



Profilers

Tools for performance analysis

Nikolaos Nikoloutsakos

GRNET

Athens, 11 - 12 Dec. 2019



Contents

1 Introduction to application performance

2 Performance Tools

- Manual Methods
- Gprof
- VTUNE
- Scalasca

3 Conclusions



- Efficient programming on **HPC** architectures is difficult because we have to deal with:
 - ▶ diverse systems
 - ▶ complex hardware architecture , memory hierarchies, cpus, multiple filesystems(local, parallel fs)
 - ▶ network topologies
 - ▶ accelerators
 - ▶ parallel paradigms
 - ▶ software builds
- It is essential to measure performance in order to optimise:
 - ▶ **Developers**: compiler options, parallelization, communication, ...
 - ▶ **Users**: choose the best build, library, software, scalability, ...



- **Profiler** is a software that gets metrics on source execution, without addition of timers in source code.
- *Serial Profilers*
 - ▶ One can find detailed time spent in code procedures, i.e. How many times a procedure was called, average time per call, total time spent in procedure, from which point in source was called etc.
 - ▶ Standard Unix profiler **gprof** and its variants, for example **sprof**.
 - ▶ Compiler specific profilers, like **vtune** for Intel compilers or **pgprof** for PGI.
- *MPI*
 - ▶ **mpiP**: Traces MPI calls and gives performance indicators, possible bottlenecks etc. OpenSource, Works with any compiler and MPI implementation.



- ▶ MPI implementations profilers , for example OpenMPI **VampirTrace**.
- Hybrid MPI,OpenMP,Threads Profilers
 - ▶ **scalasca**: Traces MPI calls, as well as OpenMP calls, provides detailed information, timing information per thread, task, node, code line. Graphical Interface to explore profile information.
- Other mainly commercial profilers,debuggers, for example **DDT**



Performance Metrics

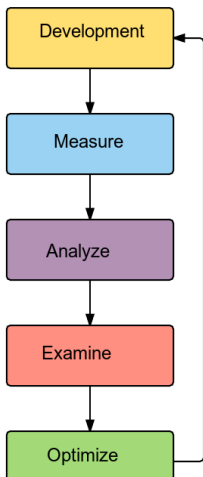
What to measure;

- Execution time
 - ▶ Wall-clock (computation, I/O, system)
 - ▶ CPU time
 - ▶ **non-deterministic** average measurement...
- Number of call functions, time spent in the calls
- CPU cycles per instruction
- FLOPs
- Number of messages, size, bytes transmitted ...
- Energy to solution



Performance Factors

- **Serial** factors
 - ▶ Computations (algorithm, compilation optimizations)
 - ▶ Cache & Main memory (architecture, fine-tuning)
 - ▶ I/O
 - ▶ Hardware
- **Parallel** factors
 - ▶ Partitioning, Distribution
 - ▶ Communication (message passing)
 - ▶ Multithreading, Load balancing, Amdahl's Law
 - ▶ Synchronization/ Locking



- Performance is an iterative process
- Measure is better than guess
- Find bottlenecks, hot spots
- Compare alternatives
- Validate the results!



Hot Spot

An area of code within the program that uses a disproportionately high amount of processor time.

Bottleneck

An area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays.



When to Stop ?

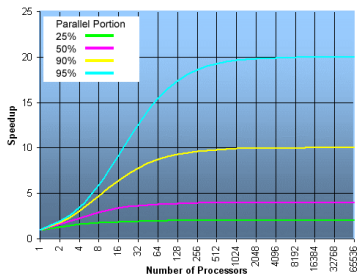
Procedure	CPU Time
<code>main()</code>	13%
<code>procedure1()</code>	17%
<code>procedure2()</code>	20%
<code>procedure3()</code>	50%

- A 20% increase in the performance of `procedure3()` results in a 10% performance increase overall
- A 20% increase in the performance of `main()` results in only a 2.6% performance increase overall.



When to Stop ?

- Amdahl's law states that $speedup = \frac{1}{\frac{P}{N} + S}$
P = parallel fraction, N = number of processors, S = serial fraction
- It soon becomes obvious that are limits to the scalability of parallelism



“Famous” quote: You can spend a lifetime getting 95% of your code to be parallel, and never achieve better than 20x speedup no matter how many processors you throw at it!



Measuring execution time without source code

- **time**

```
time ./a.out
real    0m32.658s
user    0m32.628s
sys     0m0.002s
```

- **date**

```
start=$(date +%s)
./a/out
end=$(date +%s)
echo $(( $end-$start ))
```



Time (parallel program)

```
OMP_NUM_THREADS=2 time ./mmm
```

```
5.45user 0.01system 0:02.75elapsed 198%CPU  
(0avgtext+0avgdata 25128maxresident)k  
0inputs+0outputs (0major+2280minor)pagefaults 0swaps
```

- "user" time is keeping cpus busy
- percentage of CPU usage is 200% ?
- there is no I/O
- no page faults
- maximum resident set size of the program



Measure execution time within program

Fortran 90

```
integer :: counti, countf, count_rate  
real    :: dt  
call system_clock (counti,count_rate)  
... work ....  
call system_clock (countf)  
dt=REAL(countf-counti)/REAL(count_rate)
```



C

```
static double get_second (void){\newline
struct timeval tv;
gettimeofday(&tv, NULL);
return (double)tv.tv_sec +
        (double)tv.tv_usec / 1000000.0; } //1.e6

start = get_second();
...work...
stop = get_second();
double totaltime = stop - start;
```



MPI, openMP

```
double precision :: t1,t2,dt
...
t1 = MPI_WTIME()
... work ...
t2 = MPI_WTIME()
deltat = t2-t1
```

```
double t1, t2;
t1 = MPI_WTIME();
...work...
t2 = MPI_WTIME();
deltat = t2-t1;
```

```
openMP t1 = omp_get_wtime()
```



Profiling using Gprof

Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing

- gprof GNU Profiler
- sourceware.org/binutils/docs/gprof/
- only user space (no information on library routines eg. BLAS, MKL, ...)
- not for parallel programming



Gprof

Setup Compile and Link

```
gcc [flags] -g <source file> -o <output file> -pg
```

Error

```
gprof: gmon.out file is missing call-graph data
```



Gprof

```
gprof options [executable-file  
[profile-data-files...]] [> outfile]
```

● Output parameters

- ▶ -b doesn't print the verbose blurbs
- ▶ -p print a flat profile
- ▶ -p<func name> function specific
- ▶ -P suppress flat profile
- ▶ -q, -Q call graph only

● Analyze parameters

- ▶ -a : suppress the printing of statically declared (private) functions.



Gprof

```
gcc -Wall -pg test_gprof.c -o test_gprof
```

```
gprof test_gprof gmon.out > analysis.txt
```

```
%           the percentage of the total running time of the  
time        program used by this function.  
  
cumulative a running sum of the number of seconds accounted  
seconds    for by this function and those listed above it.  
  
self       the number of seconds accounted for by this  
seconds    function alone. This is the major sort for this  
           listing.  
  
calls      the number of times this function was invoked if
```



Gprof

```
Flat profile:
```

```
Each sample counts as 0.01 seconds.
```

%	cumulative	self	self	self	total	name
time	seconds	seconds	calls	s/call	s/call	
33.74	10.52	10.52	1	10.52	20.96	func1
33.51	20.97	10.45	1	10.45	10.45	func2
33.48	31.42	10.44	1	10.44	10.44	new_func1
0.16	31.47	0.05				main

```
Call graph (explanation follows)
```

```
granularity: each sample hit covers 2 byte(s) for 0.03% of 31.47 seconds
```

index	% time	self	children	called	name
[1]	100.0	0.05	31.42		<spontaneous>
		10.52	10.44	1/1	main [1]
					func1 [2]



Gprof

```
gprof -P -b test_gprof gmon.out
```

```
Flat profile:
```

```
Each sample counts as 0.01 seconds.
```

%	cumulative	self	self	self	total	total	name
time	seconds	seconds	calls	s/call	s/call	s/call	
102.69	10.52	10.52	1	10.52	10.52	10.52	func1

- shows relations between subroutines and functions and time used.



Intel VTune I

- Intel's VTune profiling tool for scalar, multi-threaded (and MPI...) applications
- <http://software.intel.com/en-us/intel-vtune-amplifier-xe>
- *Although VTune can be used with MPI programs, its functionality is aimed at analysing multi-threaded programs.*
- best used in interactive environment.



Intel VTune II

User Interface

amplxe-gui

Command Line

amplxe-cl



Command Line Environment

```
amplxe-cl -help
```

```
amplxe-cl -help collect
```

```
amplxe-cl -collect hotspots -result-dir mydir  
home/test/sample
```

```
amplxe-cl -R summary -r mydir
```



Intel VTune

Available parameters

- advanced-hotspots: **Advanced Hotspots**
- bandwidth: **Bandwidth**
- concurrency: **Concurrency**
- hotspots: **Basic Hotspots**
- locksandwaits: **Locks and Waits**



Intel VTune

```
amplxe-gui r000hs/r000hs.amplxe
```

The screenshot shows the Intel VTune Amplifier XE 2015 interface. The window title is "<no current project> - Intel VTune Amplifier (on login01)". The main area displays "Basic Hotspots" for the project "r000hs". The interface includes a navigation bar with options like "Collection Log", "Analysis Target", "Analysis Type", "Summary", "Bottom-up", "Caller/Callee", "Top-down Tree", and "Tasks and".

Elapsed Time: 12.860s

- CPU Time: 180.829s**
 - Effective Time: 175.169s**
 - Spin Time: 5.660s**
 - Overhead Time: 0s**
- Total Thread Count: 16
- Paused Time: 0s

OpenMP Analysis. Collection Time: 12.860s

- Serial Time (outside any parallel region): 0.127s (1.0%)
- Parallel Region Time: 12.733s (99.0%)**
 - Estimated Ideal Time: 10.950s (85.1%)
 - OpenMP Potential Gain: 1.784s (13.9%)



Intel VTune

<no current project> - Intel VTune Amplifier (on login01)

Welcome **r000hs** x

Basic Hotspots

 Hotspots by CPU Usage viewpoint (change) ⓘ Intel VTune Amplifier XE 2015

Collection Log Analysis Target Analysis Type Summary Bottom-up Caller/Callee Top-down Tree Tasks and

Grouping: Function / Call Stack

Function / Call Stack	CPU Time		S Ti.	O Ti.	Module
	Effective Time by Utilization				
	Idle	Poor	Ok	Ideal	Over
▼ [Loop at line 182 in	175.059s		0s	0s	matrix.icc [Loop at line 182
↳ [Loop at line 18	175.059s		0s	0s	matrix.icc [Loop at line 181
▼ [Loop at line 181 in	0.080s		0s	0s	matrix.icc [Loop at line 181
↳ [Loop at line 180	0.080s		0s	0s	matrix.icc [Loop at line 180
▼ [Loop at line 144 in	0.030s		0s	0s	matrix.icc [Loop at line 144
↳ [Loop at line 144	0.030s		0s	0s	matrix.icc [Loop at line 144

Selected 1 row(s): 175.059s 0s 0s

OMP Wor... 1s 2s 3s 4s 5s 6s 7s 8s 9s 10s 11s 12s

Ruler Area Regio...

Data Of Interest (CPU Metrics)
★ Viewing 1 of 1 selected stack
100.0% (175.059s of 175.059s)
matrix.icc! [Loo...9] - multiply.c
matrix.icc! [Loo... multiply.c:182
matrix.icc! [Loo... multiply.c:182
libc-2.12.so! [Lo...ibc-start.c:226
matrix.icc! [Ou...own]: [Unknown]



Intel VTune

Intel VTune Amplifier (on login01)

Basic Hotspots Hotspots by CPU Usage viewpoint (change) Intel VTune Amplifier XE 2015

Collection Log Analysis Target Analysis Type Summary Bottom-up Caller/Callee Top-down Tree Tasks and

Source Assembly Assembly grouping: Address

S. Li.	Source	CPU Time
174	void multiply1(int msize, int tid, int numt, TYPE a[]	0.0%
175	{	0.0%
176	int i,j,k;	0.0%
177		0.0%
178	// Basic parallel implementation	0.0%
179	#pragma omp parallel for	0.0%
180	for(i=0; i<msize; i++) {	0.0%
181	for(j=0; j<msize; j++) {	0.0%
182	for(k=0; k<msize; k++) {	205.7%
183	c[i][j] = c[i][j] + a[i][k] * b[k][j];	94.1%
184	}	0.0%

Data Of Interest (CPU Metrics)

★ Viewing 1 of 1 selected stack

100.0% (175.059s of 175.059s)

- matrix.iccl[Loo...9] - multiply.c
- matrix.iccl[Loo... multiply.c:182
- matrix.iccl[Loo... multiply.c:182
- libc-2.12.so![Lo...ibc-start.c:226
- matrix.iccl[Ou...own]:[Unknown]



Intel VTune

Intel VTune Amplifier XE 2015

Choose Analysis Type

Analysis Type

Advanced Hotspots

Copy

Start

Start Paused

Project Properties

Copy Command Line to Clipboard (on login01)

Command line:

```
/users/apps/compilers/intel/vtune_amplifier_xe_2015.3.0.403110/bin64/amp/xe-cl -collect advanced-hotspots -app-working-dir /users/staff/nikoloutsa/projects/aris_training/profile/vtune/tachyon -- /users/staff/nikoloutsa/projects/aris_training/profile/vtune/tachyon/tachyon_find_hotspots /users/staff/nikoloutsa/projects/aris_training/profile/vtune/tachyon/dat/balls.dat
```

Copy

Use -collect-with action

Hide knobs with default values

This command line can be used to collect data on a remote machine. To do



Intel VTune

```
#!/bin/bash
```

```
#SBATCH --job-name=vtune
```

```
#SBATCH --partition=compute
```

```
#SBATCH --nodes=1
```

```
#SBATCH --cpus-per-task=16
```

```
#SBATCH --ntasks-per-node=1
```

```
#SBATCH --time=00:10:00
```

```
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK
```

```
srun ampxe-cl -r myresult -quiet -collect advanced-  
hotspots ./matrix.icc
```



Scalasca

- *Scalable performance analysis of large-scale parallel applications*
- <http://www.scalasca.org>
- Open source
- **ScoreP**: instrumentation, mpi/shmem, openmp/threads hybrid
- **Cube**: Analysis report
- **Scalasca**: trace analysis

module load scalasca



Scalasca 2.x

scalasca

Toolset **for** scalable performance analysis of large-scale parallel applications

usage: scalasca [OPTION]... ACTION <argument>...

1. **prepare application** objects and executable **for** measurement:
scalasca -instrument <compile-or-link-command> # skin (using scorep)
2. **run application** under control of measurement system:
scalasca -analyze <application-launch-command> # scan
3. interactively explore measurement **analysis report**:
scalasca -examine <experiment-archive|report> # square



Summary Report I

(1) Instrumentation **scorep**

```
# compile AND link with scorep
```

```
MPIF77 = scalasca -instrument mpif77
```

Check scorep env

```
scorep-info config-vars -full
```



Summary Report II

(2) Run **scan** (scalasca -analyze)

```
export
```

```
SCOREP_EXPERIMENT_DIRECTORY=scorep_bt-mz_W_4x4_sum
```

```
scan mpirun -np 4 ./bt-mz_W.4
```

(3) Analysis report **square** (flat summary)

```
square -s scorep_bt-mz_W_4x4_sum
```

```
scorep-score scorep_bt-mz_W_4x4_sum/profile.cubex
```

```
scorep-score -r scorep_bt-mz_W_4x4_sum/profile.cubex
```



Summary Report III

type	max_buf[B]	visits	time[s]	time[%]	time/visit[us]	region
ALL	277,805,118	41,158,333	37.01	100.0	0.90	ALL
USR	274,792,492	40,418,321	14.82	40.0	0.37	USR
OMP	6,882,860	685,952	21.71	58.7	31.65	OMP
COM	377,156	46,744	0.19	0.5	4.07	COM
MPI	102,286	7,316	0.29	0.8	39.68	MPI
USR	85,774,338	12,516,672	5.32	14.4	0.43	binvcrhs_
USR	85,774,338	12,516,672	3.84	10.4	0.31	matvec_sub_
USR	85,774,338	12,516,672	4.57	12.4	0.37	matmul_sub_
USR	7,974,876	1,170,624	0.43	1.2	0.36	binvrhs_
USR	7,974,876	1,170,624	0.46	1.2	0.39	lhsinit_
USR	3,473,912	526,848	0.20	0.5	0.39	exact_solution
OMP	410,040	25,728	0.01	0.0	0.52	!\$omp parallel
OMP	410,040	25,728	0.01	0.0	0.52	!\$omp parallel
OMP	410,040	25,728	0.01	0.0	0.53	!\$omp parallel
OMP	410,040	25,728	0.01	0.0	0.54	!\$omp parallel

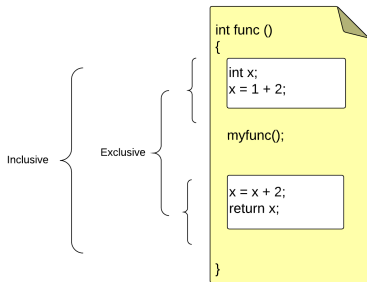


Analysis report examination GUI

```
cube scorep_bt-mz_W_4x4_sum/profile.cubex  
square scorep_bt-mz_W_4x4_sum  
scalasca -examine scorep_bt-mz_W_4x4_sum
```

The screenshot displays the analysis report examination GUI with the following components:

- File Display Plugins Help** menu bar.
- Restore Setting Save Settings** toolbar.
- Metric tree** panel (Absolute):
 - 4.12e7 Visits (occ)
 - 37.01 Time (sec)
 - 0.00 Minimum Inclusive Time (sec)
 - 2.41 Maximum Inclusive Time (sec)
 - 0 bytes_put (bytes)
 - 0 bytes_get (bytes)
 - 3.07e7 bytes_sent (bytes)
 - 3.07e7 bytes_received (bytes)
- Call tree** panel (Absolute):
 - 4.27e5 initialize_
 - 6.88e4 exact_rhs_
 - 88 timer_clear_
 - 1704 exch_qbc_
 - 2.00e5 adi_
 - 8 MPI_Barrier
 - 4 timer_start_
 - 800 ITERATION
 - 3.41e5 exch_qbc_
 - 3200 adi_
 - 2.21e5 compute_rhs_
- System tree** panel (Absolute):
 - machine Linux
 - node login0
 - MPI Rank 0
 - 1 Master thread
 - 0 OMP thread 1
 - 0 OMP thread 2
 - 0 OMP thread 3
 - MPI Rank 1
 - 1 Master thread
 - 0 OMP thread 1
 - 0 OMP thread 2



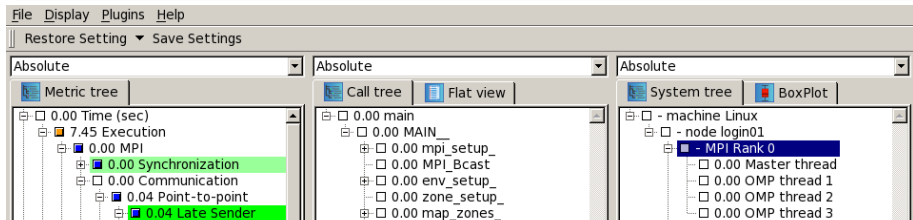
- **Inclusive** Time is the amount of time spent in a function and all child functions, i.e. the time from start of the function up until it's return.
- **Exclusive** Time is the amount of time spent purely in that function, excluding time spent in child functions

Scalasca event trace analysis

```
scan -t
```

As of time-averaged summaries, it is possible to generate also time-stamped event traces.

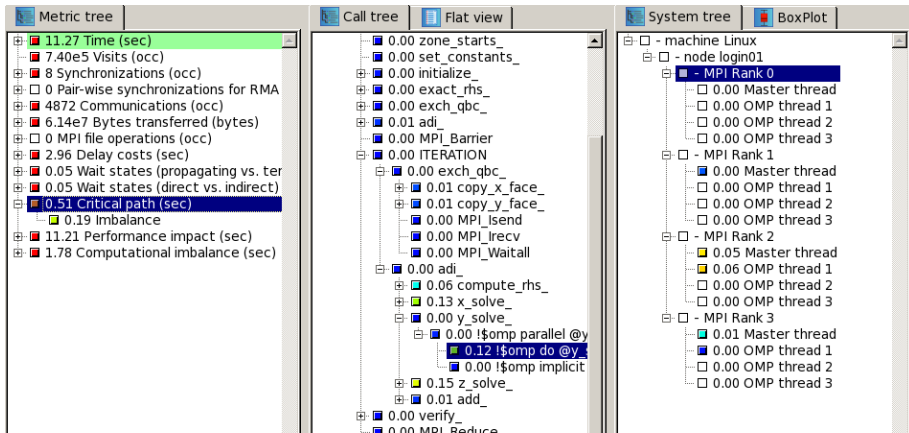
- Profiling: Summarization of events over execution interval
- Tracing: Chronologically ordered sequence of event records



The screenshot displays the Scalasca GUI interface with three main panels:

- Metric tree:** Shows a hierarchical view of execution metrics. The root is '0.00 Time (sec)', which branches into '7.45 Execution', '0.00 MPI', '0.00 Synchronization', '0.00 Communication', and '0.04 Point-to-point'. Under '0.04 Point-to-point', there is a sub-entry '0.04 Late Sender'.
- Call tree:** Shows a hierarchical view of function calls. The root is '0.00 main', which branches into '0.00 MPI_Bcast', '0.00 env_setup_', '0.00 zone_setup_', and '0.00 map_zones_'.
- System tree:** Shows a hierarchical view of the system tree. The root is '- machine Linux', which branches into '- node login01', which further branches into '- MPI Rank 0'. Under '- MPI Rank 0', there are four entries: '0.00 Master thread', '0.00 OMP thread 1', '0.00 OMP thread 2', and '0.00 OMP thread 3'.

Critical-path





Conclusions

- Is there a performance issue?
 - ▶ time, speedup, scalability
- What is the basic cause of lag? (calculation, communication, memory)
 - ▶ MPI/ OpenMP / flat profiling
- Where is the lag?
 - ▶ Call-path
- Why is it there?
 - ▶ hardware count analysis
- Scalability issues
 - ▶ load imbalance, compare profiles various sizes function by function