Profilers and performance evaluation

Tools and techniques for performance analysis

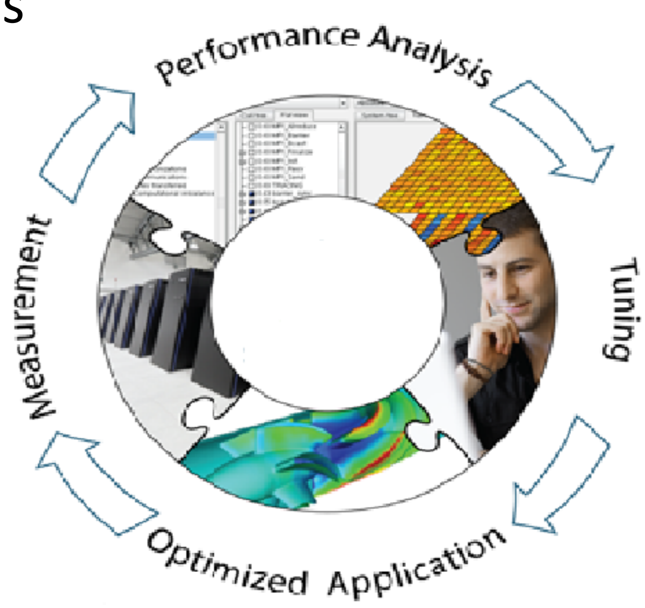
Andrew Emerson, Alessandro Marani

Contents

- Motivations
- Manual Methods
  - Measuring execution time
  - Profiling PMPI
- Performance Tools
  - Prof and gprof
  - Papi
  - Scalasca, Extrae, 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 advantage 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()
```

(Debugging) and profiling 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

```
// profiling example
static int send_count=0;
int MPI_Send(void*start,int count, MPI_Datatype datatype,
   int dest, int tag, MPI_Comm comm)
{
send_count++;
return PMPI_Send(start, count, datatype, dest, tag, comm);
}
```

Debugging

```
! Unsafe uses of MPI_Send
! MPI_Send can be implemented as MPI_Ssend (synchronous send)
subroutine MPI_Send( start, count, datatype, dest,
 tag, comm, ierr )
 integer start(*), count, datatype, dest, tag, comm
 call PMPI_Ssend( start, count, datatype,
 dest, tag, 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 it’s common to use tools such as gprof or third-party tools (some commercial) such as Scalasca, Vtune, Allinea 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, like 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 coulmn 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 it's not easy to 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.

Common profiler/tracing packages

- There are very many profiling packages available. A (very) partial list includes:

Tool name	Suited for	Comments
Scalasca	Profiling and limited tracing of many task programs	Free (GPL)
Intel Trace Analyser and Collector (ITAC)	Quick tool for tracing intel-compiled apps	Intel licensed MPI lightweight tool *
Intel Vtune Amplifier	Detailed profiling with intel applications	Intel licensed profiler *
Extrae/Paraver	General purpose tracing tool	Not currently available at Cineca (but can be installed)
Valgrind	Memory and thread debugging	
Allinea DDT	Commercial debugger+profiler	Not currently available at Cineca (under consideration)
Tau	Profiling and tracing	PAPI based. Can use paraprof for visualisation
Vampir	Tracing	

* limited licenses available at Cineca!

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 mpiifort -c prog.f90
   - scorep mpiifort -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 it is 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)

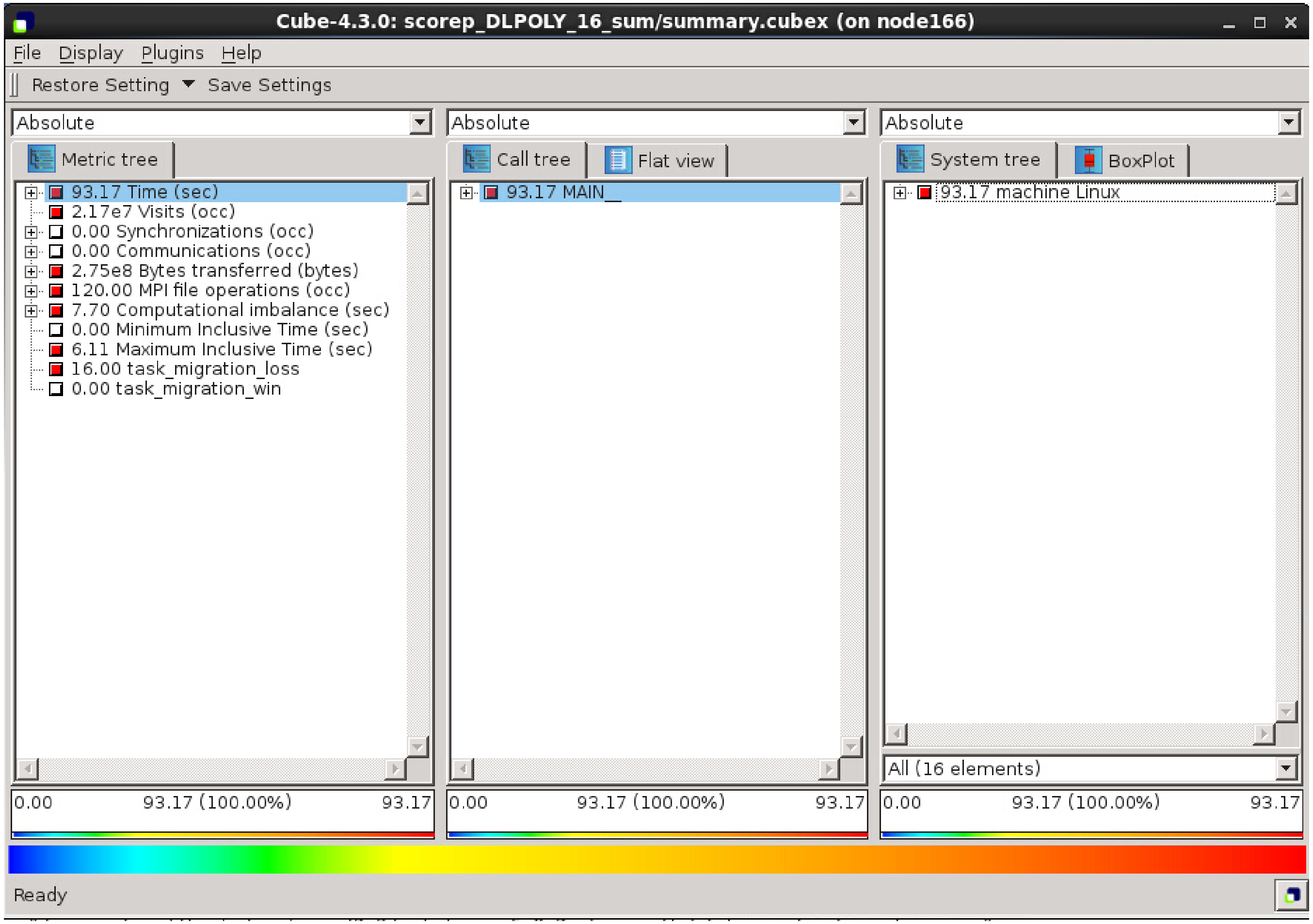
```
SCOREP_REGION_NAMES_BEGIN
        EXCLUDE
                vdw_forces
                images_
SCOREP_REGION_NAMES_END

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

Using scalasca 2.x - GUI

2. GUI

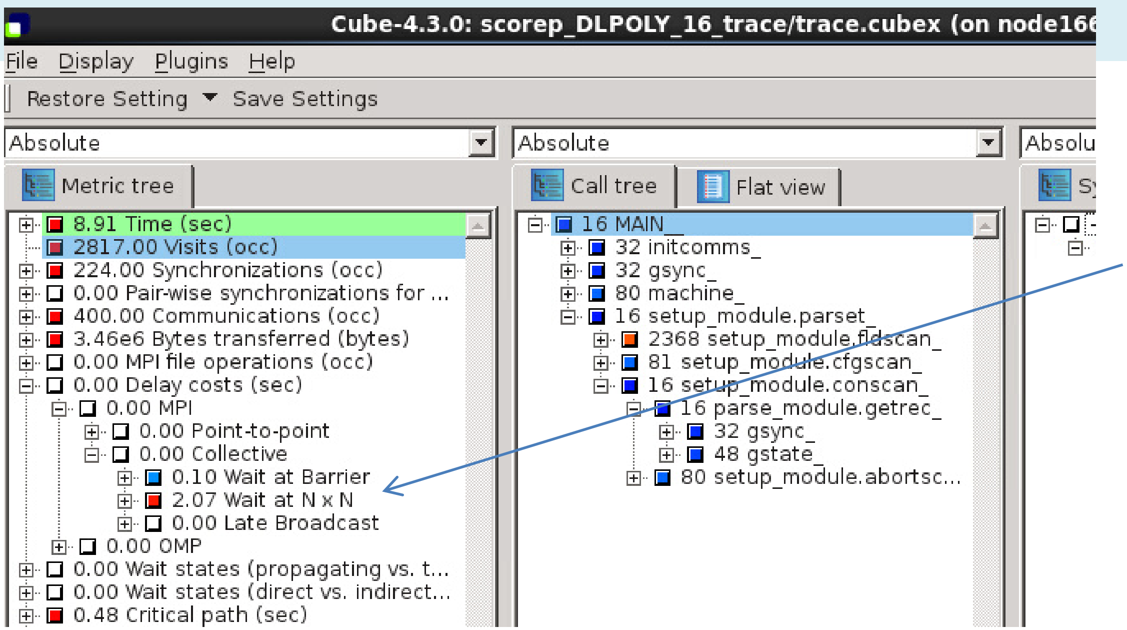
- square scorep_DLPOLY_16_sum



Scalasca and event tracing

- As well as time-averaged summaries, it is 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
```



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

cd $PBS_O_WORKDIR
module load autoload intelmpi
module load mkl
source $ITAC_2017_DIR/intel64/bin/itacvars.sh
mpirun -trace -n 2 ./rept90-mkl.x
---
traceanalyzer  ./rept90-mkl.stf
```

ITAC output

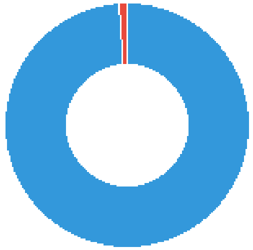
Summary: rept90-mkl.x.stf

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

Continue >

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 %

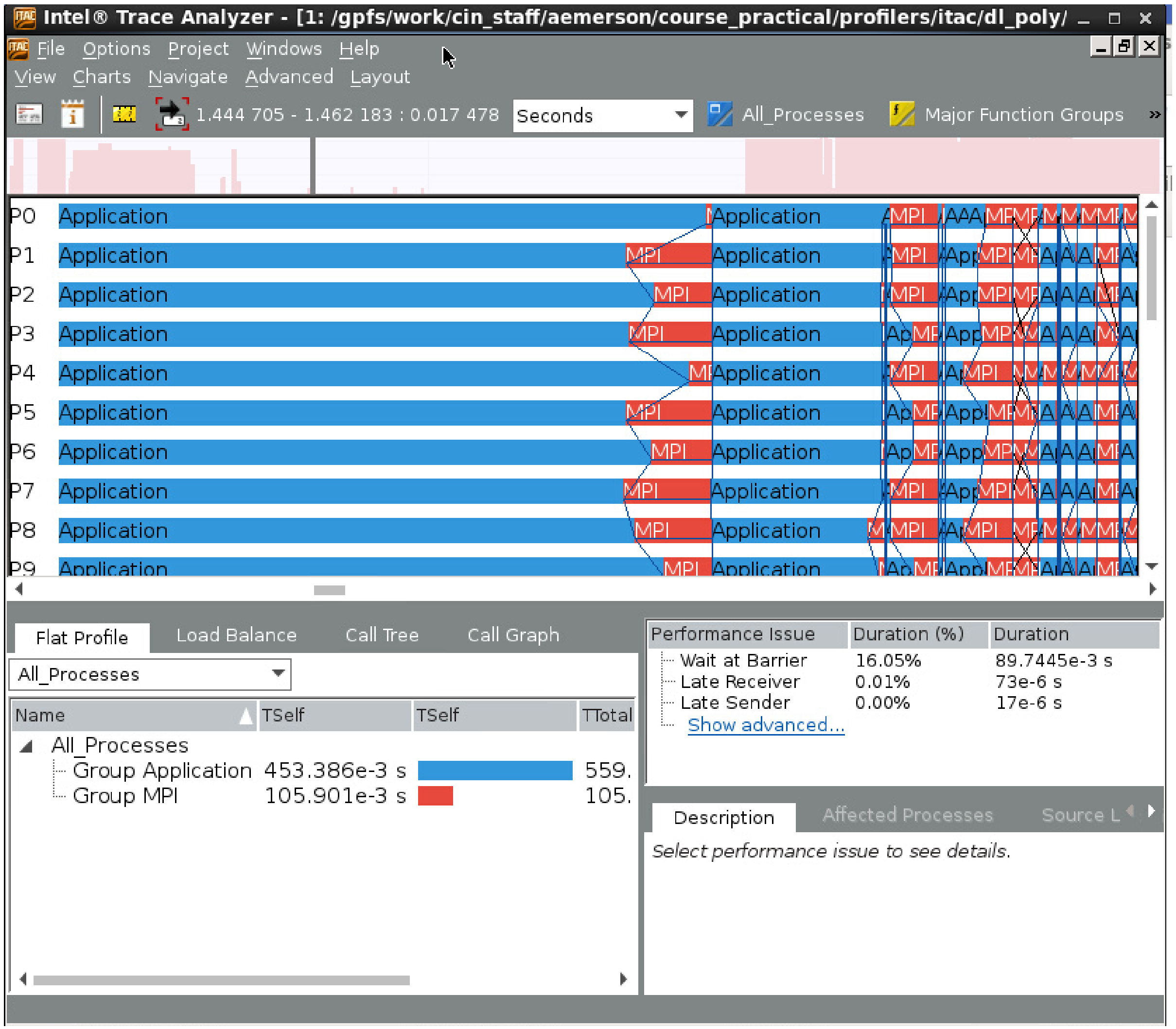
Top MPI functions

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



shows more detailed interactions between MPI processes

ITAC-MPI Performance Snapshot (mps)

- Lightweight, scalable performance tool designed to give quickly information for very large number of processes.
- Included in the Intel ITAC package (>9.0.3), requires Intel MPI >=5.0.3 and Intel compilers >=15.0.1 for OpenMP data.

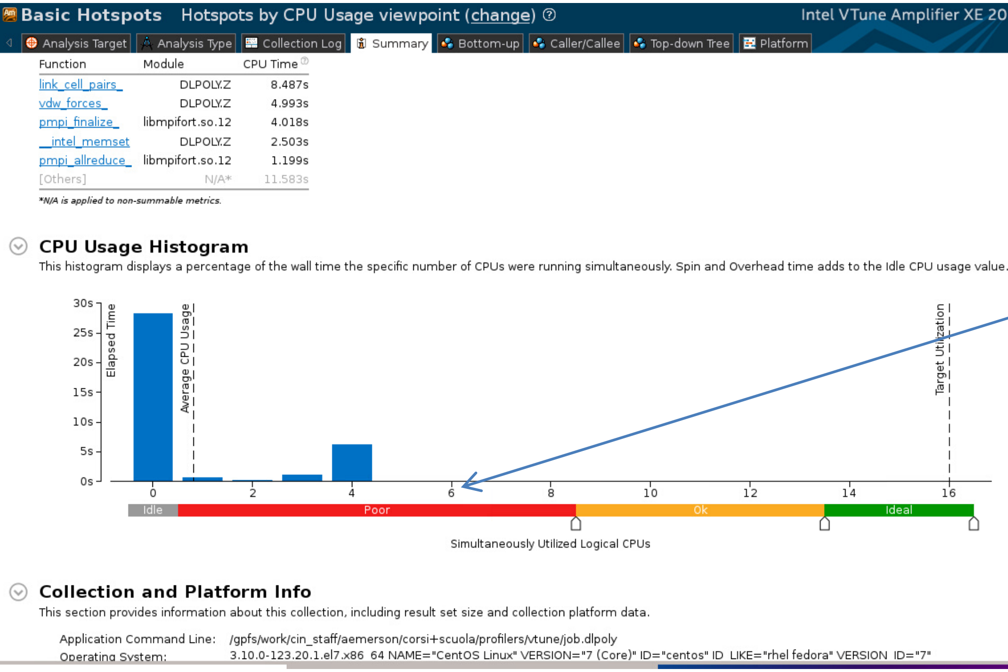
```
source $ITAC_2017_DIR/…/itacvars.sh
mpirun –mps –n 32 ./poisson
```

Intel Vtune Amplifier

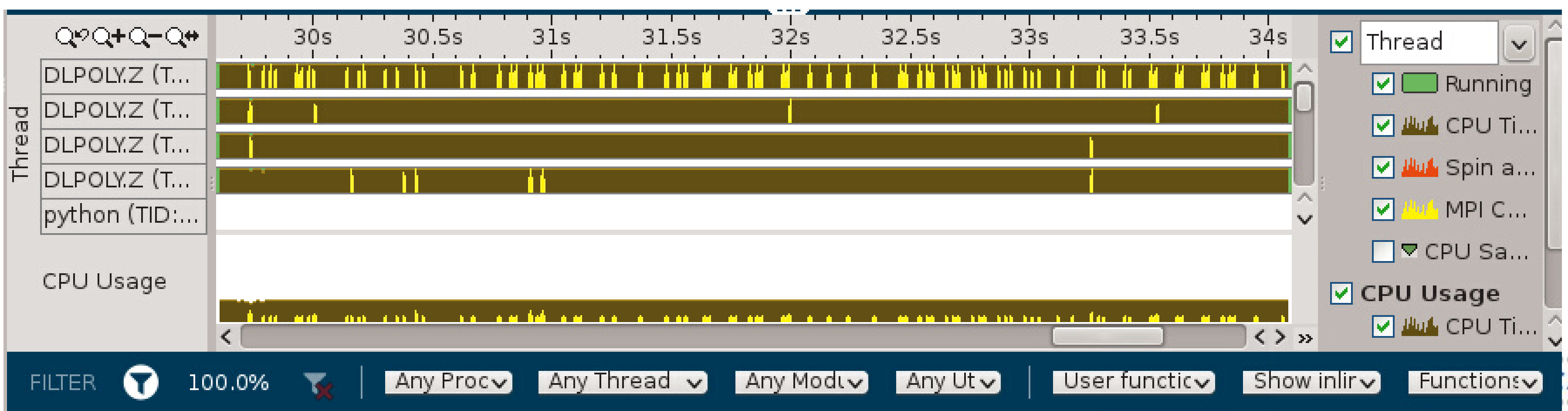
- Comprehensive Intel Performance profiler.
- Best used in interactive mode of PBS.

```
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



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



Extrae and paraver

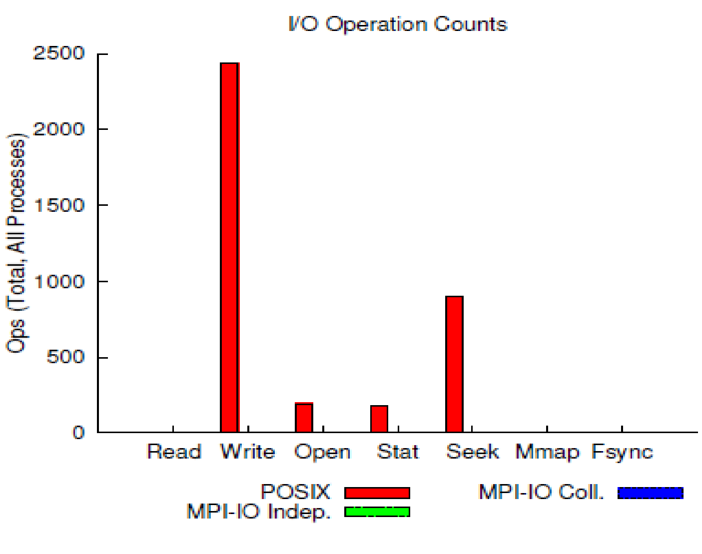
- Profiling package developed by Barcelona Supercomputing Centre (BSC). Extrae inserts “probes” into the application to produce trace files which can be read by paraver.
- Available for a wide range of platforms, incl. ARM and Xeon PHI.

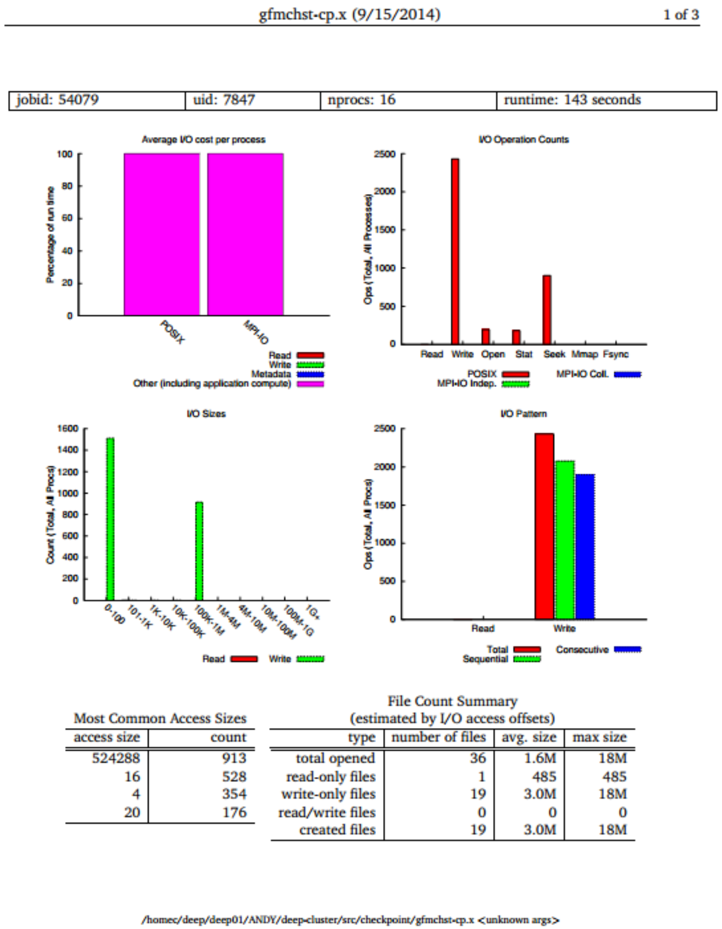
Supported programming models	Supported platforms
MPI	Linux clusters (x86 and x86-64)
OpenMP*	BlueGene/Q
CUDA*	Cray
OpenCL*	nVidia GPUs
pthread*	Intel Xeon Phi
OmpSs*	ARM
Java	Android
Python	

```
LD_PRELOAD=${EXTRAE_HOME}/lib/lib
mpitracef.so
mpirun -env LD_PRELOAD ./mympi
```

I/O performance

- Not so many user-level profiling tools for I/O (file write and read profiling mainly done by sysadmins).
- One example is Darshan.




gfmchst-cp.x (9/15/2014) 1 of 3

jobid: 54079	uid: 7847	nprocs: 16	runtime: 143 seconds

Most Common Access Sizes

access size	count
524288	913
16	528
4	354
20	176

File Count Summary
(estimated by I/O access offsets)

type	number of files	avg. size	max size
total opened	36	1.6M	18M
read-only files	1	485	485
write-only files	19	3.0M	18M
read/write files	0	0	0
created files	19	3.0M	18M

/homec/deep/deep01/ANDY/deep-cluster/src/checkpoint/gfmchst-cp.x <unknown args>

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