Profiling Techniques and Tools

Tools and techniques for performance analysis

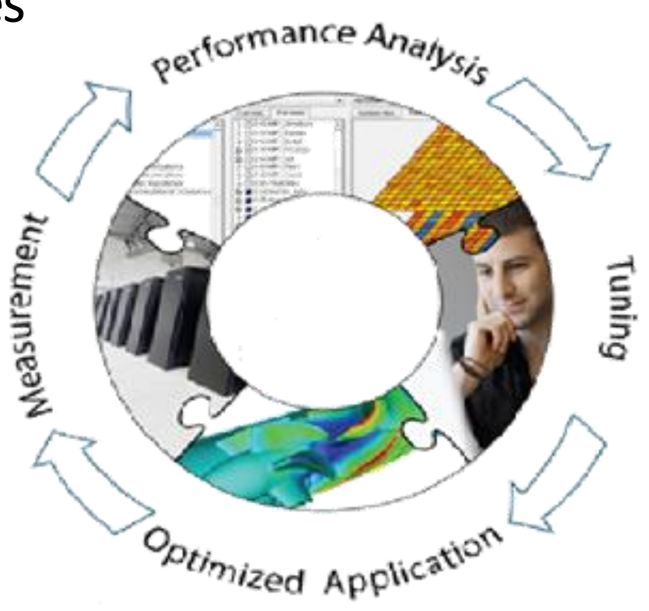
Piero Lanucara, Andy Emerson, Alessandro Marani SCAI team

Contents

- Motivations
- Manual Methods
  - Measuring execution time
  - Profiling PMPI
- Performance Tools
  - gprof
  - Papi
  - Scalasca, Vtune and other packages
- Some advice

Motivations for performance profiling

- Efficient programming on HPC architectures is difficult
  - because modern HPC architectures are complex:
    - different types and speeds of memory (memory hierarchies)
    - presence of accelerators such as MICs, FPGAs and GPUs
    - mutiple filesystem technologies (local, gpfs, SSD, etc)
    - network topologies
    - PARALLELISM !
- For programmers it is essential to use profiling tools in order to optimise and parallelise their applications. Just using –O3 is not usually enough.
- Even for users (rather than programmers) it may be useful to profile in order to choose the best build, hardware and input options.



Measuring execution time without source code

- UNIX/Linux users often use the time command.
- This has the advantages that the source code does not need to be re-compiled and has no overhead (i.e. non-intrusive). Note the different formats of the UNIX and the bash versions.
- In a script, convenient to report on the wall time using date.

```
/usr/bin/time ./a.out
0.00user 0.00system 0:10.07elapsed 0%CPU (0avgtext+0avgdata
848maxresident)k inputs+0outputs (0major+259minor)pagefaults 0swaps

time ./a.out
real    0m10.695s
user    0m0.001s
sys     0m0.006s

start_time=$(date +"%s")
...
end_time=$(date +"%s")
walltime=$(($end_time-$start_time))
echo "walltime $walltime"
```

Using time

- For running benchmarks we are normally most interested in the *elapsed* or *walltime*, i.e. the difference between program start and program finish (for parallel programs this means when all tasks and threads have finished).
- But the various time commands can also give other useful information on resources used:

```
/usr/bin/time ./loop
40.90user 0.00system 0:41.00elapsed 99%CPU
(0avgtext+0avgdata 848maxresident)k
0inputs+0outputs
(0major+284minor)pagefaults 0swaps

/usr/bin/time ./sleep
0.00user 0.00system 0:10.00elapsed 0%CPU
(0avgtext+0avgdata 848maxresident)k
0inputs+0outputs
(0major+259minor)pagefaults 0swaps
```

In the first example we have kept the CPU busy with 99% of the CPU used.
In the second example the CPU has been sent to sleep!

Using top and MPI programs

- For MPI programs convenient to log onto the node where the program is running and use the **top** command.

```
PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
8462 aemerson  20   0 12.284g 102952  64044 R 102.9  0.1  14:18.64 namd2
8460 aemerson  20   0 12.284g  96320  57064 R  96.5  0.1  14:17.86 namd2
8461 aemerson  20   0 12.284g 104240  65024 R  96.5  0.1  14:18.58 namd2
8463 aemerson  20   0 12.283g 100728  62076 R  96.5  0.1  14:18.85 namd2
8464 aemerson  20   0 12.284g 105200  65816 R  96.5  0.1  14:18.58 namd2
8465 aemerson  20   0 12.284g 102668  63400 R  96.5  0.1  14:19.09 namd2
8466 aemerson  20   0 12.284g 105540  66424 R  96.5  0.1  14:18.42 namd2
8467 aemerson  20   0 12.283g 102896  64240 R  96.5  0.1  14:19.20 namd2
```

In this way you can check that you really are running a parallel program and multiple cores are being used in a “balanced” fashion(i.e. %CPU=~100%).
top is also useful for the checking the memory required for each process.

OpenMP threads

For OpenMP the top command can give something like this

```
Tasks: 337 total,   6 running, 331 sleeping,   0 stopped,   0 zombie
%Cpu(s): 93.1 us,  0.1 sy,  0.0 ni,  6.8 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem:  13174488+total, 14130592 used, 11761428+free,     1232 buffers
KiB Swap: 32767996 total,        0 used, 32767996 free.  6393776 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 4419 aemerson  20   0  933224 279780   5856 R 800.2  0.2   0:47.07 test

 4428 aemerson  20   0  123820   1824   1160 R   0.3  0.0   0:00.01 top
29436 root       0 -20 9897424 1.217g 109248 S   0.3  1.0 156:38.92 mmfsd
    1 root      20   0   55496   4936   2400 S   0.0  0.0   4:45.87 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:02.66 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   6:40.96 ksoftirqd/0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00 kworker/0:0H
    8 root      rt   0       0      0      0 S   0.0  0.0   0:12.88 migration/0
    9 root      20   0       0      0      0 S   0.0  0.0   0:02.28 rcu_bh
```

8 OpenMP threads

Measuring execution time within the program (serial)

- Programmers generally want more information on which parts of the program consume the most time.
- Both C/C++ and Fortran programmers are used to instrument the code with timing and printing functions to measure and collect or visualize the time spent in critical or computationally intensive code' sections.
  - ❑ **Fortran77**
    - ❑ `etime(),dtime()`
  - ❑ **Fortran90**
    - ❑ `cputime(), system_clock(), date_and_time()`
  - ❑ **C/C++**
    - ❑ `clock()`
- The programmer must be aware though that these methods are intrusive, and introduce overheads to the program code.

Measuring execution time - example

```
C example:


#include <time.h>
clock_t time1, time2;
double dub_time;
…
time1 = clock();
for (i = 0; i < nn; i++)
for (k = 0; k < nn; k++)
for (j = 0; j < nn; j ++)
c[i][j] = c[i][j] + a[i][k]*b[k][j];
time2 = clock();
dub_time = (time2 - time1)/(double) CLOCKS_PER_SEC;
printf("Time -----------------> %lf \n", dub_time);
```

Measuring execution time in parallel programs

- Both MPI and OpenMP provide functions for measuring the elapsed time.

```
double t1,t2;
t1=MPI_Wtime()
..
t2=MPI_Wtime()
elaspsed=t2-t1;
! In FORTRAN MPI_Wtime is a function
double precision t1,t2
t1 = MPI_Wtime()
..
---
// OpenMP
t1 = omp_get_wtime()
```

Profiling (Debugging) MPI with PMPI

- Most MPI implementations provide a profiling interface called PMPI.
- In PMPI each standard MPI function (MPI_) has an equivalent function with prefix PMPI_ (e.g. PMPI_Send, PMI_RECV, etc).
- With PMPI it is possible to customize normal MPI commands to provide extra information useful for profiling (or debugging).
- Not necessary to modify source code since the customized MPI commands can be linked as a separate library during debugging. For production the extra library is not linked and the standard MPI behaviour is used.
- Many third-party profilers (e.g. Scalasca, Vtune, etc) are based on PMPI.

PMPI Examples

```
// profiling example
…
count=0
call MPI_Reduce(count,sum,total_sum,1,MPI_DOUBLE,MPI_SUM,
0,MPI_COMM_WORLD,ierr)
call MPI_Reduce(count,sum,total_sum,1,MPI_DOUBLE,MPI_SUM,
0,MPI_COMM_WORLD,ierr)
…
subroutine MPI_Reduce( count,sum, total_sum,
   one,datatype,op,dest,comm,ierr)
     real(kind(1.d0)) :: sum, total_sum
     integer ierr,count, datatype, dest, tag, comm,op,one
     count=count+1
     call PMPI_Reduce( sum, total_sum, one,datatype,op,
   dest, comm, ierr )
end
```

Profiling using tools and libraries

- The time command may be ok for benchmarking based on elapsed time but is not sufficient for detailed performance analysis.
- Inserting time commands in the source is tedious and not without overheads. There may also be problems of portability between architectures and compilers.
- For these reasons common to use tools such as gprof or third-party tools (some commercial) such Scalasca, Vtune and so on.
- Such profiling tools generally provide a wide variety of performance data:
  - no. of calls and timings of subroutines and functions
  - use of memory, including cache ("cache hits and misses") and presence of memory leaks
  - info related to parallelism, e.g. load balancing, thread usage, use of MPI calls, etc.
  - I/O related performance data
- Other related tools, tracing tools, can give information on the MPI communication patterns.
- All profiling tools have some degree of overhead but unless the analysis is very detailed (i.e. at the statement level) the overheads should be low.

Profiling using gprof

- The GNU profiler “gprof” is an open-source tool that allows the profiling of serial and parallel codes.
- It works by using Time Based Sampling : at intervals the “program counter” is interrogated to decide at which point in the code the execution has arrived.
- To use the GNU profiler:
  - Recompile the source code using the compiler profiling flag:

```
gcc -pg source code

g++ -pg source code

gfortran -pg source code
```

  - Run the executable to allow the generation of the files containing profiling information:
    - At the end of the execution in the working directory will be generated a specific file generally named “gmon.out” containing all the analytic information for the profiler
  - Results analysis

```
gprof executable gmon.out
```

gprof output – Flat profile

Flat profile :

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 48.60      0.41     0.41    10000    41.31    81.61  init(double*, int)
 27.26      0.64     0.23    10000    23.17    40.30  mysum(double*, int)
 20.15      0.82     0.17 100000000     0.00     0.00  add3(double)
  3.56      0.85     0.03                             frame_dummy
```

gprof - flat profile column meanings

- The meaning of the columns displayed in the **flat profile** is:
- **% time**: percentage of the total execution time your program spent in this function
- **cumulative seconds**: cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table
- **self seconds**: number of seconds accounted for by this function alone.
- **calls**: total number of times the function was called
- **self us/calls**: represents the average number of microseconds spent in this function per call
- **total us/call**: represents the average number of microseconds spent in this function and its descendants per call if this function is profiled, else blank
- **name**: name of the function

gprof – call graph

- Also possible to show relations between subroutines and functions and the time used:

```
Call graph (explanation follows)
index % time    self  children    called     name
                                                <spontaneous>
[1]     96.4    0.00    0.82                 main [1]
                0.41    0.40  10000/10000    init(double*, int) [2]
-----------------------------------------------
                0.41    0.40  10000/10000        main [1]
[2]     96.4    0.41    0.40  10000          init(double*, int) [2]
                0.23    0.17  10000/10000        mysum(double*, int) [3]
```

With appropriate compile options various other outputs are also possible (call trees, line-level timings, etc)

gprof limitations

- gprof gives no information on library routines such as MKL (but MKL should already be well optimised)
- The profiler has a fairly high “granularity”, i.e. for complex programs not easy identify performance bottlenecks.
- Can have high performance overheads.
- Not suited for parallel programming (requires analysing a gmon.out file for each parallel process).

PAPI (Performance Application Programming Interface)

- The PAPI is a standard for accessing information provided by *hardware counters*.
- The hardware counters are special registers built into processors which monitor low-level events such as cache misses, no. of floating point instructions executed, vector instructions, etc.
- The hardware counters available depend on the specific CPU model or architecture and are quite difficult to use since they may have different names.
- The aim of PAPI is to provide a portable interface to hardware counters.

PAPI tools

- PAPI can provide low-level information not available from software profilers.
- The PAPI library defines a large number of *Preset Events* including:
  - PAPI_TOT_CYC- total no. of cycles
  - PAPI_TOT_INS – no. of completed instructions
  - PAPI_FP_INS – floating point instructions
  - PAPI_L1_DCM – cache misses in L1
  - ....
- Although you can call directly the PAPI routines from your C or FORTRAN programs you are more likely to use tools or libraries based on PAPI.
- Examples of PAPI tools include:
  - Tau
  - HPC Toolkit
  - Perfsuite
- Others may have PAPI as an option (e.g. Vtune)
- The general procedure (e.g. Tau) is to recompile with the PAPI-enabled library.

Scalasca

- Scalable performance analysis of large-scale applications.
- Tool originally developed by Felix Wolf and co-workers from the Juelich Supercomputing Centre.
- Available for most HPC architectures and compilers and suitable for systems with many thousands of cores (often the best option for Bluegene)
- Free to download and based on “the New BSD open-source license” (i.e. free but copyrighted)
- Scalasca 2.x based on the **Score-P** profiling and tracing infrastructure and uses the and **CUBE4** format profiles and **OTF2** (Open Trace Format 2) format for event traces.
- Score-P and the CUBE-GUI need to be downloaded separately.

Using Scalasca 2.x

1. Compile and link as normal but with scorep:
   - scorep mpif90 -c prog.f90
   - scorep mpif90 –o prog.exe prog.o
2. Run using the scan (= scalasca –analyze) command + mpirun
   - scan mpirun –n 4 ./prog.exe
3. This will create a directory e.g. scorep_DLPOLY_16_sum which can analysed with the square (=scalasca –examine) command
   - square scorep_DLPOLY_16_sum

Using scalasca 2.x

1. Flat (summary) profile
   - square -s scorep_DLPOLY_16_sum
   - less ./scorep_DLPOLY_16_sum/scorep.score

```
Estimated aggregate size of event trace:                   544MB
Estimated requirements for largest trace buffer (max_buf): 35MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       37MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=37MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)

flt     type max_buf[B]     visits time[s] time[%] time/visit[us]  region
         ALL 36,686,355 21,696,937   93.17   100.0           4.29  ALL
         USR 35,811,984 21,377,014   15.56    16.7           0.73  USR
         MPI    695,056    205,337   30.43    32.7         148.20  MPI
         COM    186,446    114,586   47.18    50.6         411.76  COM

         USR 16,463,174 10,100,000    8.16     8.8           0.81  vdw_forces_
         USR 16,463,174 10,100,000    3.24     3.5           0.32  images_
         USR    982,540    304,475    0.21     0.2           0.68  parse_module.strip_blanks_
         USR    657,332    204,422    0.11     0.1           0.54  parse_module.get_word_
         USR    633,126    382,636    0.08     0.1           0.20  uni_
         USR    326,352    100,802    0.63     0.7           6.27  parse_module.word_2_real_
         MPI    272,344     73,856    5.80     6.2          78.58  MPI_Allreduce
         USR    244,764    150,024    0.11     0.1           0.76  box_mueller_
```

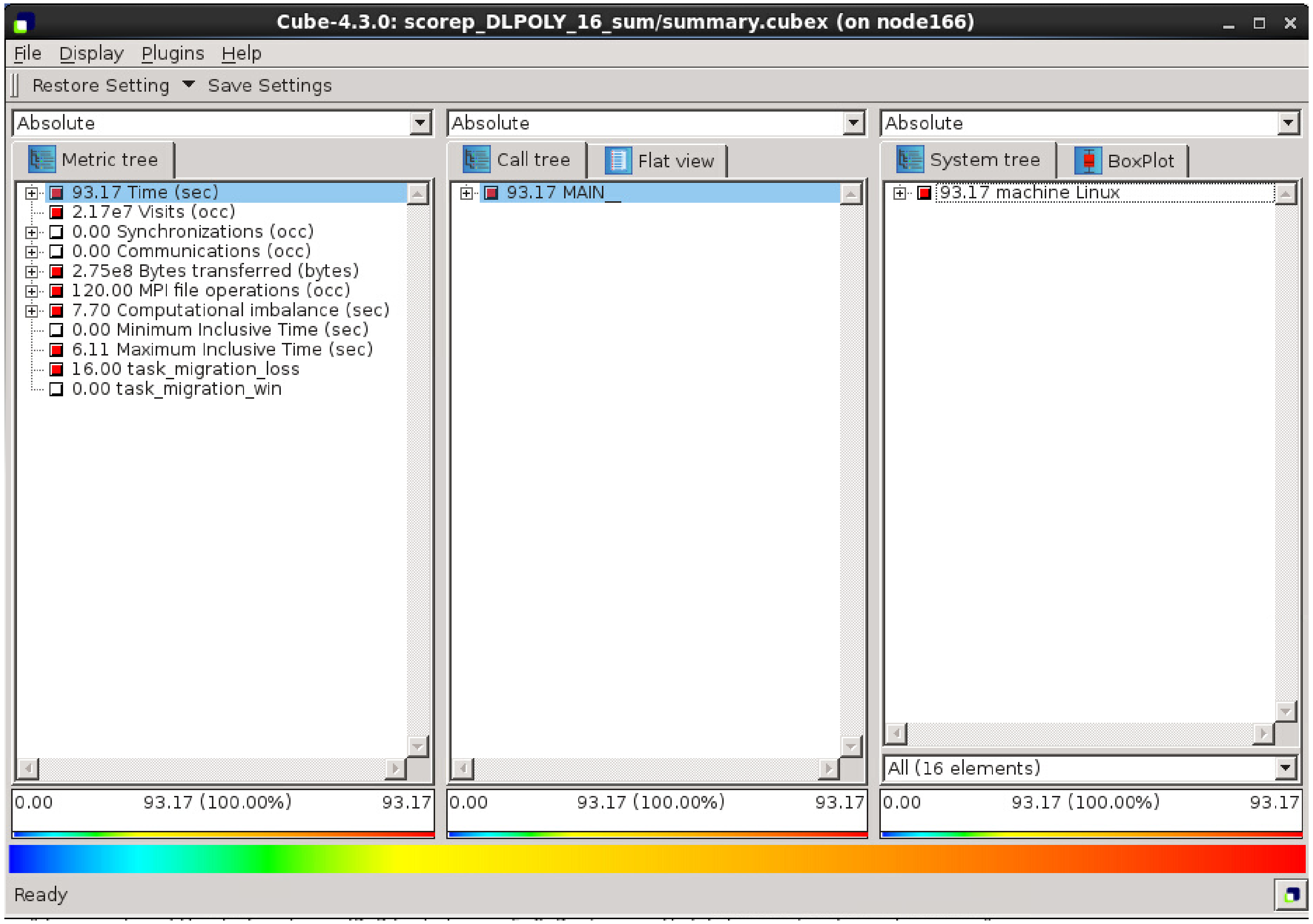
Using scalasca - filters

- Just like any profiling tool, scalasca induces some overhead which may skew the results.
- Particularly relevant for user routines which although require little time are called very frequently: the relative overhead is then quite large.
- In these cases possible to filter the profiling such that these functions are not measured.
- Filtering also useful if the program to be profiled is large and a full event trace is likely to exceed the memory available (look at the first few lines of the summary)

```
SCOREP_REGION_NAMES_BEGIN
        EXCLUDE
                vdw_forces
                images_
SCOREP_REGION_NAMES_END

square -s -f my.filt scorep_DLPOLY_16_sum
```

Using scalasca 2.x - GUI

2. GUI
   – square scorep_DLPOLY_16_sum



Cube-4.3.0: scorep_DLPOLY_16_sum/summary.cubex (on node166)

File Display Plugins Help

Restore Setting Save Settings

Metric tree (Absolute):
```
⊞ 93.17 Time (sec)
  2.17e7 Visits (occ)
⊞ 0.00 Synchronizations (occ)
⊞ 0.00 Communications (occ)
⊞ 2.75e8 Bytes transferred (bytes)
⊞ 120.00 MPI file operations (occ)
⊞ 7.70 Computational imbalance (sec)
  0.00 Minimum Inclusive Time (sec)
  6.11 Maximum Inclusive Time (sec)
  16.00 task_migration_loss
  0.00 task_migration_win
```

Call tree / Flat view (Absolute):
```
⊞ 93.17 MAIN__
```

System tree / BoxPlot (Absolute):
```
⊞ 93.17 machine Linux
```
All (16 elements)

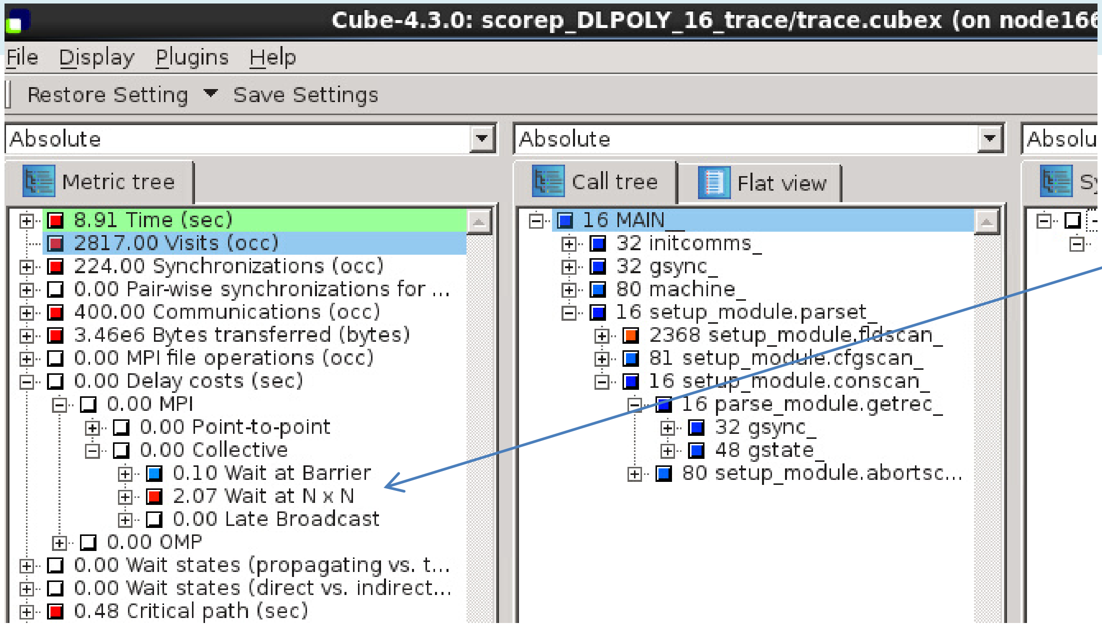
```
0.00    93.17 (100.00%)    93.17
```

Ready

Scalasca and event tracing

- As well as time-averaged summaries, possible to generate also time-stamped event traces.
- Note that because trace profiles can be very large it is strongly recommended to set the total memory allowed and use filters.

```
export SCOREP_TOTAL_MEMORY=55M
scan -q -t -f myfilter.filt mpirun -n 64 ./myexe
square scorep_DLPOLY_16_trace
```


Cube-4.3.0: scorep_DLPOLY_16_trace/trace.cubex (on node166
File Display Plugins Help
Restore Setting Save Settings
Absolute
Metric tree
Call tree
Flat view

```
8.91 Time (sec)
2817.00 Visits (occ)
224.00 Synchronizations (occ)
0.00 Pair-wise synchronizations for ...
400.00 Communications (occ)
3.46e6 Bytes transferred (bytes)
0.00 MPI file operations (occ)
0.00 Delay costs (sec)
  0.00 MPI
    0.00 Point-to-point
    0.00 Collective
      0.10 Wait at Barrier
      2.07 Wait at N x N
      0.00 Late Broadcast
  0.00 OMP
0.00 Wait states (propagating vs. t...
0.00 Wait states (direct vs. indirect...
0.48 Critical path (sec)
```

```
16 MAIN__
  32 initcomms_
  32 gsync_
  80 machine_
  16 setup_module.parset_
    2368 setup_module.fldscan_
    81 setup_module.cfgscan_
    16 setup_module.conscan_
      16 parse_module.getrec_
        32 gsync_
        48 gstate_
      80 setup_module.abortsc...
```

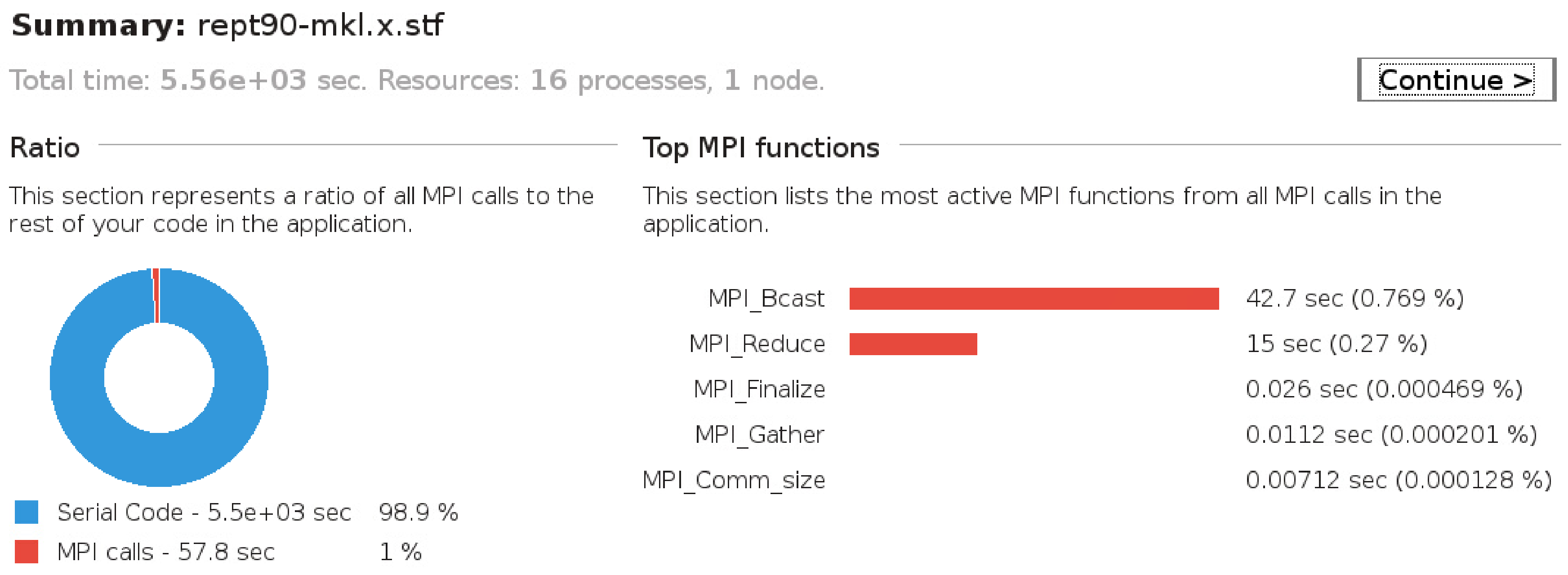
Similar output to a profile but gives time-dependent information.

Intel Trace Analyzer and Collector (ITAC)

- Graphical tool from Intel for understanding MPI application behaviour.
- Convenient because no need to re-compile the program.

```
#!/bin/bash
#PBS -l select=1:ncpus=4:mpiprocs=4
#PBS -l walltime=30:00
#PBS -A cin_staff
#PBS -W group_list=cin_staff

cd $PBS_O_WORKDIR

module load autoload intelmpi
module load mkl
source /cineca/prod/compilers/intel/pe-xe-
2016/binary/itac/9.1.1.017/intel64/bin/itacvars.sh
mpirun -trace -n 2 ./rept90-mkl.x
---
traceanalyzer  ./rept90-mkl.stf
```

ITAC output



Summary: rept90-mkl.x.stf

Total time: **5.56e+03** sec. Resources: **16** processes, **1** node.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

🟦	Serial Code - 5.5e+03 sec	98.9 %
🟥	MPI calls - 57.8 sec	1 %

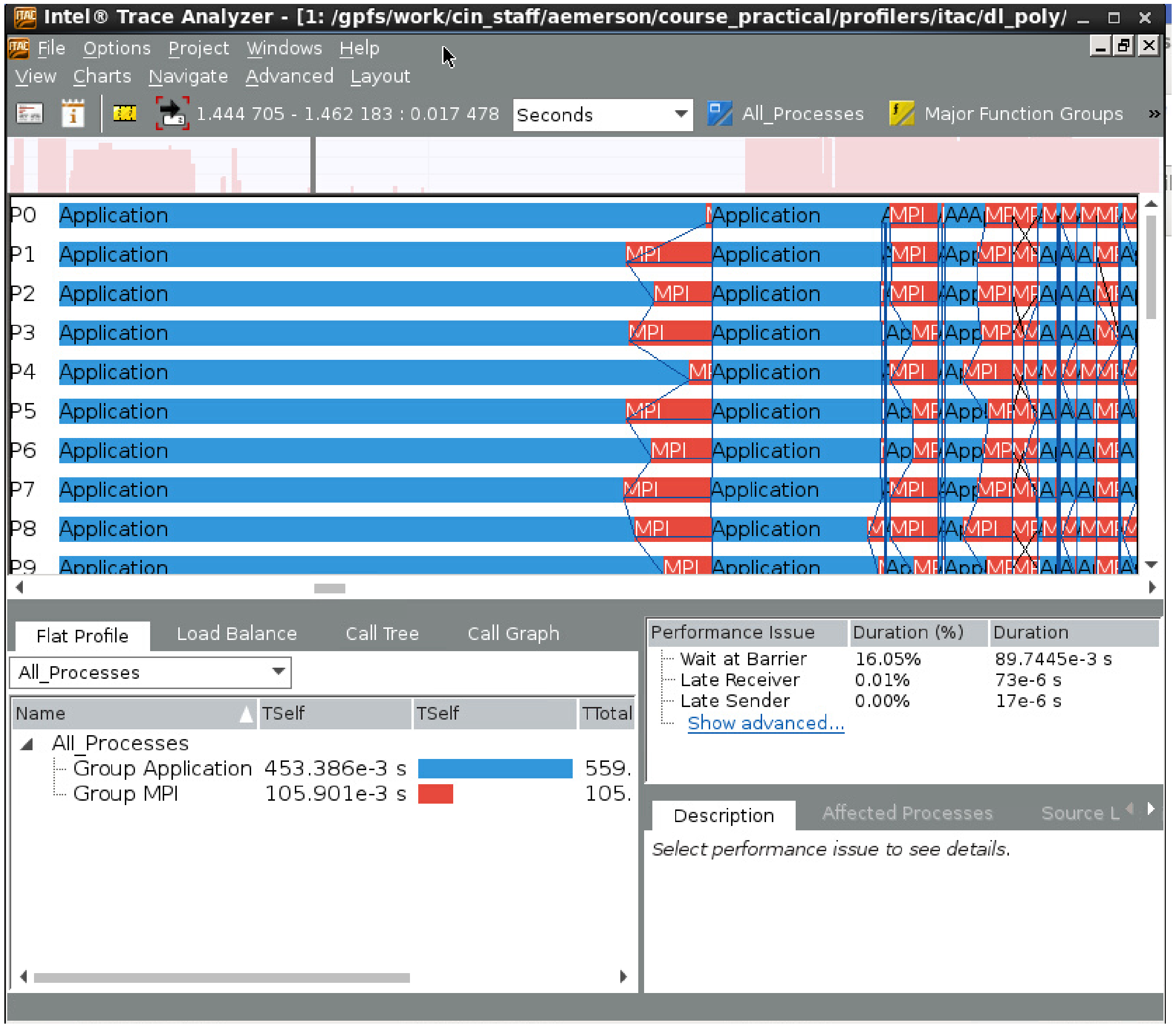
Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Bcast	42.7 sec (0.769 %)
MPI_Reduce	15 sec (0.27 %)
MPI_Finalize	0.026 sec (0.000469 %)
MPI_Gather	0.0112 sec (0.000201 %)
MPI_Comm_size	0.00712 sec (0.000128 %)

This example shows that the application spends very little time in MPI calls and when it does only in collectives.

ITAC output -2


Intel® Trace Analyzer - [1: /gpfs/work/cin_staff/aemerson/course_practical/profilers/itac/dl_poly/
File Options Project Windows Help
View Charts Navigate Advanced Layout
Seconds
All_Processes
Major Function Groups
P0
P1
P2
P3
P4
P5
P6
P7
P8
P9
Application
MPI
Flat Profile
Load Balance
Call Tree
Call Graph
Description
Affected Processes
Source L

1.444 705 - 1.462 183 : 0.017 478

All_Processes

Name	TSelf	TSelf	TTotal
All_Processes			
Group Application	453.386e-3 s	🟦	559.
Group MPI	105.901e-3 s	🟥	105.

Performance Issue	Duration (%)	Duration
Wait at Barrier	16.05%	89.7445e-3 s
Late Receiver	0.01%	73e-6 s
Late Sender	0.00%	17e-6 s
Show advanced...		

Select performance issue to see details.

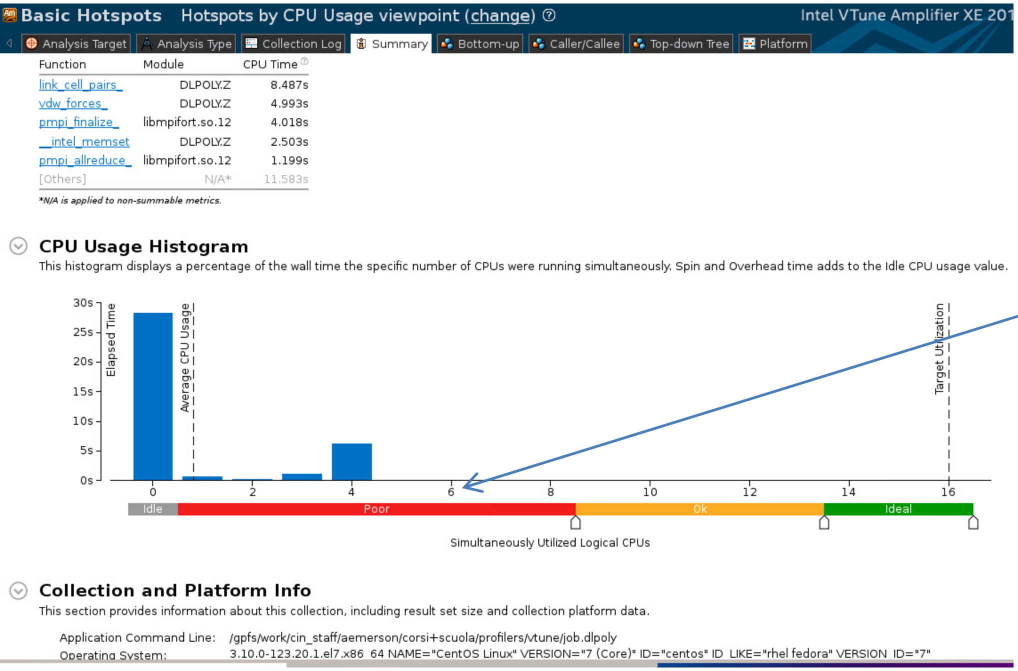
shows more detailed interactions between MPI processes

Intel Vtune Amplifier

- Comprehensive Intel Performance profiler.
- Best used in interactive mode of PBS.

```
qsub -l select=1:ncpus=16,walltime=30:00 -A cin_staff -I
cd $PBS_O_WORKDIR
module load autoload vtune
amplxe-gui &
# or command line
amplxe-cl -collect hotspots -- home/myprog
```

Vtune



Basic Hotspots Hotspots by CPU Usage viewpoint (change) Intel VTune Amplifier XE 201

Function	Module	CPU Time
link_cell_pairs_	DLPOLY.Z	8.487s
vdw_forces_	DLPOLY.Z	4.993s
pmpi_finalize_	libmpifort.so.12	4.018s
__intel_memset	DLPOLY.Z	2.503s
pmpi_allreduce_	libmpifort.so.12	1.199s
[Others]	N/A*	11.583s

*N/A is applied to non-summable metrics.

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

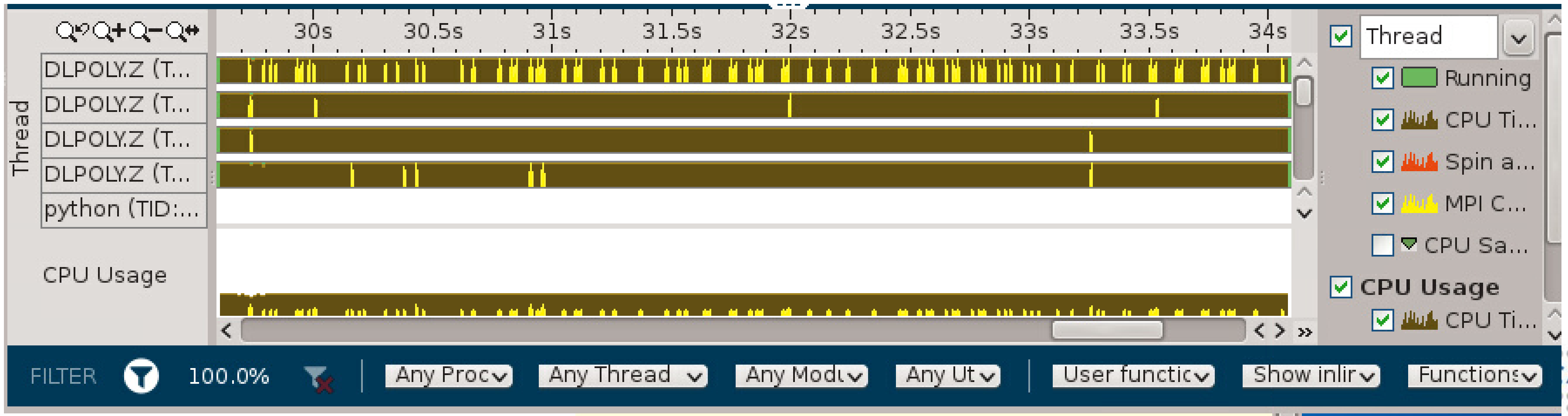
Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Application Command Line: /gpfs/work/cin_staff/aemerson/corsi+scuola/profilers/vtune/job.dlpoly

Operating System: 3.10.0-123.20.1.el7.x86_64 NAME="CentOS Linux" VERSION="7 (Core)" ID="centos" ID_LIKE="rhel fedora" VERSION_ID="7"

colours are misleading because assumes all cores in the node should be used



Some considerations

- Debugging and profiling/tracing are closely related – unexpected poor performance or parallel scaling are also bugs.
- Like debugging, parallelism complicates the profiling procedure. Parallel profiling tools require time and effort. Useful to start with serial program and/or flat profiles before full-scaling profiling.
- Other useful hints:
  - use multiple test cases to activate all the code parts
  - use "realistic" test cases, and with different sizes
  - try different tools and, if possible, different architectures
  - for very complex programs consider isolating the critical code in mock-ups or miniapps to simplify the procedure

Hands-on Session: Profiling

Hands-on Session: Himeno benchmark

- The Hands-on session is based on a **modified** version of the well-known Himeno benchmark
- Himeno benchmark (from Dr. Ryutaro Himeno) takes the core of Poisson equation solver (Jacobi iterative scheme)
- Performances (MFLOPS/GFLOPS) obtained immediately
- It can be used to quickly evaluate computer performances
- Many programming paradigms (Serial C, Serial Fortran, OpenMP, MPI, MPI+openMP, MPI+CUDA, ….)
- Fortran MPI version used for this session

Hands-on Session: Himeno benchmark

- A simple bug (in performances) is injected in Himeno benchmark main'solver part (jacobi subroutine)
- Overall computational performances are affected by simply doubling the process-0 workload
- We know the (performance) bug in advance….
- ….sadly, this is very often not the case!
- The idea is to use different profiling tools to understand their path-to-solution effectiveness…

Hands-on Session: Exercise 1

- Download ***profiling.tgz*** from hpc-forge CINECA repository
- *Extract the **SUMMER2017_PROFILING** directory*
- *Enter in the main directory. Follow the instructions contained in the **README** file for the **BASIC HIMENO IMPLEMENTATION** part*

Lesson learned: Exercise 1

- You should probably have verified the impact of the performance bug running the two executables (balanced and unbalanced versions)
- This investigation should be fine, at least for this simple test.
- More in general, try to span other useful parameters (number of MPI processes, size of the test case…)

SuperComputing Applications and Innovation

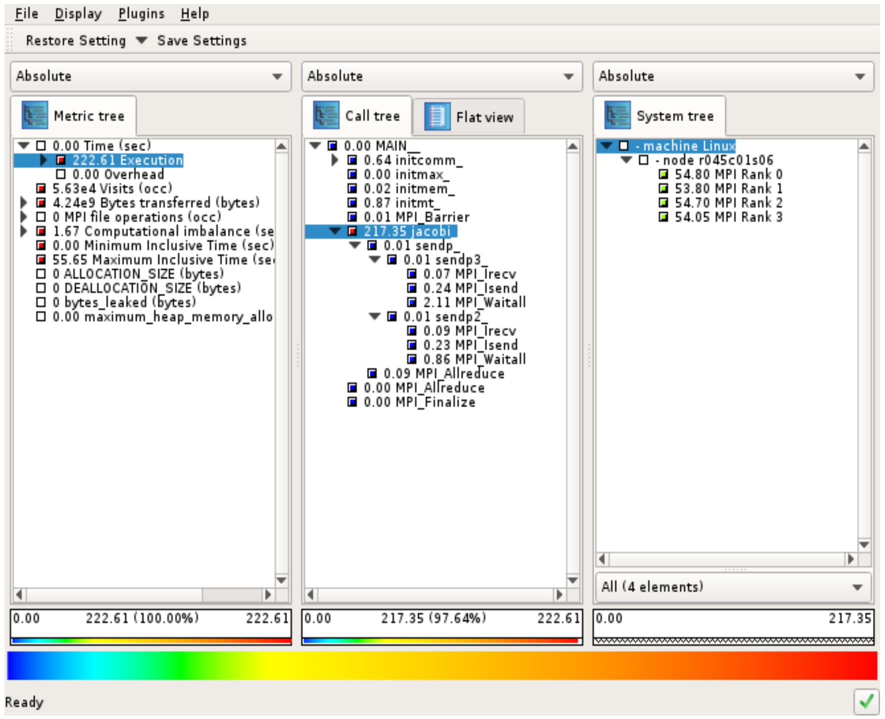
Profiling tools: three different ones

Hands-on Session: Exercise 2

- *Enter in the **SCALASCA** sub-directory. Follow the instructions contained in the **README** file for the **SCALASCA HIMENO PROFILING** part*

Lesson learned: Exercise 2

- You should probably have the SCOREP and TRACES folders, both for balanced and unbalanced tests.
- the SCOREP output (this is the summary SCALASCA profiling) should be fine at least for the good (balanced) test case.



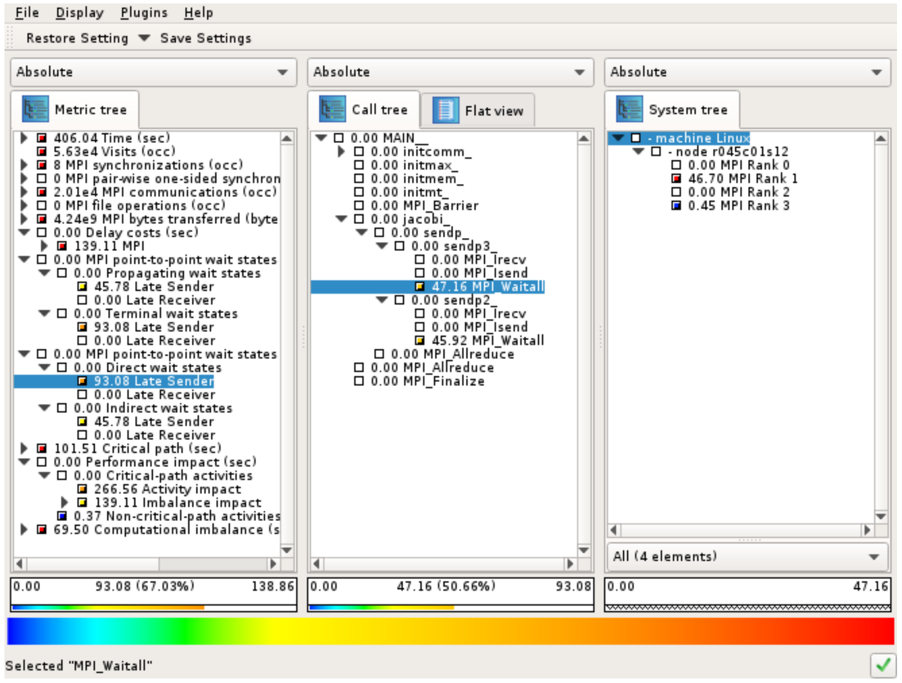
```
Metric tree
0.00 Time (sec)
  222.61 Execution
  0.00 Overhead
5.63e4 Visits (occ)
4.24e9 Bytes transferred (bytes)
0 MPI file operations (occ)
1.67 Computational imbalance (se
0.00 Minimum Inclusive Time (sec)
55.65 Maximum Inclusive Time (se
0 ALLOCATION_SIZE (bytes)
0 DEALLOCATION_SIZE (bytes)
0 bytes_leaked (bytes)
0.00 maximum_heap_memory_allo

0.00        222.61 (100.00%)        222.61
```

```
Call tree
0.00 MAIN__
  0.64 initcomm_
  0.00 initmax_
  0.02 initmem_
  0.87 initmt_
  0.01 MPI_Barrier
  217.35 jacobi
    0.01 sendp_
      0.01 sendp3_
        0.07 MPI_Irecv
        0.24 MPI_Isend
        2.11 MPI_Waitall
      0.01 sendp2_
        0.09 MPI_Irecv
        0.23 MPI_Isend
        0.86 MPI_Waitall
    0.09 MPI_Allreduce
  0.00 MPI_Allreduce
  0.00 MPI_Finalize

0.00        217.35 (97.64%)        222.61
```

```
System tree
- machine Linux
  - node r045c01s06
      54.80 MPI Rank 0
      53.80 MPI Rank 1
      54.70 MPI Rank 2
      54.05 MPI Rank 3

0.00                               217.35
```

Lesson learned: Exercise 2

- Instead, for the bad (unbalanced) test case is not so easy to understand the root of the poor performances.
- On deeper TRACES analysis it is apparent how the problem is not only confined to the bug injected into the jacobi subroutine but more on the delay (direct and indirect) of process-0 which causes late sender wait states.



```
406.04 Time (sec)
5.63e4 Visits (occ)
8 MPI synchronizations (occ)
0 MPI pair-wise one-sided synchron
2.01e4 MPI communications (occ)
0 MPI file operations (occ)
4.24e9 MPI bytes transferred (byte
0.00 Delay costs (sec)
  139.11 MPI
0.00 MPI point-to-point wait states
  0.00 Propagating wait states
    45.78 Late Sender
    0.00 Late Receiver
  0.00 Terminal wait states
    93.08 Late Sender
    0.00 Late Receiver
0.00 MPI point-to-point wait states
  0.00 Direct wait states
    93.08 Late Sender
    0.00 Late Receiver
  0.00 Indirect wait states
    45.78 Late Sender
    0.00 Late Receiver
101.51 Critical path (sec)
0.00 Performance impact (sec)
  0.00 Critical-path activities
    266.56 Activity impact
    139.11 Imbalance impact
  0.37 Non-critical-path activities
69.50 Computational imbalance (s
```

```
0.00 MAIN__
  0.00 initcomm_
  0.00 initmax_
  0.00 initmem_
  0.00 initmt_
  0.00 MPI_Barrier
  0.00 jacobi_
    0.00 sendp_
      0.00 sendp3_
        0.00 MPI_Irecv
        0.00 MPI_Isend
        47.16 MPI_Waitall
      0.00 sendp2_
        0.00 MPI_Irecv
        0.00 MPI_Isend
        45.92 MPI_Waitall
    0.00 MPI_Allreduce
  0.00 MPI_Allreduce
  0.00 MPI_Finalize
```

```
- machine Linux
  - node r045c01s12
      0.00 MPI Rank 0
      46.70 MPI Rank 1
      0.00 MPI Rank 2
      0.45 MPI Rank 3
```

```
0.00    93.08 (67.03%)    138.86
0.00    47.16 (50.66%)    93.08
0.00                      47.16
```

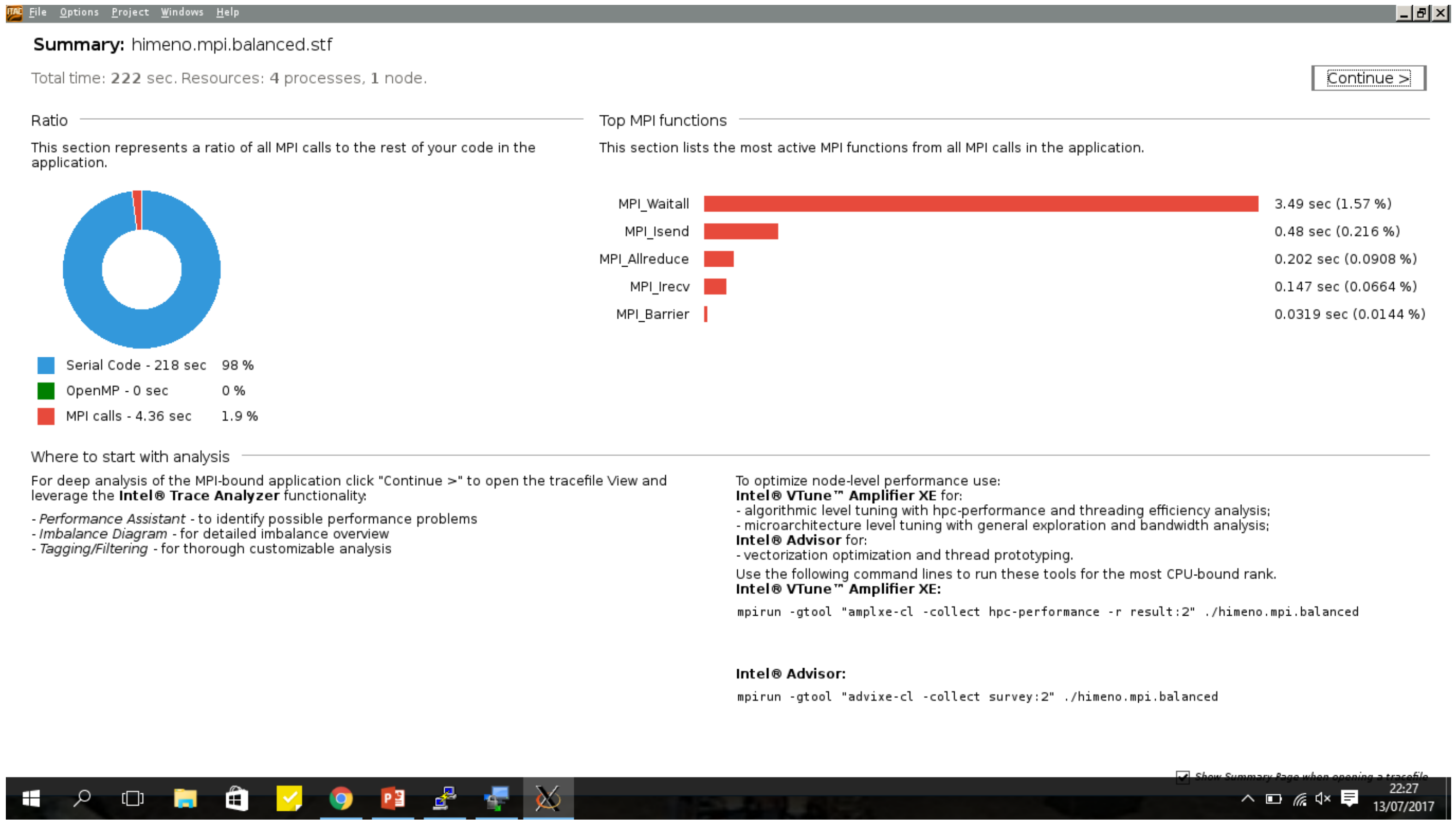
Selected "MPI_Waitall"

Hands-on Session: Exercise 3

- *Enter in the **ITAC** sub-directory. Follow the instructions contained in the **README** file for the **ITAC HIMENO PROFILING** part*

Lesson learned: Exercise 3

- You should probably have the *stf* files, both for balanced and unbalanced tests.
- the ITAC analysis is quite clear and all the relevant informations are easily included in the profiler output windows.


File Options Project Windows Help
Continue >
☑ Show Summary Page when opening a tracefile
22:27
13/07/2017

Summary: himeno.mpi.balanced.stf

Total time: **222** sec. Resources: **4** processes, **1** node.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

🟦	Serial Code - 218 sec	98 %
🟩	OpenMP - 0 sec	0 %
🟥	MPI calls - 4.36 sec	1.9 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the application.

MPI_Waitall	3.49 sec (1.57 %)
MPI_Isend	0.48 sec (0.216 %)
MPI_Allreduce	0.202 sec (0.0908 %)
MPI_Irecv	0.147 sec (0.0664 %)
MPI_Barrier	0.0319 sec (0.0144 %)

Where to start with analysis

For deep analysis of the MPI-bound application click "Continue >" to open the tracefile View and leverage the **Intel® Trace Analyzer** functionality:

- *Performance Assistant* - to identify possible performance problems
- *Imbalance Diagram* - for detailed imbalance overview
- *Tagging/Filtering* - for thorough customizable analysis

To optimize node-level performance use:
Intel® VTune™ Amplifier XE for:
- algorithmic level tuning with hpc-performance and threading efficiency analysis;
- microarchitecture level tuning with general exploration and bandwidth analysis;

Intel® Advisor for:
- vectorization optimization and thread prototyping.

Use the following command lines to run these tools for the most CPU-bound rank.
Intel® VTune™ Amplifier XE:

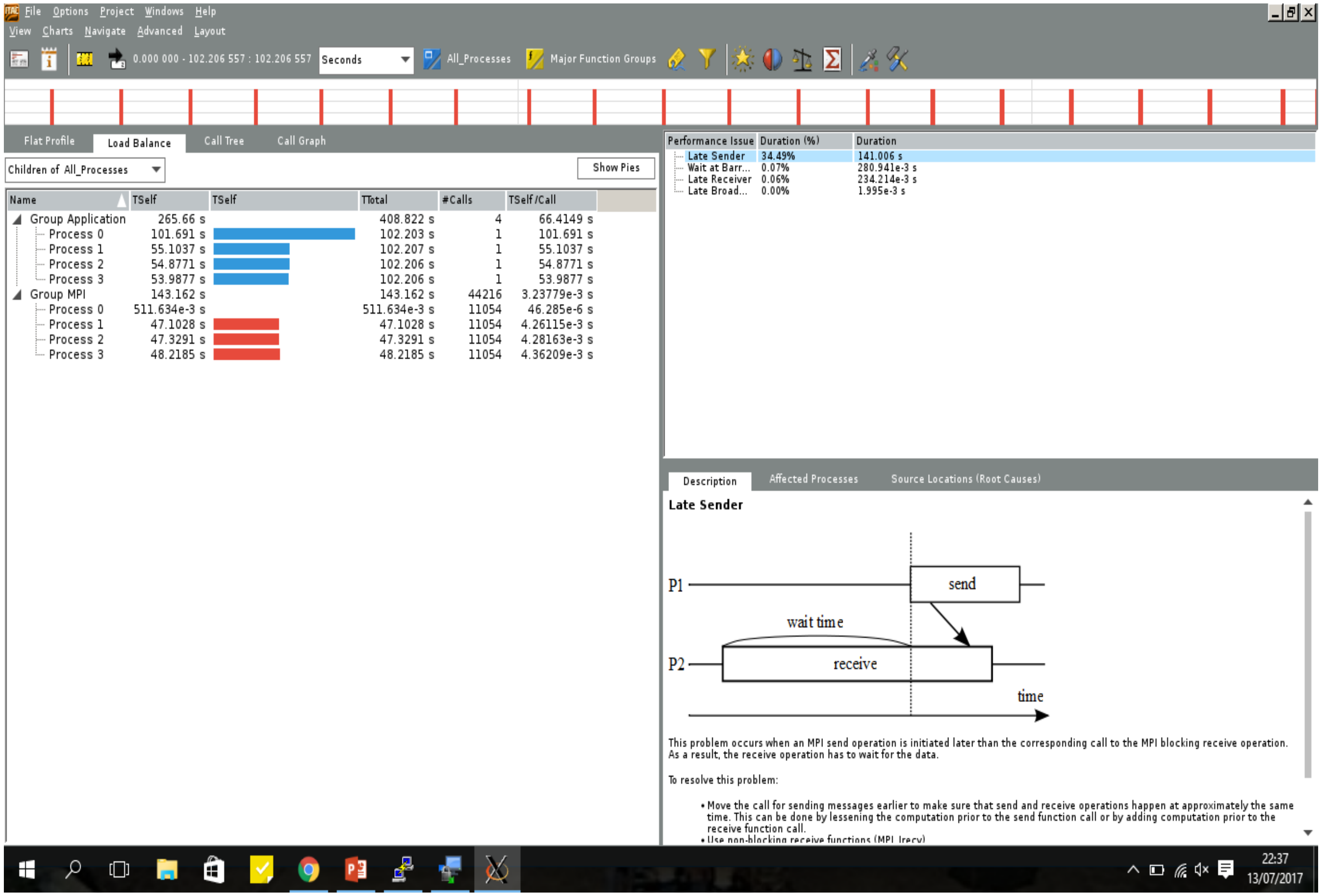
```
mpirun -gtool "amplxe-cl -collect hpc-performance -r result:2" ./himeno.mpi.balanced
```

Intel® Advisor:

```
mpirun -gtool "advixe-cl -collect survey:2" ./himeno.mpi.balanced
```

Lesson learned: Exercise 3

- For the bad (unbalanced) test case you have to navigate into the different levels of investigation, but this operation is easily understood.
- In the end, ITAC tool is quite good for a quick-and-dirty analysis of simple codes. May not be the best choice for huge size, complicated parallel applications.



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Group Application	265.66 s		408.822 s	4	66.4149 s
Process 0	101.691 s	🟦	102.203 s	1	101.691 s
Process 1	55.1037 s	🟦	102.207 s	1	55.1037 s
Process 2	54.8771 s	🟦	102.206 s	1	54.8771 s
Process 3	53.9877 s	🟦	102.206 s	1	53.9877 s
Group MPI	143.162 s		143.162 s	44216	3.23779e-3 s
Process 0	511.634e-3 s		511.634e-3 s	11054	46.285e-6 s
Process 1	47.1028 s	🟥	47.1028 s	11054	4.26115e-3 s
Process 2	47.3291 s	🟥	47.3291 s	11054	4.28163e-3 s
Process 3	48.2185 s	🟥	48.2185 s	11054	4.36209e-3 s

Performance Issue	Duration (%)	Duration
Late Sender	34.49%	141.006 s
Wait at Barr...	0.07%	280.941e-3 s
Late Receiver	0.06%	234.214e-3 s
Late Broad...	0.00%	1.995e-3 s

Late Sender

This problem occurs when an MPI send operation is initiated later than the corresponding call to the MPI blocking receive operation. As a result, the receive operation has to wait for the data.

To resolve this problem:

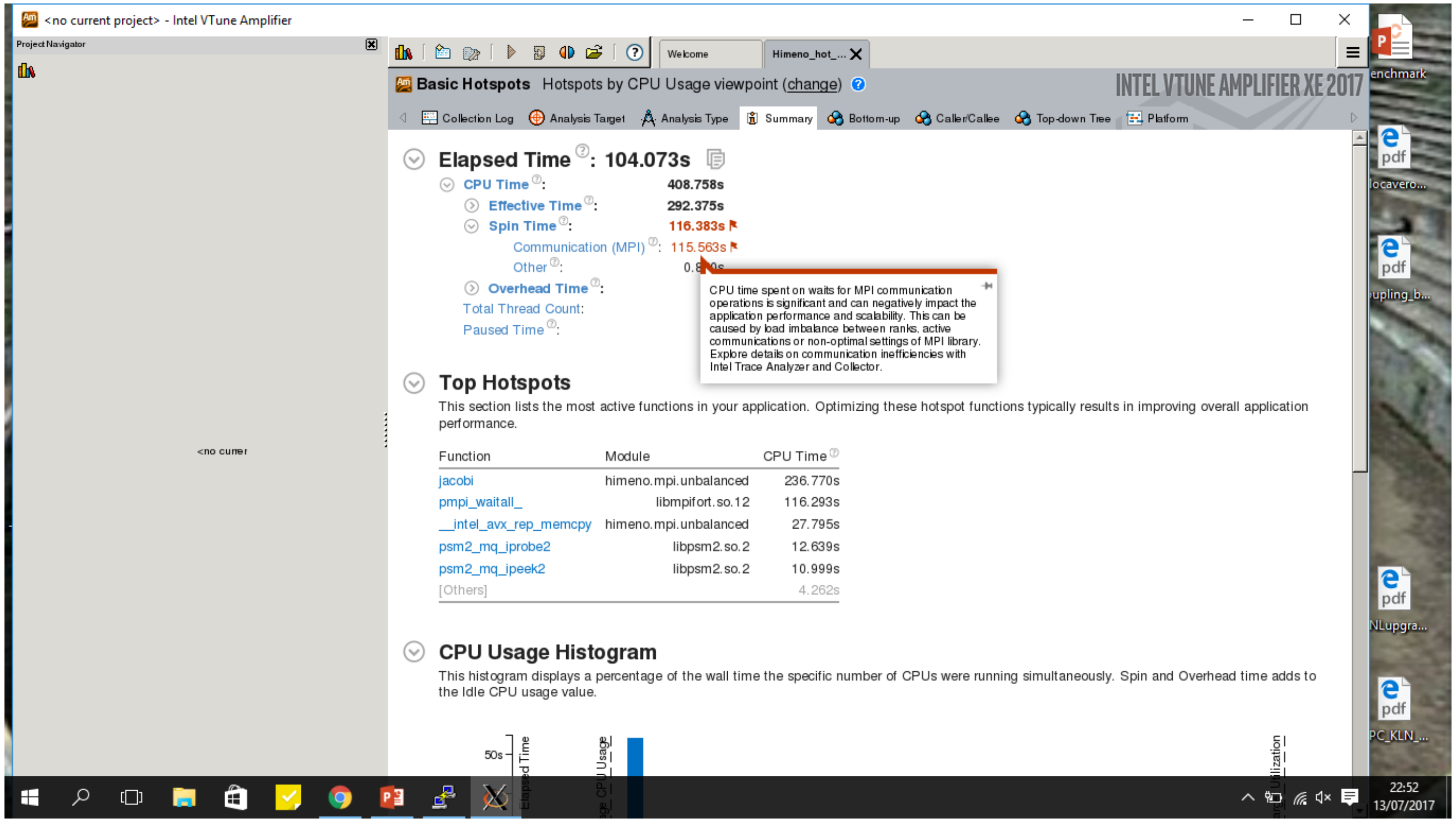
- Move the call for sending messages earlier to make sure that send and receive operations happen at approximately the same time. This can be done by lessening the computation prior to the send function call or by adding computation prior to the receive function call.
- Use non-blocking receive functions (MPI_Irecv)

Hands-on Session: Exercise 4

- *Enter in the **VTUNE** sub-directory. Follow the instructions contained in the **README** file for the **VTUNE HIMENO PROFILING** part*

Lesson learned: Exercise 4

- You should probably have the **Himeno_hot_unbalanced** directory (for the unbalanced tests)
- the VTUNE analysis report related to the so-call (basic or advanced) **HotSpots**
- Bottlenecks finding in an intuitive and clear representation



Elapsed Time: 104.073s

CPU Time:	408.758s
Effective Time:	292.375s
Spin Time:	116.383s
Communication (MPI):	115.563s
Other:	0.8?0s
Overhead Time:	
Total Thread Count:	
Paused Time:	

CPU time spent on waits for MPI communication operations is significant and can negatively impact the application performance and scalability. This can be caused by load imbalance between ranks, active communications or non-optimal settings of MPI library. Explore details on communication inefficiencies with Intel Trace Analyzer and Collector.

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

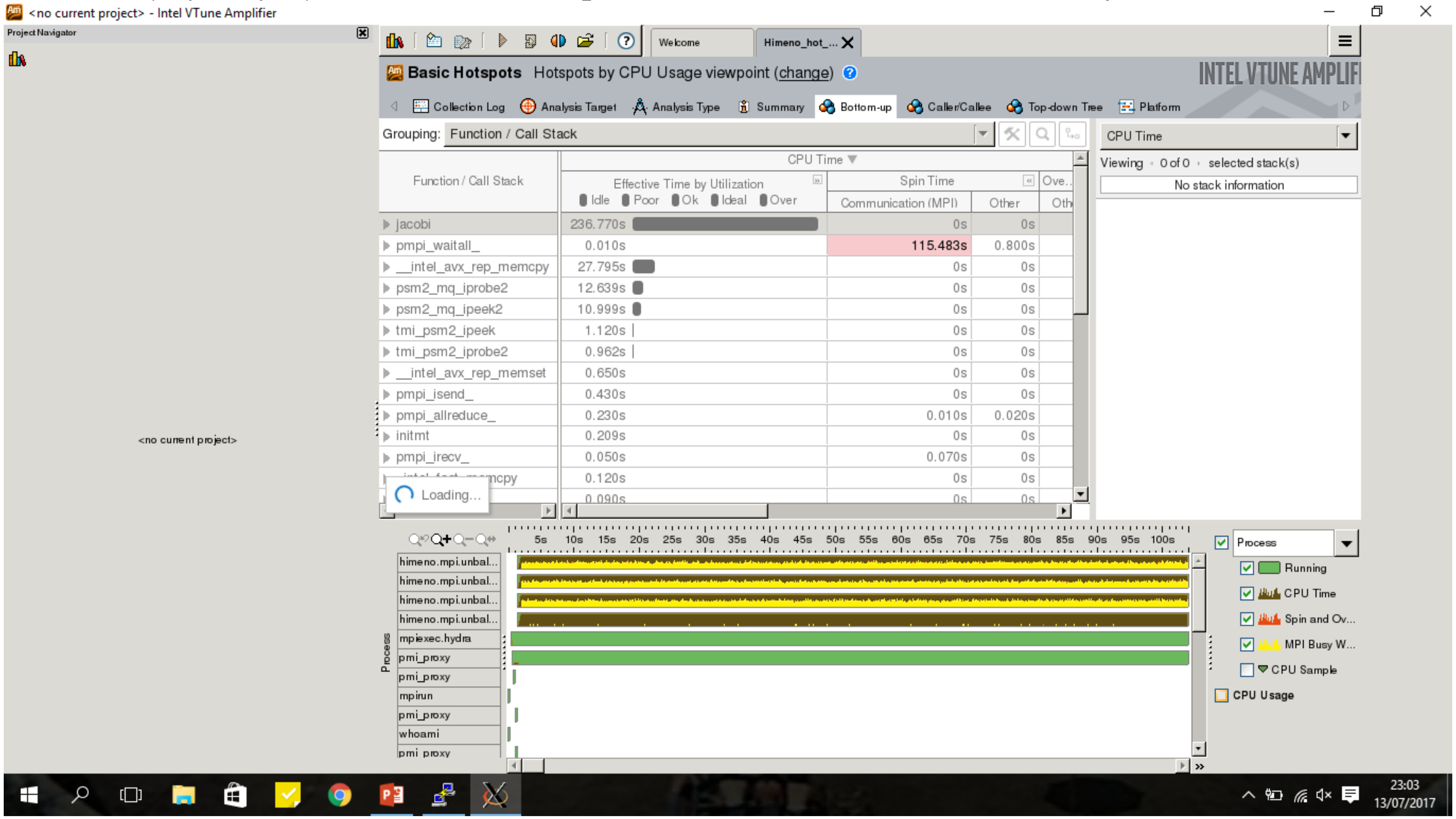
Function	Module	CPU Time
jacobi	himeno.mpi.unbalanced	236.770s
pmpi_waitall_	libmpifort.so.12	116.293s
__intel_avx_rep_memcpy	himeno.mpi.unbalanced	27.795s
psm2_mq_iprobe2	libpsm2.so.2	12.639s
psm2_mq_ipeek2	libpsm2.so.2	10.999s
[Others]		4.262s

CPU Usage Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU usage value.

Lesson learned: Exercise 4

- You should have done the VTUNE analysis report related to **HotSpots** for the unbalanced test case can be enriched.
- Besides, Top HotSpots, the most time consuming functions for the unbalanced test case and the processes



Grouping: Function / Call Stack

Function / Call Stack	CPU Time: Effective Time by Utilization (Idle, Poor, Ok, Ideal, Over)	CPU Time: Spin Time: Communication (MPI)	CPU Time: Spin Time: Other	CPU Time: Ove..: Oth
jacobi	236.770s	0s	0s	
pmpi_waitall_	0.010s	115.483s	0.800s	
__intel_avx_rep_memcpy	27.795s	0s	0s	
psm2_mq_iprobe2	12.639s	0s	0s	
psm2_mq_ipeek2	10.999s	0s	0s	
tmi_psm2_ipeek	1.120s	0s	0s	
tmi_psm2_iprobe2	0.962s	0s	0s	
__intel_avx_rep_memset	0.650s	0s	0s	
pmpi_isend_	0.430s	0s	0s	
pmpi_allreduce_	0.230s	0.010s	0.020s	
initmt	0.209s	0s	0s	
pmpi_irecv_	0.050s	0.070s	0s	
...mcpy	0.120s	0s	0s	
	0.090s	0s	0s	

Timeline: 5s 10s 15s 20s 25s 30s 35s 40s 45s 50s 55s 60s 65s 70s 75s 80s 85s 90s 95s 100s

Process:
- himeno.mpi.unbal...
- himeno.mpi.unbal...
- himeno.mpi.unbal...
- himeno.mpi.unbal...
- mpiexec.hydra
- pmi_proxy
- pmi_proxy
- mpirun
- pmi_proxy
- whoami
- pmi_proxy

23:03
13/07/2017