilities, this attempt will fail, but only after all data has been written to `logs/`d`/data`. Copy these sub-directories to the same relative path of the artifact project on your machine with plotting capabilities. You can then plot the data by calling the respective scripts `reports/plot-*.sh` with the same arguments as how you called `reports/report-*.sh`.

[1] `https://www.rdmamojo.com/2012/05/24/ibv_fork_init`

We use a custom Python script for plotting data. You find how it is called in the files `reports/plot-*.sh`. You can modify each call to this Python script to your liking. Most importantly, you can set the visualized bounding box of the plot with the options `-xmin=/-xmax=/-ymin=/-ymax=`, and you can output the plot into a PDF file using `-o=path/to/output.pdf`.

2 Instructions for Experiments

In the following, we explain how each of our experiments needs to be set up and how results can be retrieved. For each set of experiments, we provide two different sets of instructions: How to run our original experiments ("Original Setup"), and how to run a shorter suite of experiments on a single machine ("Small Setup"). We first describe these two setups in general and then explain how each of our evaluation sections can be reproduced with both setups. For our original setup we refer to our original paper for reference results. For our small setup we provide reference results in `sample-output/`.

2.1 Original Setup

For documentation purposes, we describe the exact setup we have used for our experiments. We used up to 128 "thin" compute nodes of SuperMUC-ng[2]. Each node consists of two Intel Xeon Platinum 8174 processors with 24 physical cores (48 hardware threads) each. Each node features a total of 96 GB of RAM. To quote the official documentation of SuperMUC-ng: "*The internal interconnect is a fast OmniPath network with 100 Gbit/s. The compute nodes are bundled into 8 domains (islands). Within one island, the OmniPath network topology is a 'fat tree' for highly efficient communication. The OmniPath connection between the islands is pruned (pruning factor 1:4).*"[3]
In addition to the base installation on SuperMUC-ng, we have loaded the following modules (for building as well as execution):

```
1) admin/1.0     5) intel/19.0.5      9) gcc/9.3.0
2) tempdir/1.0   6) intel-mkl/2019   10) intel-mpi/2019-gcc
3) lrz/1.0       7) lrztools/2.0     11) cmake/3.14.5
4) spack/21.1.1  8) slurm_setup/1.0  12) gdb/9.1
```

SuperMUC-ng runs the SLURM workload manager, which is a centralized scheduling system. Jobs in SLURM are described via a script file with special *SLURM directives* which describe the kind of deployment, the scale, maximum processing time, and other meta data. Jobs are then submitted to SLURM via the command `sbatch $file`. Each of our runs of Mallob is deployed as one particular SLURM job: In `sbatch/` we provide SLURM script files equivalent to the ones we have used to submit our runs. The SLURM directives for our experiments on SuperMUC-ng look like this:

```
#SBATCH -t 00:06:00
#SBATCH -nodes=128
#SBATCH -ntasks-per-node=12
#SBATCH -cpus-per-task=8
#SBATCH -ntasks-per-core=2
```

The first two directives specify the maximum run time and the number of nodes (machines) of the experiment. The subsequent directives describe how we subdivide each node into a number of MPI processes ("tasks") with four cores, i.e., eight hardware threads, each. The option `--ntasks-per-core` is named confusingly: It specifies to use hardware threads instead of cores (i.e., use eight hardware threads, not eight cores, per process). Our `sbatch` scripts load the environment modules listed above, create logging directories in advance (since this is more efficient than having all MPI processes create their own log directory at the same time), and execute Mallob as follows:

[2] `https://doku.lrz.de/display/PUBLIC/Hardware+of+SuperMUC-NG`
[3] `https://doku.lrz.de/display/PUBLIC/SuperMUC-NG`

```
PATH="build:$PATH" RDMAV_FORK_SAFE=1 srun -n $SLURM_NTASKS build/mallob $options
```

Depending on the particular system available to you, **it is likely that you need to modify or extend the `sbatch` files**. For instance, you may arrive at a different number of MPI processes per node (twelve in our case) depending on the cores available per machine. Other differences may include how the call to MPI looks like (`mpirun` vs. `srun` vs. `mpiexec.hydra` vs. ...) and how exactly you need to provide binding/mapping information. As we are unable to account for all possible cluster configurations, please consult the documentation for your particular compute cluster to find which exact SLURM configuration is required for hybrid (MPI + multithreading) programs.

2.2 Small Setup

If you intend to run Mallob on a single shared-memory machine with interactive access, you can call `mpirun` or `mpiexec` directly instead of the above procedure involving SLURM. We provide our small-scale experiments with Open MPI in mind, since this is open source software which should be available to everyone. The SLURM configuration provided above roughly corresponds to this `mpirun` command:

```
mpirun -np $numprocesses -map-by numa:PE=4 -bind-to core $command
```

This setup assigns four physical cores (eight hardware threads) to each MPI process. To maximize scheduling effects on a small scale, we suggest to increase the number of MPI processes in order to observe more fine-grained scheduling: Use the directive `numa:PE=1` instead and set the program option `-t=1` in Mallob. As such you can quadruple the number of MPI processes while giving each solver process only one instead of four cores. Therefore, for a single machine with 32 cores and 64 hardware threads, we suggest to run the following command:

```
PATH="build:$PATH" RDMAV_FORK_SAFE=1 mpirun -np 32 -map-by numa:PE=1 -bind-to core ↩
  build/mallob -t=1 $options
```

For this specific "small setup", we provide scripts which run all experiments automatically – see the highlighted boxes in each subsection.
We tested all "small setup" experiments on a machine with four 8-core Intel Xeon E5-4640 CPUs clocked at 2.4 GHz and with 512 GB of RAM. The machine ran Ubuntu 20.04 with Linux kernel 5.4, and we used Open MPI 4.0.3 and compiled with GCC 9.3.0.

2.3 Uniform Jobs

This set of experiments corresponds to Section 5.1 of our paper. We configure a subset of MPI processes ("clients") to introduce jobs to the system such that each client has k active jobs in the system at any point in time. All processes (including clients) participate in job scheduling and processing.

2.3.1 Original Setup

Commands for Original Setup

```
for f in sbatch/uniform-*.sh; do sbatch $f; done  # wait until finished!
for d in logs/uniform-{3,6,12,24,48,96,192,384,768}; do ↩
  reports/report-uniform-jobs.sh $d; done
```

This section involves **nine experiments** which take **six minutes each**. For each one, we allocate 128 machines configured as in Section 2.1. We use the following command-line arguments for Mallob:

```
-t=4 -q -c=$numclients -ajpc=$activejobsperclient -ljpc=$loadedjobsperclient ↩
  -T=320 -log=logs/uniform-$npar -v=4 -warmup -job-template=$jobtemplate
```

We instantiate a job template file (`$jobtemplate`) for a specific CPU limit as follows:

```
jobtemplate=templates/job-template-sat-r3unknown_100k-${coreminperjob}coremin.json
sed 's/CPUMINUTES/'$coreminperjob'/g' templates/job-template-sat-r3unknown_100k.json ↩
  > $jobtemplate
```

We define the shell variables referenced above as follows for the different runs:

$npar	$coreminperjob	$numclients	$activejobsperclient	$loadedjobsperclient
3	640	1	3	6
6	320	2	3	6
12	160	4	3	6
24	80	8	3	6
48	40	16	3	6
96	20	16	6	12
192	10	16	12	24
384	5	16	24	48
768	2.5	32	24	48

N.B.: The more jobs we process in parallel, the more PEs we configure to introduce jobs ("clients"). This is done in order to prevent a single client to become a bottleneck for introducing jobs. As such, we can focus on the decentralized scheduling capabilities of our system.
Please refer to the shell script `reports/report-uniform-jobs.sh` for precise information on how we extract and compute statistics from the runs. Running the script as indicated above, you should obtain a number of measures which correspond to the ones reported in Table 1 of our paper. The different efficiencies reported ($\theta/\theta_{\text{opt}}$, η, and u) should be close to one (>97%) for all experiments. If $\theta/\theta_{\text{opt}}$ varies considerably over the different runs, we expect that the lowest values are reported for the two extreme cases and that better values are reported in between.

2.3.2 Short / Small Setup

Commands for Small Setup

```
scripts/run/reproduce-small-uniform.sh   # runs for 36 min
for d in logs/uniform-{1,2,4,8,16,32}; do ↩
  reports/report-uniform-jobs.sh $d; done
```

For our "Small Setup" (Section 2.2) with 32 cores, we suggest to divide the CPU time allotted to each job by 6144/32 = **192**. Due to these changes we suggest to adjust the runs as provided in the below table. (Note that n_{par} must not exceed the number of MPI processes for your particular setup.)
We expect the same qualitative results on such a smaller scale. To compare your output with the output we observed, see `sample-output/uniform-jobs/report.txt`.

$npar	$coreminperjob	$numclients	$activejobsperclient	$loadedjobsperclient
1	10	1	1	6
2	5	1	2	6
4	2.5	1	4	8
8	1.25	1	8	16
16	0.625	2	8	16
32	0.3125	2	16	32

2.4 Impact of Priorities

This experiment corresponds to Section 5.2 of our paper. We configure a number of clients each of which is associated with a certain priority and introduces a stream of jobs of this priority. Again, all

MPI processes participate in job scheduling and processing, and our job scheduling enforces that jobs of higher priority receive more resources. We configure each stream of jobs to cycle through a random permutation of 80 carefully selected SAT instances (see our paper) and only use the first 80 jobs of each stream for analyzing its performance. As such, the system state remains constant throughout the entire run even if one of the streams finishes its batch of 80 instances before another stream.

2.4.1 Original Setup

Commands for Original Setup

```
sbatch sbatch/priorities.sh   # wait until finished!
reports/report-impact-of-priorities.sh logs/priorities
```

We use **32 machines** for **six hours** (see below for how to reduce execution time) with an otherwise identical configuration to the one from Section 2.3. We use the following command line arguments:

```
-t=4 -q -c=9 -ajpc=1 -jwl=300 -T=21500 -log=logs/priorities -v=4 -warmup -pls=0 ↩
  -sjd=1 -job-template=templates/job-template-priorities.json ↩
  -job-desc-template=templates/selection_isc2020.txt
```

The script `reports/report-impact-of-priorities.sh` should recognize automatically how many streams with which priorities have been used. The script outputs a plain text table and a plot, similar to Fig. 4 in our paper. Check that the mean assigned volume grows proportional to the assigned priority, except for rounding offsets. Response times should improve (i.e., decrease) for growing priorities.

2.4.2 Short / Small Setup

Commands for Small Setup

```
scripts/run/reproduce-small-priorities.sh   # runs for 1 h
reports/report-impact-of-priorities.sh logs/priorities
```

To reduce the run time of the experiment, we suggest to reduce the wallclock time limit per job from 300 s down to 60 s. To achieve this, we set `-jwl=60`. As such we can reduce the overall run time to one hour (`-T=3600`). Note, however, that this setup leads to less pronounced differences between the individual job streams' performances.
On 32 cores (Section 2.2), we suggest to reduce the number k of parallel streams to four in order to obtain meaningful differences in the jobs' volumes. For this means, set the Mallob command line option `-c=4`. For custom changes in the configuration, make sure that the scheduling algorithm is able to assign different volumes to the jobs (e.g., on two processes each job would receive a single worker no matter their priorities).
In general, to adjust the number k of streams and the values of priorities, edit the files `templates/job-template-priorities.json.$i`, where $i \in \{0, \dots, k-1\}$, and replace the value under the key `"priority"`, and set k as above.
You can find our results in `sample-output/priorities`. Mean volumes should grow proportionally with respect to job priority except for rounding, just as in the original setup. Also, response times should decrease for higher priorities, but perhaps only by a marginal amount. (Note that response times are in particular influenced by the number of solved jobs, which is quite low on such a small scale with only a minute of time per instance.)

2.5 Realistic Job Arrivals

This set of experiments corresponds to Section 5.3 from our paper. We configure a number of clients which introduce jobs from a wide variety of SAT problems at poisson-distributed arrival rates and random priorities and budgets.

2.5.1 Original Setup

Commands for Original Setup

```
for f in sbatch/realistic-*.sh; do sbatch $f; done   # wait until finished!
reports/report-active-jobs.sh logs/realistic-{2,1,3}
reports/report-utilization.sh logs/realistic-1
reports/report-latencies.sh logs/realistic-{1,4,5,6}
reports/report-worker-reuse.sh logs/realistic-{7,8,1}
```

This section features **eight experiments** which run for **one hour each**. We use the same SLURM directives as in Section 2.3 except for the time limit (`01:10:00`). We use the following base options for Mallob:

```
-t=4 -q -c=4 -ajpc=384 -ljpc=4 -T=3600 -log=logs/realistic-$rno ↩
  -v=4 -warmup -satsolver=kclkclcl -pls=0 -sjd=1 -ba=8 ↩
  -job-template=instances/job-template-priorities.json ↩
  -job-desc-template=instances/selection_isc2020_384.txt ↩
  -client-template=instances/$clienttemplate
```

With the following additional configuration:

`$rno`	Run description	`$clienttemplate`	Additional options
1	**Default**, $1/\lambda = 5\,s$	client-template.json	`-dc=0 -huca=0 -rs=1`
2	$1/\lambda = 2.5\,s$	client-template-doublejobs.json	`-dc=0 -huca=0 -rs=1`
3	$1/\lambda = 10\,s$	client-template-halfjobs.json	`-dc=0 -huca=0 -rs=1`
4	$h = 10$	client-template.json	`-dc=0 -huca=10 -rs=1`
5	$h = 100$	client-template.json	`-dc=0 -huca=100 -rs=1`
6	$h = \infty$	client-template.json	`-dc=0 -huca=-1 -rs=1`
7	No worker reuse	client-template.json	`-dc=0 -huca=0 -rs=0`
8	Basic worker reuse	client-template.json	`-dc=1 -huca=0 -rs=0`

To output plots as in Fig. 5 in our paper, use these commands:

```
bash reports/report-active-jobs.sh logs/realistic-{2,1,3}
bash reports/report-utilization.sh logs/realistic-1
```

In the first plot, the number of active jobs should fluctuate considerably over time and should be highest for run 2 and lowest for run 3. The utilization plot should reveal that consistently high utilization is achieved, namely 99% or more. If utilization drops at certain points, this may mean that the present jobs in the system are not capable of employing all present PEs ($\sum_j d_j < m$). In that case, the number of active jobs at that point in time should be very low.
Similarly, to reproduce Fig. 6 from our paper, execute:

```
bash reports/report-latencies.sh logs/realistic-{1,4,5,6}
```

For tree growth and initial scheduling latencies, we expect consistently low latencies for run 1 ($<$ 10 ms almost always) and higher worst-case latencies the higher h is chosen. Our script also outputs the balancing latencies (not included in the paper) which should be around the same for all runs.
And last but not least, to reproduce Table 2 from our paper, execute:

```
bash reports/report-worker-reuse.sh logs/realistic-{7,8,1}
```

Run 1 ("Ours") should generally lead to the best performance, although the margin in terms of mean reponse times and processed jobs may be small or non-existent. An improvement in terms of Context Creation Ratio (CCR) and context switches (CS) should be noticeable. The final five values should be monotonically increasing and the final value should be very close to 1 for each run.

2.5.2 Short / Small Setup

Commands for Small Setup

```
scripts/run/reproduce-small-realistic.sh   # runs for 50 min
reports/report-active-jobs.sh logs/realistic-{2,1,3}
reports/report-utilization.sh logs/realistic-{1,2}
reports/report-latencies.sh logs/realistic-1
reports/report-worker-reuse.sh logs/realistic-{7,8,1}
```

For shorter runs, we suggest to simply reduce the duration of each run to ten minutes by setting `-T=600`.
For small-scale runs (Section 2.2), we prepend "`small-`" to each value of `$clienttemplate` to use a configuration with a reduced number and size of jobs. In particular, we reduced the arrival rates to $1/\lambda \in \{3.\overline{3}\,s, 6.\overline{6}\,s, 13.\overline{3}\,s\}$, reduced the mean wallclock time limit per job from 300 s to 30 s, and slightly reduced the size of individual bursts of jobs.
We suggest to **omit runs 4, 5, and 6** because the different behavior of these runs in terms of latencies unfortunately cannot be reproduced on such a small scale. The script we provide will skip these runs per default; you can force to run and display them by calling the reproduce script with option `--full` and calling the `report-latencies` script with options `logs/realistic-{1,4,5,6}`.
Overall, results will look differently on this small scale because individual variations of single jobs have a much higher impact than in our large setup where the Law of Large Numbers is in effect. Please compare your output to the sample output we provide in `sample-output/realistic/`.

- In terms of active jobs over time, you should be able to reproduce the same qualitative results as in our distributed setup, albeit with higher fluctuation due to fewer jobs.
- You should be able to observe consistent reasonably high utilization ($\approx 99\%$) at least for run 2. The relatively higher fluctuation on this small scale leads to lower utilization on average due to frequent re-scheduling, and, especially for low arrival rates, may lead to time intervals where not all PEs can be employed (see Section 2.5.1).
- The latencies reported by the respective plots for run 1 should be similarly low as in our distributed setup or lower.
- In terms of worker reuse, you should be able to discern improvements in terms of CCR and CS for our approach over the other two approaches. The absolute values for CCR and CS should be lower (since a single job experiences less relative rescheduling than in a large system) whereas we expect the values for $\Pr[\cdot]$ to increase more quickly than in our paper – in our own small scale experiments, no worker was rescheduled five or more times.

3 Troubleshooting

3.1 Known Issues

During the experiments you may encounter errors being reported. There are two known issues with the provided version of Mallob:

- Due to a race condition, an internal assumption may fail in the solver process, which leads to this process aborting. As we have included some degree of fault-tolerance on this level, the solver process will be restarted and the scheduling of the system is not affected. This error occurs rarely (approximately once every 1 000–10 000 core-h), hence the impact on response times is negligible.
- A memory error may occur during the transfer of a large job description. This may lead to the job description being corrupted. The solver process recognizes this inconsistency and crashes. Unfortunately, as there is no fallback protocol for corrupted job descriptions, the solver process will crash repeatedly as attempts are made to restart it. In our experience, this error occurs very rarely (approximately once every 10 000 core-h) which also renders the error negligible.

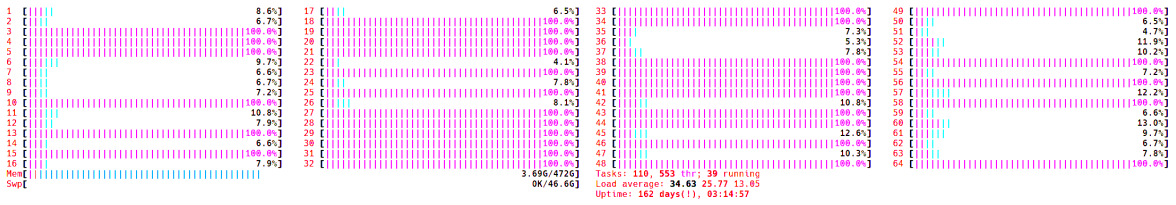
These errors have been fixed in more recent versions of Mallob. However, as the fixes for these problems are non-trivial and change (improve) the performance of our system, we decided to rather provide the exact version of Mallob our experiments were conducted with. Errors in Mallob generate files `mallob_thread_trace_*` in the base directory. As long as a run finishes successfully, these files can be removed afterwards; however, they may provide some insights if an unexpected error occurs.

3.2 Performance Problems

If you encounter worse performance than expected in some of the experiments, please search the log files for the text "`[WARN] Watchdog: No reset`". If this line occurs frequently and the indicated number of milliseconds for which no reset occurred exceeds 100 ms, then your setup is likely to suffer from a performance problem. Check the following points:

- The machine is not oversubscribed and the binding to cores is correct. In general, each MPI process should have access to $2n$ hardware threads if the option `-t=`n is set.
- The file system where the log directory resides allows for sufficiently fast logging. If possible, use a local file system or a file system with high bandwidth for your log directory. Try to reduce Mallob's verbosity (e.g., `-v=3` or `-v=2`) to probe whether this might be a problem.
- The machine is not running demanding processes apart from Mallob. Administration and monitoring tasks such as `ssh`, `htop`, or `tail -f` are usually fine, but computationally heavy tasks which make full use of one or multiple hardware threads are not.

If you check your local system's utilization while running Mallob, e.g., using `htop`, it should look something like this (coloring changed for better visibility):



Note that for $1 \leq c < 32$, hardware thread c is "almost idle" while hardware thread $c + 32$ is fully utilized, or vice versa. The only exception here is $c = 32$ where both hardware threads are fully utilized; this is a "client" PE which currently parses a job description (SAT formula).

4 Closing Notes

Our software named Mallob can be accessed at `https://github.com/domschrei/mallob`. Please see the license information provided there for using the different modules of Mallob. If any questions about the evaluation process arise, please contact `dominik.schreiber@kit.edu`. In case this address is not active any longer, please try `mail@dominikschreiber.de`.