Assignment 4 - Profiler

Zirong Cai, Phuong Nguyen

19. Dezember 2021

1. Quicksort – Intel VTune Amplifier XE (2P)

a. Optimal final clause

Using Basic Hotspots, the two important matrics we can get are Elapsed Time and CPU time which are defined as:

- Elapsed Time: the wall clock time from start to end of the execution
- CPU time: the quantity of processor time taken by the processes.

As a end user, what we want is of course faster response time, and we don't care about CPU pressure. In this case, we want to get the sorted array as soon as possible, meaning lower elapsed time.
In the table 1, we listed the elapsed time of our quicksort application with respect to different thread numbers and different final-clauses. cond{i} correspond to:

$$CurrentArrayLength < DataSize/i, i = 2, 4, 8, 14, 28$$

We can see that the clauses that give us the lowest elapsed time are either under condition4 or condition5. We first assume that there's a best final clause between condition4 and condition5, we didn't find it because the granularity we use here is too large.

NUM_THREAD	cond2	cond4	cond8	cond14	cond28
2	0.791	0.920	0.670	**0.658**	0.876
4	0.583	0.600	0.511	0.916	**0.489**
8	0.791	0.503	0.679	0.656	**0.436**
14	0.638	0.752	0.648	0.661	**0.547**
28	0.694	1.043	0.954	**0.677**	0.901

Tabelle 1: Elapsed Time With Different final-clauses Using Datasize=2000000

So we reduce the granularity to 1 and examine all possibilities. And we find that for different thread numbers, the final clause

$$CurrentArrayLength < 75000(\approx DataSize/26)$$

gives us the best performance.

b. Other Analysis types

- performance-snapshot: useful to get a quick snapshot of our application performance and identify next steps for deeper analysis.
- memory-consumption: Analyze memory consumption by our application, its distinct memory objects and their allocation stacks. Since our application is compute bound but not memory bound, this feature is not useful for us.

- microarchitecture: Analyze CPU microarchitecture bottlenecks affecting the performance of our application. Since our code only uses some compare and exchange operations, this is also not useful for our application.
- threading: Discover how well our application is using parallelism to take advantage of all available CPU cores. This could be useful for us to find the best final clause.
- io: Analyze utilization of IO subsystems, CPU, and processor buses. Not useful since our application don't have much IO.

c. Does the optimal final clause depend on the number of threads or the array length?

As discussed above, the optimal final clause dose not depend on the number of threads. To find out whether the optimal final clause depend on the array length or not, we fixed the number of threads to 14 and profile our allication with different data size, we then get the table 2 which gives us a similar result that the optimal final clause is around 26, not exactly 26, but near by 26, so we may conclude that the optimal final clause does also not depend on the data size.

Data Size	cond2	cond4	cond8	cond14	cond16	cond20	cond22	cond24	cond26	cond28
5000000	5.884	3.382	0.598	17.064	3.510	3.866	4.742	4.356	3.566	**0.538**
10000000	1.545	1.261	0.915	7.590	2.669	3.355	8.594	**0.833**	3.651	10.860

Tabelle 2: Elapsed Time With Different Final-clauses Under Different Data Size

d. Question

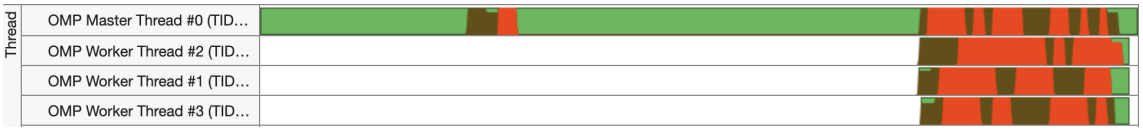


Abbildung 1: BottomUp with 8 threads and cond14 under Data Size=2000000

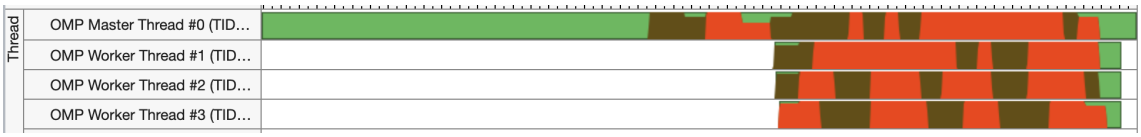


Abbildung 2: BottomUp with 8 threads and cond8 under Data Size=2000000

In figure 1, we saw that after the initialization of the data array, thread 0 start to execute the code of quicksort, but then the process stays idle for a quite long time, while under other conditions, e.g. in figure 2 there's no such idle time. I am quite confused since there are no I/O or barrier(well there does exist a task wait barrier but as far as i know, task wait won't block a thread) after the first execution.

2. CG – Scalasca (3P)

a. Automatic instrumentation

We profiled the CG implementation with 28 MPI processes, the output data was visualized with Cube and presented in Figure 3. The program spent 1274 second in the main computation which is 94.1% of the total runtime. As we can see in the third column, each rank spent roughly 45 seconds. This result looks pretty good in term of load balancing. From this figure, we also saw that the program took 1354 seconds in total but only 6.09 seconds for Computational Load Imbalance, which is 0.5% only. Figure 4 shows the detail of load imbalance in each rank. We can see that some ranks are overloaded while some other are underloaded, but the difference is not significant.

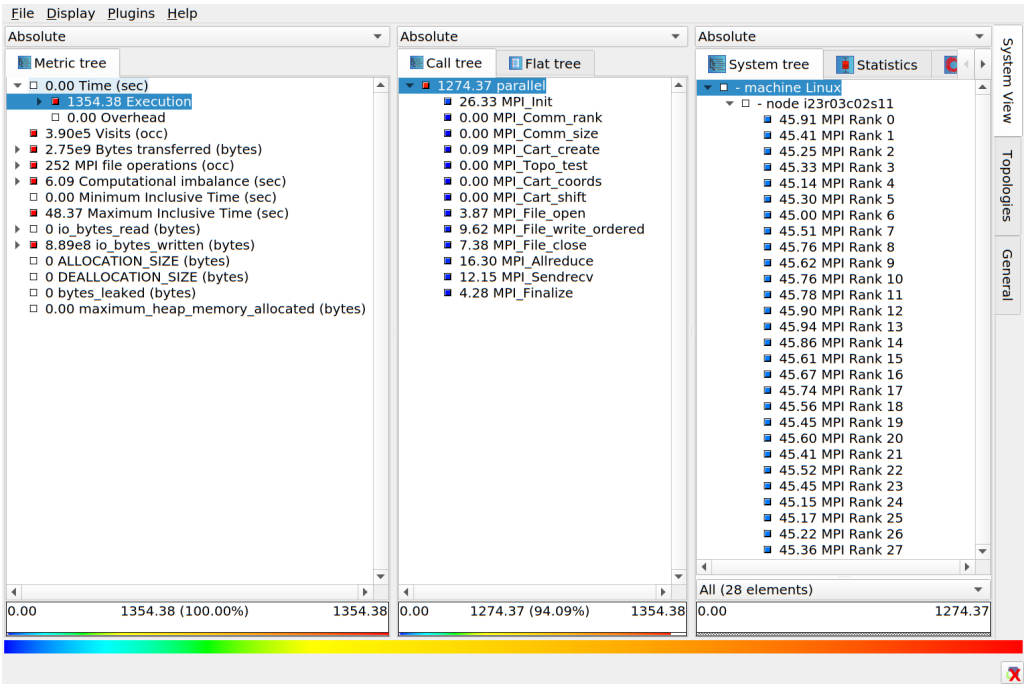


Abbildung 3: Profiling of CG, Computation Load

b. Manual instrumentation

For manual instrumentation, we decided to profile the *solver* function only since it's the heart of the program. We did it by defining a profiling region inside the function as describe below:

```
void solve(struct_param *param, std::size_t cg_max_iterations, double cg_eps)
{
  SCOREP_USER_REGION_DEFINE( handle )
  SCOREP_USER_REGION_BEGIN( handle, "solve", SCOREP_USER_REGION_TYPE_FUNCTION)

    // Do the actual computation
    ...

  SCOREP_USER_REGION_END( handle)
}
```

The profiling output is presented in the Figure 5. Compares to the previous profile output,

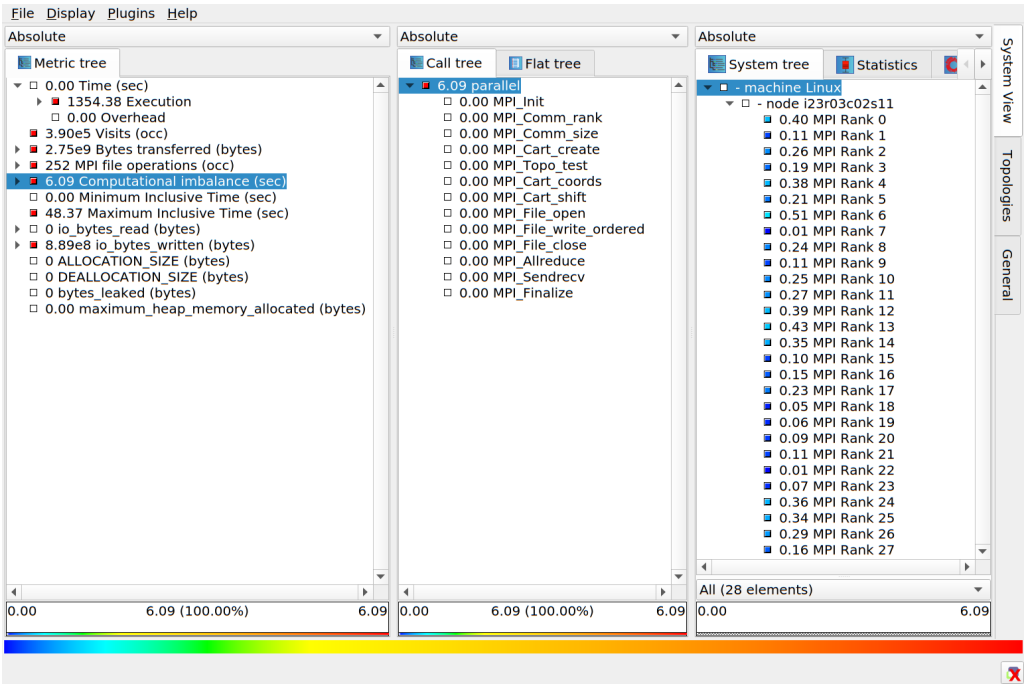


Abbildung 4: Profiling of CG, Load Imbalance

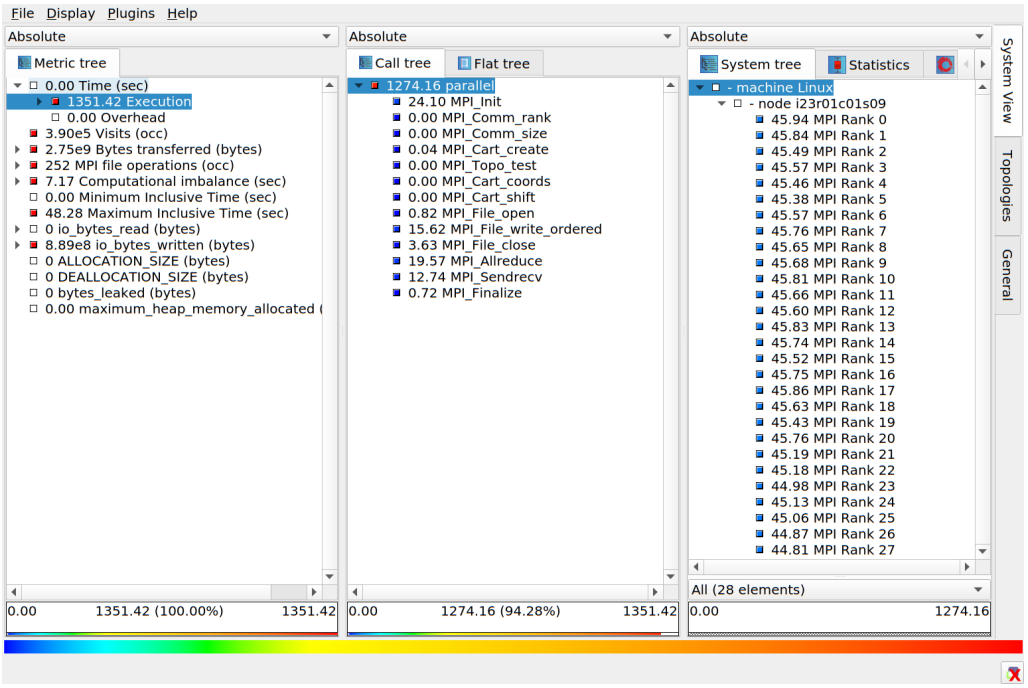


Abbildung 5: Manual profiling of solve function

there is no big difference. However, one can notice that the computational imbalance time is different, so is the total runtime. There is no much different since the program spent most of the time in this solver function, rather then others.

c. Hybrid Implementation of CG

Implementation We implemented OpenMP *parallel for* for loops in following functions: *g_copy*, *g_dot_product*, *g_scale*, *g_scale_add*, *g_product_operator*. Besides, to optimize for NUMA, we also implemented "First-touch"which is parallelizing the data initialization in *init_grid*, *init_b*, *g_copy*. In addition to the MPI code from the last worksheet, we have successfully implemented MPI File I/O for parallelized outputing.

Performance We tested out hybrid implementation with different configuration and number of threads per MPI. The best results are given when the number of OpenMP threads equal to 7, and the bind set are:

```
KMP_AFFINITY=granularity=core,compact,1,0
I_MPI_PIN_DOMAIN=omp
I_MPI_PIN_ORDER=compact
I_MPI_PIN_CELL=core
```

This seems reasonable since on our system each NUMA node has 7 cores. With this configuration, each MPI process has 7 threads in the team, binds to 7 cores in the same NUMA node which is good for data accessing.
The performance of this hybrid implemented is presented in the Figure 6, in comparison with the performance of pure MPI implementation. The performance of Hybrid approach is the similar to the pureMPI, even for the test on 4 nodes or 8 nodes. We would expect the Hybrid one has better performance since it can ultilize both advantages of shared memory and distributed memory models. To understand this, we are going to profile this code with Scalasca.

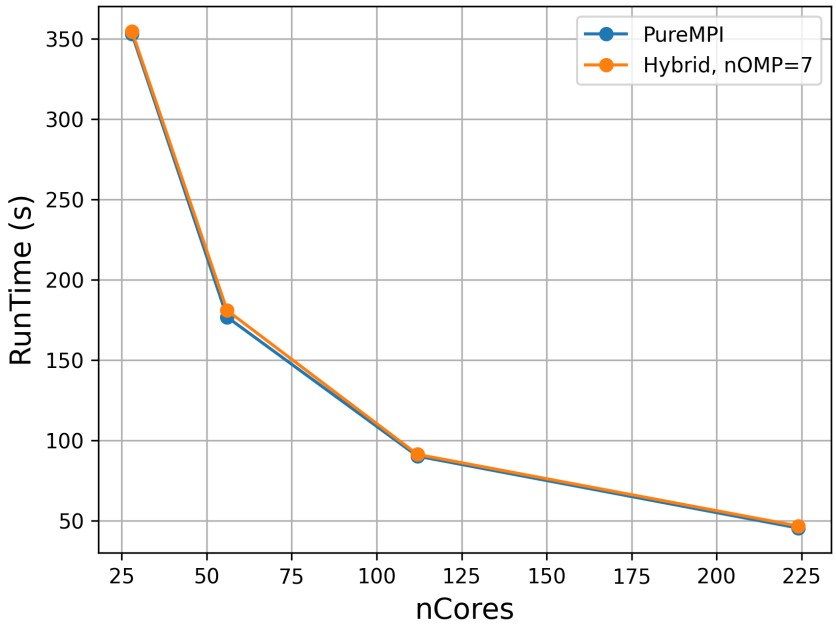


Abbildung 6: Performance Comparison between PureMPI and Hybrid Implementation

d. Profiling Hyrbid CG

With a first glance at the Figure 7, we can see that the structure of the hybrid profiling is quite different to what we had for pure MPI jobs. Besides the time spent for Computation and MPI, there are also time for OpenMP and for Idle threads. In computation, we can see that there is a big imbalance workload between master threads and child threads. For example, in the 0th rank, the master threads works for 61 seconds while its child threads only work for roughly 42 seconds.

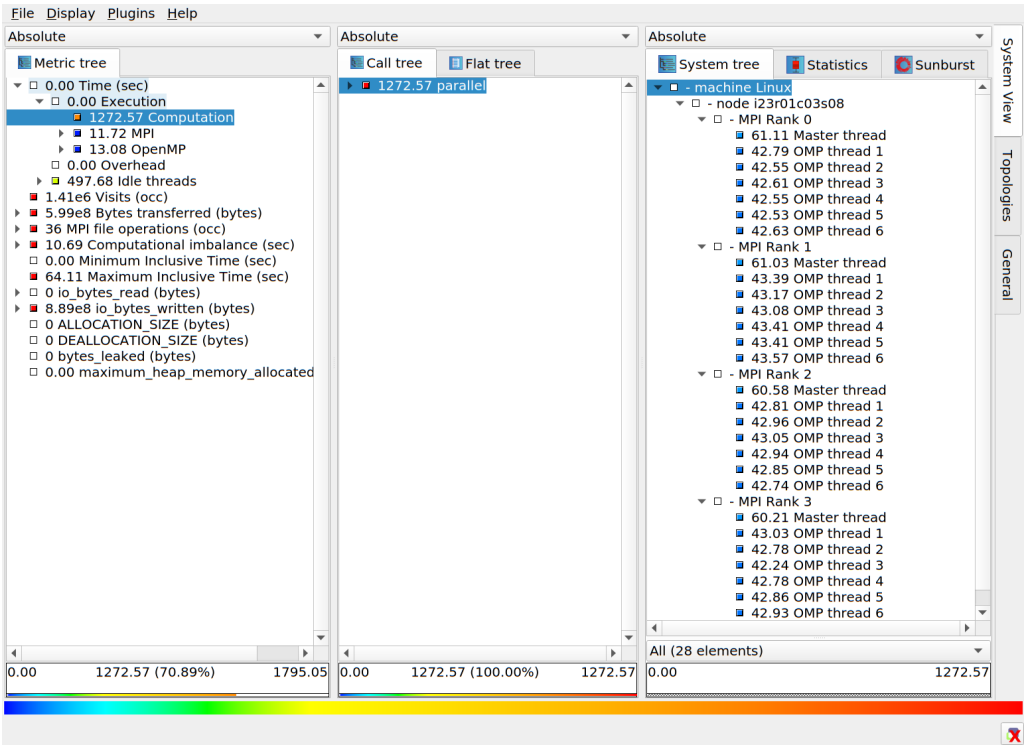


Abbildung 7: Manual Profiling of *solve()*, Computation Workload

At first we thought that this is a problem from load balance. But after have a look in more detail which is presented in the Figure 8, we see that in each *!$omp parallel* region, for example the one at *@general.cpp:254* (which is inside of *g_scale_add*), all threads have pretty equal work load and spent roughly the same time inside each function. It means the difference in runtime of master threads and child threads we observed before is due to the fact that the master threads are also the threads which handle the MPI calls, while other child threads - who only do work in parallel region - need to wait/be idle.

This can be proved by the Figure 9, where we can see that the idle time of child threads are all during the MPI calls.

e. Discussion

The Scalasca tool is very handful and simple to use, even with the Hybrid program. In our case, however, since all OpenMP load balance is pretty good and there is a very small portion of Computational Imbalance, we could not come up with any further possible optimisations based on this profile outputs, even though we expected hybrid would have a better performance then purely MPI but it did not happen.

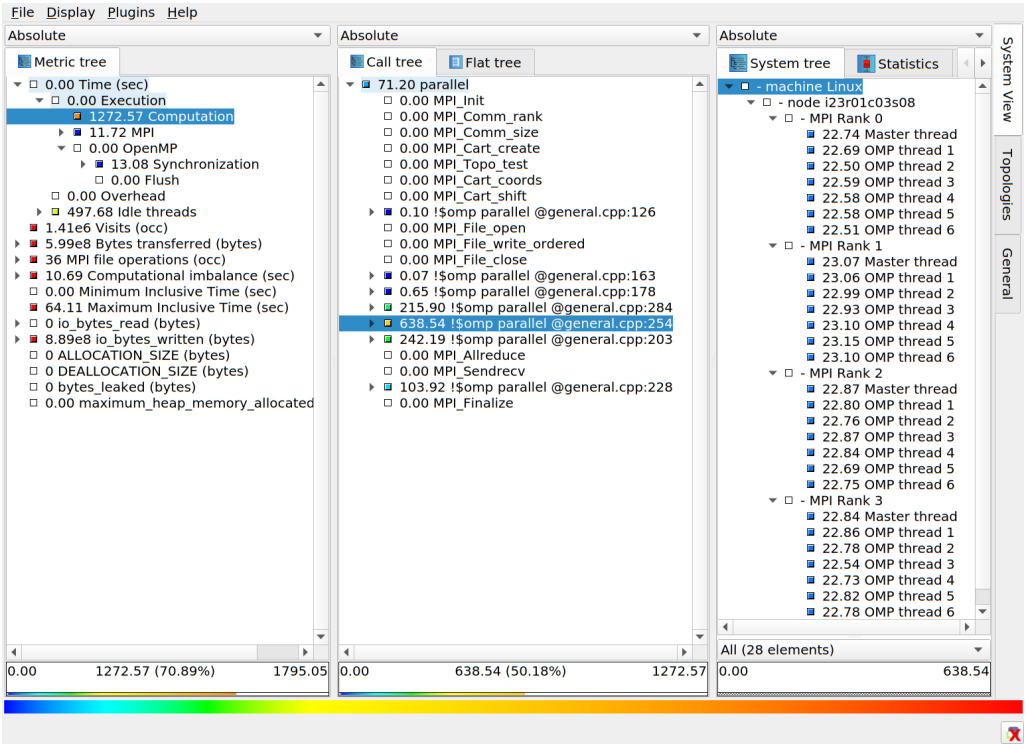


Abbildung 8: Manual Profiling of *solve()*, OpenMP Thread Load Balance

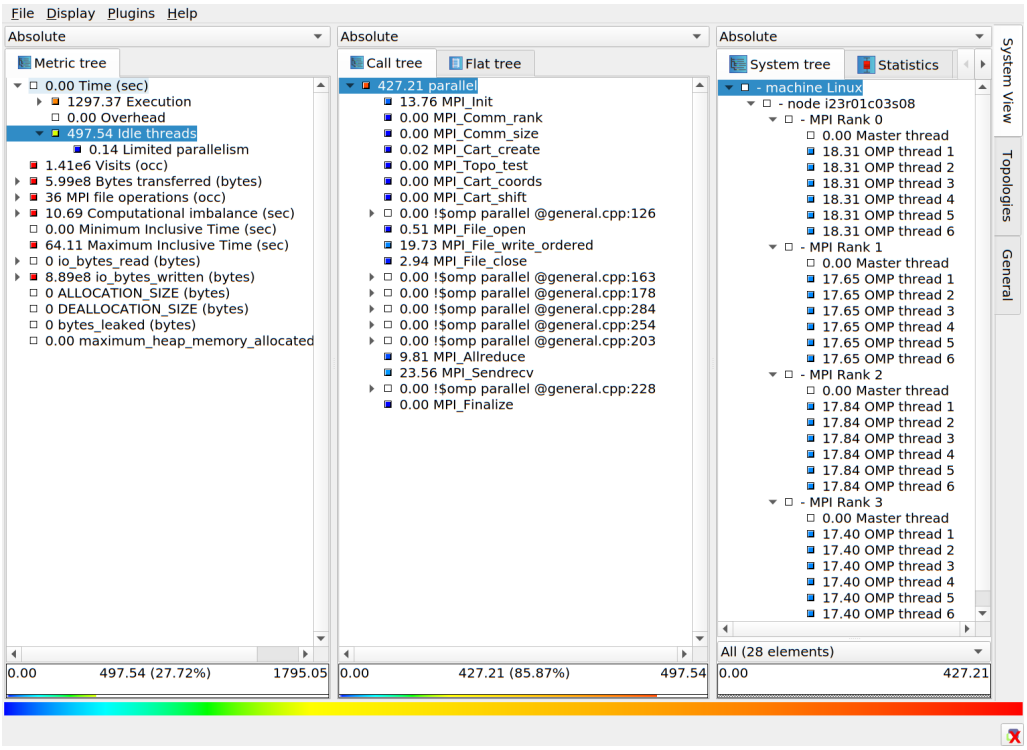


Abbildung 9: Manual Profiling of *solve()*, Thread Idle

3. Likwid-perfctr (3P)

1.a)

```
#ifdef LIKWID_PERFMON
#include <likwid-marker.h>
#else
#define LIKWID_MARKER_INIT
#define LIKWID_MARKER_THREADINIT
#define LIKWID_MARKER_SWITCH
#define LIKWID_MARKER_REGISTER(regionTag)
#define LIKWID_MARKER_START(regionTag)
#define LIKWID_MARKER_STOP(regionTag)
#define LIKWID_MARKER_CLOSE
#define LIKWID_MARKER_GET(regionTag, nevents, events, time, count)
#endif

void gravity(double dt, RigidBody* bodies, int T) {
  LIKWID_MARKER_START("gravity");
  for (int t = 0; t < T; ++t) {
    bodies->move(0.0, 0.0, 0.5 * 9.81 * dt * dt);
  }
  LIKWID_MARKER_STOP("gravity");
}

int main() {
    LIKWID_MARKER_INIT;
    ...
    LIKWID_MARKER_CLOSE;
    return 0;
}
```

The following command can be used to compile the code with LIKWID with gcc

```
$ gcc -DLIKWID_PERFMON -L$LIKWID_LIB -llikwid -I$LIKWID_INCLUDE
<SRC> -o <EXEC>
```

LIKWID_LIB and LIKWID_INCLUDE will be automatically set when we load LIKWID in our server(module load likwid).
To run the application, using the following command

```
$ likwid-perfctr -C process_ids -g event -m <EXEC>
```

b)

- FALSE_SHARE, not relevant since our application run with single core.
- TLB_INSTR L1 Instruction TLB miss rate/ratio, not relevant because our application is small.
- CACHES Cache bandwidth in MBytes/s, relevan to have a overview of cashes's information since our application is data hard related.
- UOPS_EXEC UOPs execution, not relevant since we don't have much branch in our code.
- UOPS_RETIRE UOPs retirement, not relevant, same as above.
- BRANCH Branch prediction miss rate/ratio, not relevant since no branch in our app.

- FLOPS_AVX Packed AVX MFLOP/s, not relevant, didn't use much FLOP in our app.
- L2CACHE L2 cache miss rate/ratio, relevant, to see if bodies is still in cache.
- L3 L3 cache bandwidth in MBytes/s, not quite relevant since L3 is the last layer and our app is small. L3¡-¿main memory is not so important for us.
- QPI QPI Link Layer data, not relevant, we didn't use the network.
- UOPS UOPs execution info, not relevant, same reason as in UOPS_EXEC.
- NUMA Local and remote memory accesses, not relevant, no remote memory used.
- ENERGY Power and Energy consumption, not relevant. our purpose is not energy saving.
- L3CACHE L3 cache miss rate/ratio, as the last layer, L3 may not be so relevant. same reason as above
- ICACHE Instruction cache miss rate/ratio, not relevant app is small.
- CBOX CBOX related data and metrics, may be relevant, but most likely not since our app is too small to use the last layer.
- CYCLE_ACTIVITY Cycle Activities, not relevant
- TLB_DATA L2 data TLB miss rate/ratio, relevant since our app is data hard related.
- MEM Main memory bandwidth in MBytes/s, not relevant, (result all 0?
- DATA Load to store ratio, relevant, help us to find out if store is the bottleneck or load.
- L2 L2 cache bandwidth in MBytes/s, relevant to see how many data is loaded and evicted.
- DIVIDE Divide unit information, not relevant since no divide in app.
- CYCLE_STALLS Cycle Activities (Stalls), useful for us to find out the bottleneck.
- CLOCK Power and Energy consumption, not relevant. our app is too small to measure the energy consumption.

c)

Metric	HWThread 0
Runtime (RDTSC) [s]	0.0024
Runtime unhalted [s]	0.0027
Clock [MHz]	2899.6409
CPI	13.4467
L2 request rate	1.4124
L2 miss rate	0.4627
L2 miss ratio	0.3276

We found that the cache miss rate of L2 is quite high because everytime we call the function, the rigidbody needs to be reloaded. Our idea is, instead of update all rigidbody together once in gravity, we can rewrite the gravity function so that it only update one rigidbody for a given time period.

```
void gravity(double dt, RigidBody* bodies, int T) {
    for (int t = 0; t < T; ++t) {
      bodies->move(0.0, 0.0, 0.5 * 9.81 * dt * dt);
    }
  }
int main() {
    ...
    for (int n = 0; n < N; ++n) {
        gravity(0.001, bodies[n], T);
    }
    ...
}
```

With this, we incresed the Bandwidth from 341.077062 to 1529.414464 and reduced the L2 miss rate from 0.4627 to 0.0042.

2.a) Assume we use 4 threads for our application, then the task distributed to each core is :

$$
\begin{aligned}
core0: &\quad (N + (N-1) + ... + N - N/4 + 1) = 9381250 FLOPS \\
core1: &\quad (N - N/4) + ... + (N - N/2 + 1) = 6255000 FLOPS \\
core2: &\quad (N - N/2) + ... + (N - 3 * N/4 + 1) = 3131250 FLOPS \\
core3: &\quad (N - 3 * N/4 + 1) + ... + 1 = 3750 FLOPS
\end{aligned}
$$

From the results we can easily see that there's a big gap of flops between different threads. The reason is that the task is upper-triangular matrix times vector multiplication. The default schedule of $\#pragma omp for$ is static, which means omp simply divide the task by rows with $rows/OMP\_NUM\_THREADS$. But the calculation for each row is different, the upper the row is, the more calculation it needs to be done. That is the reason why there's a load imbalance.

b)

```
+-----------------------+---------+-------------+-------------+
|         Event         | Counter | HWThread 0  | HWThread 1  |
+-----------------------+---------+-------------+-------------+
|    AVX_INSTS_CALC     |   PMC0  |   174701500 |   108035300 |
+-----------------------+---------+-------------+-------------+

+-----------------------+---------+-------------+-------------+-------------+
|         Event         | Counter | HWThread 0  | HWThread 1  | HWThread 2  |
|    AVX_INSTS_CALC     |   PMC0  |   165725700 |   283130100 |   170502800 |
+-----------------------+---------+-------------+-------------+-------------+
                                  +-------------+
                                  | HWThread 3  |
                                  |  57890290   |
                                  +-------------+
```

By running the application with FLOPS_AVX for different numbers of threads, we can see from the metrics AVX_INSTS_CALC that the task distributed to different core is quite different which means we have found a load balance here. But this metrics can not always find the load imbalance for us. Reason see in sub task d.

c) Since our application do not have much branches and thus there's no need to predict branch. So there won't be too much overhead spent on executing wrong instructions, which means INSTR_RETIRED_ANY can in some sense represent the work load for each task.

d)To solve this problem, the easiest way is using dynamic schedule of omp. We tried with $schedule(dynamic, 10)$ with 4 threads, and get the result as follows:

Event	Counter	HWThread 0	HWThread 1	HWThread 2	HWThread 3
INSTR_RETIRED_ANY	FIXC0	587691000	665369900	719993700	710925800
AVX_INSTS_CALC	PMC0	105834700	243815400	269732900	272558700

We can see now the flops of thread 1,2,3 are quite similar but not thread 0. But take a look at INSTR_RETIRED_ANY, we realize that the reason why thread 0 didn't do as much flops as other threads is that thread 0 is the master thread and need to also be responsible for scheduling stuff. So we can not say that there's a load imbalance here because the total instructions each core have done here is almost the same. In this case, Not only can't AVX_INSTS_CALC help us to find the load imbalance, it can even be misleading.

4. Reverse Engineering with Tracing (2P)

a. Description of tracing using Intel Trace Analyzer and Collector

Tracing is an application gives insights into the applications behavior by collecting timing information regarding function calls or expliclity identified code paths. This is generally done by calling two timestams: One before entering a function and one after exiting said function. The differences of these timestams are then collected and mapped to the applications call paths. In parallel applications, the respective times have also be mapped to process ids and threads, if applicable.

Intel Trace Analyzer and Collector provides the APIs, command line tools, as well as GUI-applications to first collect and later analyze and visualize these traces using the intel tools.

Usage of ITAC on CoolMUC2 To use ITAC on CoolMUC2 the following commands have to be entered:

```
module unload intel-mpi intel-mkl intel
module load intel-parallel-studio/2018
#everything is now in $INTEL_PARALLEL_STUDIO_BASE
. $INTEL_PARALLEL_STUDIO_BASE/bin/compilervars.sh intel64
. $INTEL_PARALLEL_STUDIO_BASE/vtune_amplifier/amplxe-vars.sh
. $INTEL_PARALLEL_STUDIO_BASE/itac_latest/bin/itacvars.sh
. $INTEL_PARALLEL_STUDIO_BASE_impi/2018.4.274/bin64/mpivars.sh
```

This makes the itac, vtune and compiler tools available.

Instrumentation using ITAC by commandline To compile for tracing, the *-trace* flag has to be set for the intel compiler.

```
mpiicc -trace cool_mpi_app.c -I$INTEL_PARALLEL_STUDIO_BASE/itac_latest/include
```

By giving command line arguments, such as *tcollect* all functions can be instrumented for collection automatically.

```
mpiicc -tcollect -trace cool_mpi_app.c \
-I$INTEL_PARALLEL_STUDIO_BASE/itac_latest/include
```

Additionally specific functions can be selected given by specifing the functions to be collected in a "filter file", in case not all functions are desired for collection. This is done by naming them in a file the filter file. E.g. for the the *cool_mpi_app.c*, a filter file *filterfile.flt* may look like:

```
do_something on
```

Where valid parameters for the functions are *on* of *off*. The specific functions are then automatically instrumented if set to on.

```
mpiicc -tcollect-filter filterfile.flt -trace cool_mpi_app.c \
-I$INTEL_PARALLEL_STUDIO_BASE/itac_latest/include
```

Instrumentation Using ITAC by API Alternatively event handler can be introduced by hand. (The example shown in the following is the as of *superusefulITAC.cpp*, where the manual tracing using the *Stopwatch.h* functonality is replaced by *VT* calls as described in the *ITAC* documentation.) This is done by defining a VT function:

```
static int _myregid
char *name="My region ID";
ierr=VT_funcdef(name, VT_NOCLASS, &_myregid);
assert(ierr == VT_OK);
```

This *_myregionid* handle can now be used to identify specific regions within the code using the functions *VT_begin(_myregionid)* and *VT_end(_myregionid)*. This is especially usefull when a region is not soley defined by a function or shall spann multiple functions. E.g.:

```
\\ Some work
VT_begin(_myregionid_handle)
for(int=0;i<5;i++){
function1();
function2();
function3();
}
VT_end(_myregionid_handle)
\\ Some further work
```

b. Parameter Analysis of *cool_mpi_app.exe*

In this task, we ran the *cool_mpi_app.exe* with 2 MPI processes. We tested different parameters and their influences to the total runtime, ratio between the time spent for the 'computation' and MPI rountines, and waiting time at the bariers.
From the Figure 10a to 10b, we can see that when double the fourth parameter, we got

the total runtime increase by 2, in which the time for application in the Rank 0 and the time for MPI in Rank 1 kept almost the same, but the time for MPI in rank 0 and the time for application in Rank are doubled, leading to the time for Rank 0 to wait at the barrier also doubled.
From the Figure 10a to 10c, we can see that when double the third parameter, we got the total runtime increase by 1.5. We noticed that the time for Application in both Ranks increased by 2 times in the 10c in comparison to in 10a, while the time for MPI moved toward to become equal. In this test, even though the computation load is not balanced between the two ranks, we have least waiting time at the barrier, only 0.3%.
From the Figure 10a to 10d, we can see that when double the second parameter, we got the total runtime increase by 2, in which the times in rank 0 increased by roughly 1.5, while for rank 1 the time for computation increases by 2 at the same time keeping constant time for MPI. The time spent at barrier for rank 0 increased roughly 25%.
The last parameter-set result in 10e has similar effect as in 10d.

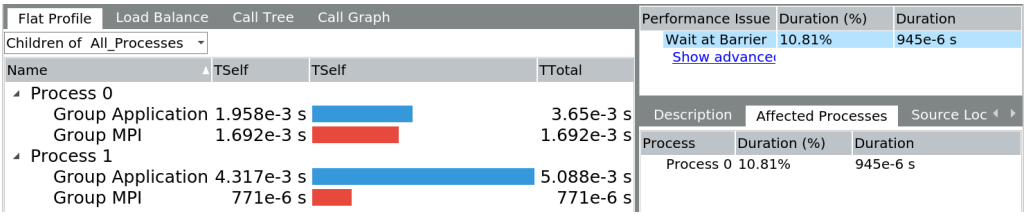
Besides, from the Trace analyzer, we also knew that in the black box binary, there is only *MPI_Allreduce* is used for communication. (Figure 11)).

c. Analysis of expert user's parameters

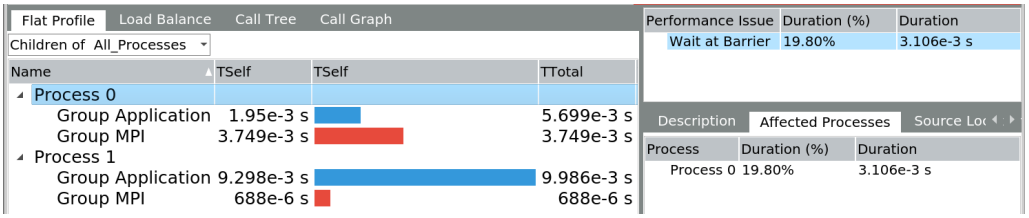
In this taks we have parameters given by an expert: They suggest $a = 10$, $b = 100$, and $c_1 = 14$,$c_2 = 14$, $c_3 = 14$, $c_4 = 14$. We run the experiment as suggested by the expert and present these in Figure 12. Figure 12 shows a total execution time of 28 seconds. The trace shows that ranks 0, 2 and 3 have substantial waiting times of 14 seconds. Rank 1 has no waiting time and their workload is double the workload of 0, 2 and 3. Therefore, we suggest an improved parameterset: The parameters 10 and 100 for a and b are fixed, similarly $c_1 + c_2 + c_3 + c_4 \stackrel{!}{=} 56$ is required. Therefore we suggest to half the time of rank 1 while minimizing the other parameters. Since $14/2 = 7$, but $(56 - 7)/3$ is not integer, we suggest $c_2 = 8$. This results in the parameters $c_1 = 16$, $c_2 = 8$, $c_3 = 16$, $c_3 = 16$, resulting in balanced parameters for ranks 0,2 and 3. We run this parameterset and evaluated as seen in Figure 13. Figure 13 shows a total execution time of 16 seconds. This is a speedup of 1.75. The processes show no waiting time and for parameters a=10 and b=100 this appears to be an optimal input parameter set from an MPI perspective. To verify this, an evaluation of reducing the paramter of c_2 further to $c_2 = 7$ is evaluated (Fig. 14a). Additionally it is evaluated if the parameter doesn't need that much reduction by setting it to $c_2 = 9$ (Fig. 14b).

Figure 14a shows that the paramters $c_2 = 7$ would improve the execution time, however due to the restriction of $c_1 + c_2 + c_3 + c_4 \stackrel{!}{=} 56$ the processes with parameter 17 are now the bottleneck. These ranks are busy, but their parameters can no be reduced further as the sum has to be 56, and at least one of them will increase when trying to reduce the remaining parameters. This inhibits to load balance this parameter suggestion further, which makes $c_2 < 8$ not practical.

Figure 14b inquires if the parameters c_2 could be increased and wether the paramter $c_2 = 8$ is choosen too small. By setting the parameter $c_2 = 9$ and choosing the other parameters accordingly ($c_1 = 17$, $c_3 = 15$, $c_4 = 15$), we can see that now rank 1 is busy while the other ranks are waiting for it. Therefore c_2 should be reduced furhter. With the considerations from both Figure 14a and 14b we see that the parameters from Figure 13 are already optimal.

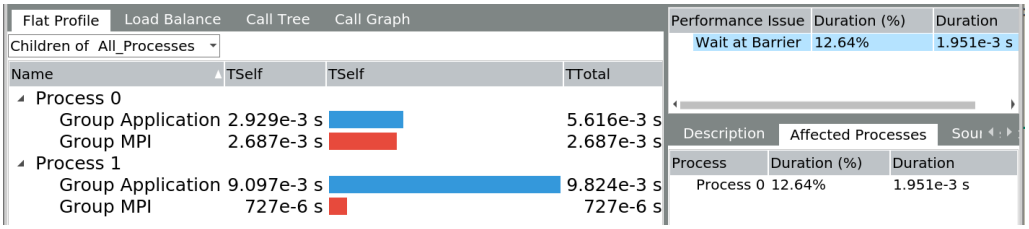


(a) Parameters=(1 1 1 1), Total runtime = 0.00874s

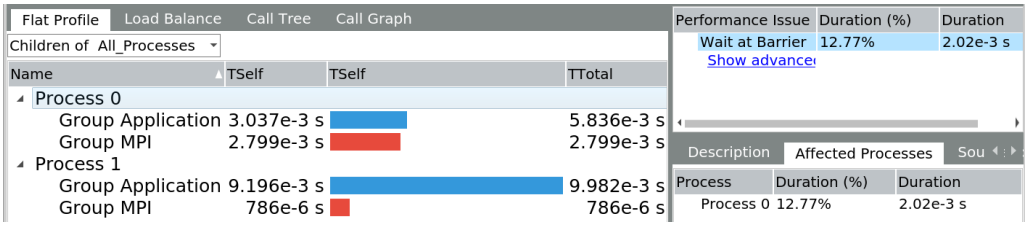


(b) Parameters=(1 1 1 2), Total runtime = 0.0157

(c) Parameters=(1 1 2 1), Total runtime = 0.012



(d) Parameters=(1 2 1 1), Total runtime = 0.0154



(e) Parameters=(2 1 1 1), Total runtime = 0.0158

Abbildung 10: Trace Analyzer of *cool_mpi_app.exe* with various parameters

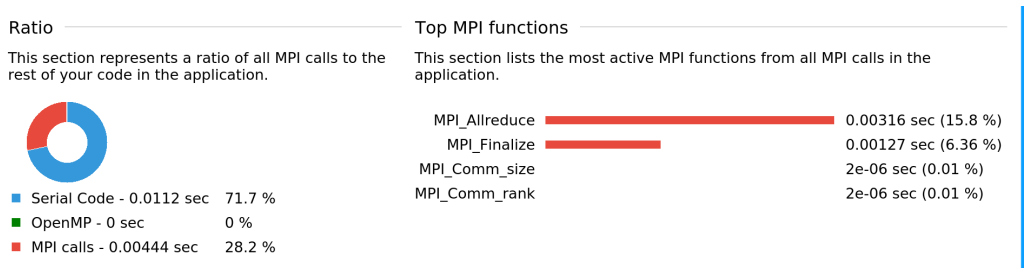


Abbildung 11: Trace Analyzer of *cool_mpi_app.exe* - MPI Functions

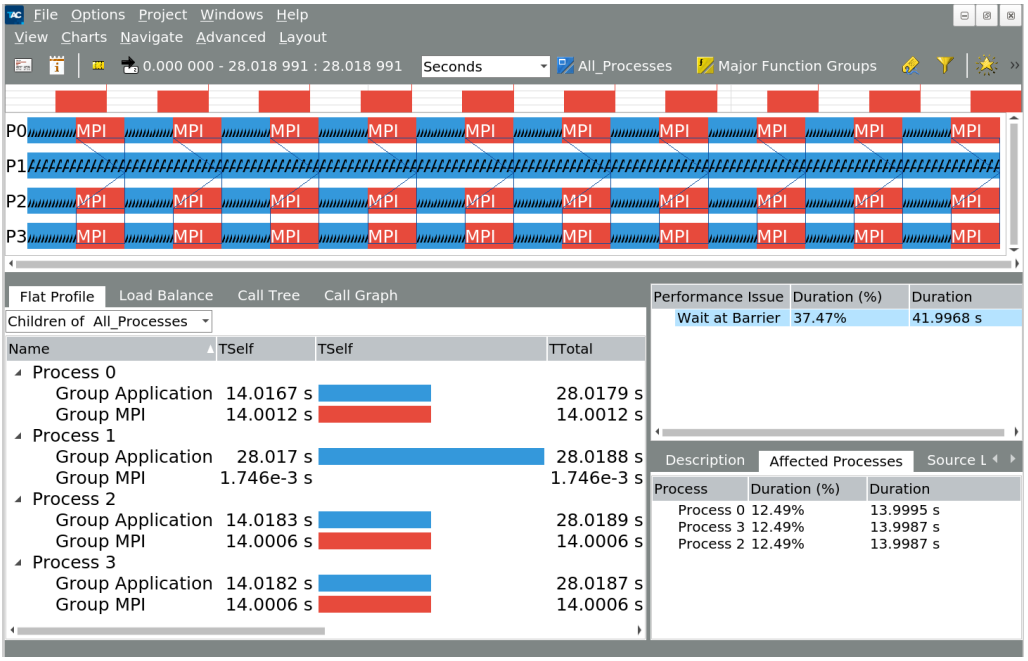


Abbildung 12: Trace Analyzer of *Expert parameters* - 14 14 14 14

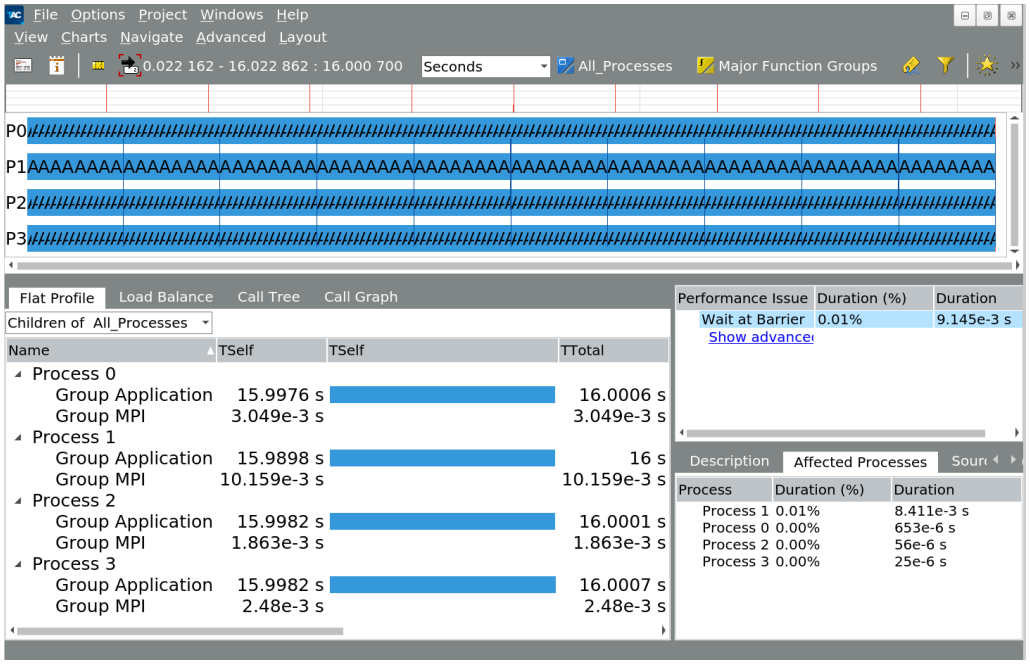
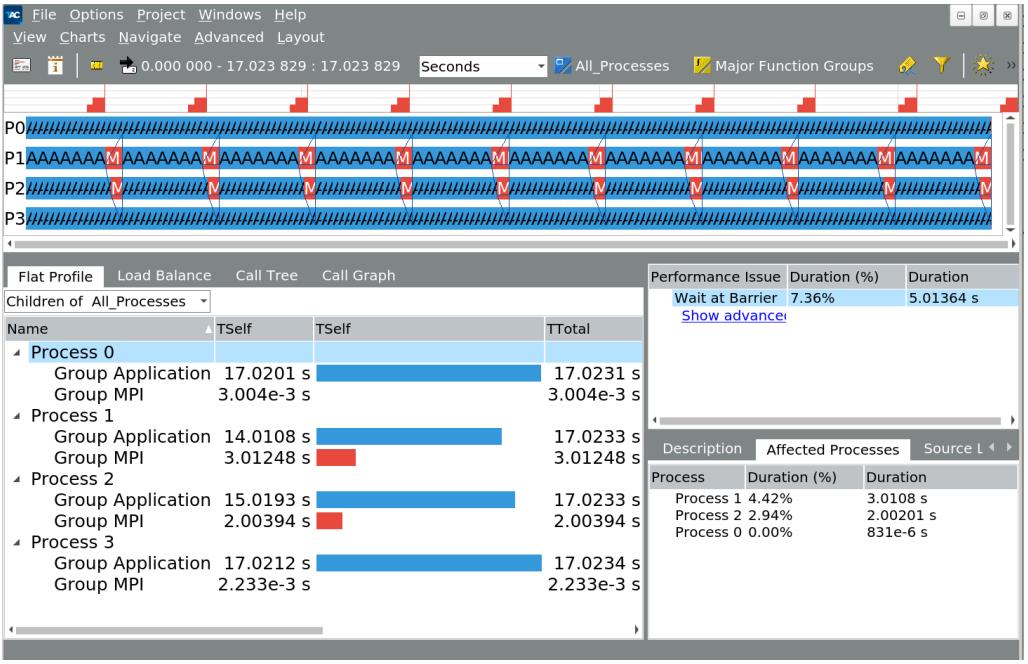
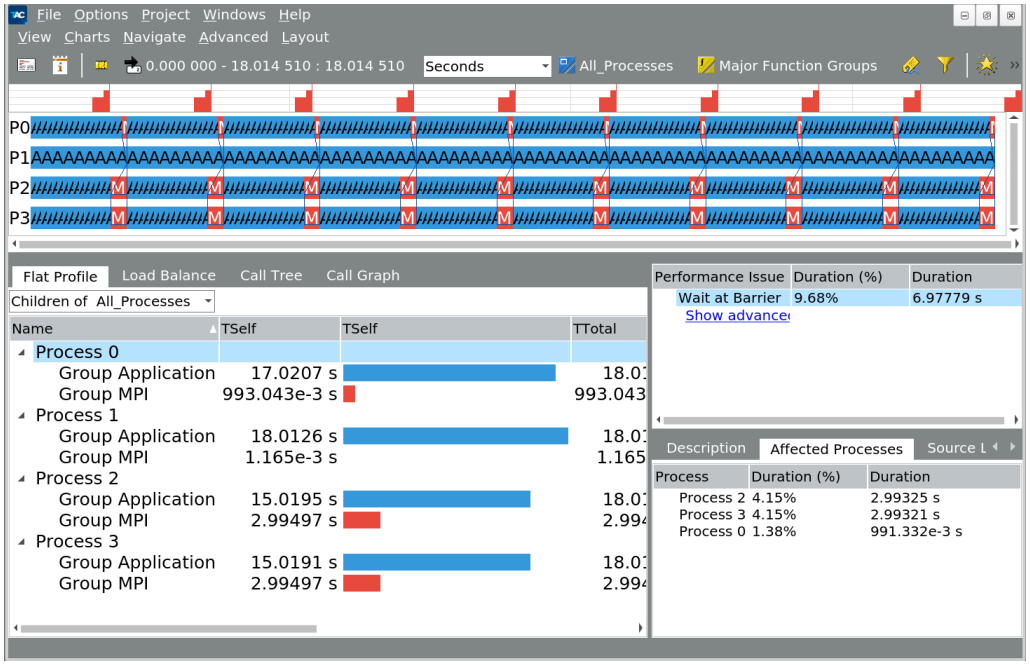


Abbildung 13: Trace Analyzer of optimal parameters - 16 8 16 16



(a) Trace of parameters 17 7 15 17



(b) Trace of parameters 17 9 15 15

Abbildung 14: Trace Analyzer of additional parameters – non-optimal

Literatur