Code profiling on Graham

Sergey Mashchenko
(SHARCNET / Compute Ontario / Compute Canada)

Outline

- Introduction
- Simple profiling
- Profiling serial, MPI and OpenMP codes with MAP
- CUDA profiling
- Questions?

Introduction

What is profiling?

- Profiling is the task of timing a code.
- It used used primarily as a part of the iterative process of improving the efficiency (reducing the wallclock runtime) of the code.
- It is often done using simple means (like inserting time measurement lines in your code), but for serious profiling work one has to use dedicated profiling tools.

Simple profiling

Timing the whole code

- On SHARCNET clusters, one can use the Linux shell command “time” to time the whole code.

  $ time ./your_code , or
  $ time mpirun -np 32 ./your_mpi_code

- This has to be done on an empty node, to improve the accuracy of timing.
- A node can be reserved with salloc command (gives interactive access to compute resources for up to 3 hours), e.g.

  $ salloc --time=0-03:00 -c 32 -A def-**account** --mem=120000M
  (for serial and multi-threaded codes), and

  $ salloc --time=0-03:00 -n 32 -A def-**account** --mem-per-cpu=4000M
  (for MPI codes).

Timing the whole code (cont.)

- On Graham, cpu cores take non-negligible time to spin up from the idle state (1200 MHz) to the maximum speed under full load (2600 MHz).
- As a result, one has to time the code multiple times in a loop, choosing the best timing, e.g.:

```
$ for ((i=0; i<10; i++)); do { time -p ./code ; } |& grep real ; done |sort -k 2 -gr
```

- This is obviously not ideal. A much better way is to place timers inside your code, to time specific parts of the code.
- This again should be ideally done in an internal (code) loop, to eliminate the cpu spin-up effect.

Timers inside your code

- gettimeofday: high precision (10μs) cpu-based timer (just google for the function timeval_subtract).

```
#include <sys/time.h>
...
struct timeval  tdr0, tdr1, tdr;
double delta_t;
gettimeofday (&tdr0, NULL);

  < The code to time >

gettimeofday (&tdr1, NULL);
tdr = tdr0;
timeval_subtract (&delta_t, &tdr1, &tdr);
printf ("Time: %e\n", delta_t);
```

Timers inside your code (cont.)

- **OpenMP code:** omp_get_wtime() can be used to time (in seconds) both entire parallel regions, or individual threads inside a parallel region.

```
#include <omp.h>
...
double t1 = omp_get_wtime();

   < The code to time >

double t2 = omp_get_wtime();
printf ("Time: %e\n", t2-t1);
```

- This can also be used to time non-OpenMP codes (as it is more convenient than gettimeofday), just don't forget to add the “#include <omp.h>” line, and compile the code with “-qopenmp” (icc) or “-fopenmp” (gcc) switches.

Timers inside your code (cont.)

- **MPI code:** MPI_Wtime() can be used the same way as omp_get_wtime() on the previous slide:

```
#include <mpi.h>
...
double t1 = MPI_Wtime();

   < The code to time >

double t2 = MPI_Wtime();
printf ("Time: %e\n", t2-t1);
```

Timers inside your code (cont.)

- **CUDA code:** to time a specific CUDA kernel, the best approach is to use CUDA events:

```
cudaEvent_t start, stop;
float time;
cudaEventCreate (&start);
cudaEventCreate (&stop);
cudaEventRecord (start, 0);
kernel_to_time <<<grid, threads>>> ();
cudaEventRecord (stop, 0);
cudaEventSynchronize (stop);
cudaEventElapsedTime (&time, start, stop);
cudaEventDestroy (start);
cudaEventDestroy (stop);
```

Timers inside your code (cont.)

- For timing CUDA code consisting of multiple kernels, and/or concurrent GPU and CPU computations, and/or concurrent GPU operations (using streams), one has to use cpu-based timers (like gettimeofday or omp_get_wtime).

Profiling serial, MPI and OpenMP codes with MAP

Overview

- Parallel profiler MAP (along with the parallel debugger DDT) are now a part of the software package Forge.
- The original company behind MAP was Allinea. In 2016 it was acquired by the CPU maker ARM.
- SHARCNET has been using (and paying for) Allinea/ARM products since 2006.
- Though the debugger DDT was a success from the beginning, the Allinea's first attempt at parallel profiling (OPT) was a failure.
- The replacement MAP (came about in 2013; originally only for serial/MPI codes) used a much better approach to parallel profiling, and is now widely used in HPC community.

Overview (cont.)

- On Graham, profiler MAP is provided via module “ddt-cpu” (or aliases “allinea-cpu” and “arm-forge-cpu”).
- The Graham's license is for up to 512 concurrent cpu cores across all users (for both MAP and DDT).
- Niagara cluster (operated by SciNet) has a smaller license (up to 128 cpu cores). Neither MAP nor DDT are available on Cedar.

How to use MAP

- MAP (and DDT) are GUI applications, so one has to enable X11 forwarding in the SSH connection to be able to use them.
  - One has to add “-Y” switch to the usual ssh command:

    ```
    $ ssh -Y user@graham.computecanada.ca
    ```

  - Windows users: use free software MobaXterm, which comes with both SSH client and X window server (required for X11 forwarding).
  - Mac users: install free app XQuartz (has X windows server).
  - Linux users: everything you need is already installed on your box.
- As an alternative, one could use VNC connection to graham (google for “VNC compute canada” for details).
  - It has a better performance, but takes longer to set up.

How to use MAP (cont.)

- When using the X11 forwarding method, you need to add “--x11” switch to your salloc command, e.g.

  $ salloc **--x11** --time=0-03:00 -c 32 -A def-account --mem=120000M

- After allocating the node(s) with salloc, load the module:

  $ module load ddt-cpu/7.1

- The code to be profiled has to be compiled with “-g” switch which adds symbolic information to the binary. You should use all your usual optimization flags (e.g. -O2).
- Run the code under MAP like this:

  $ map ./your_code <optional code arguments>

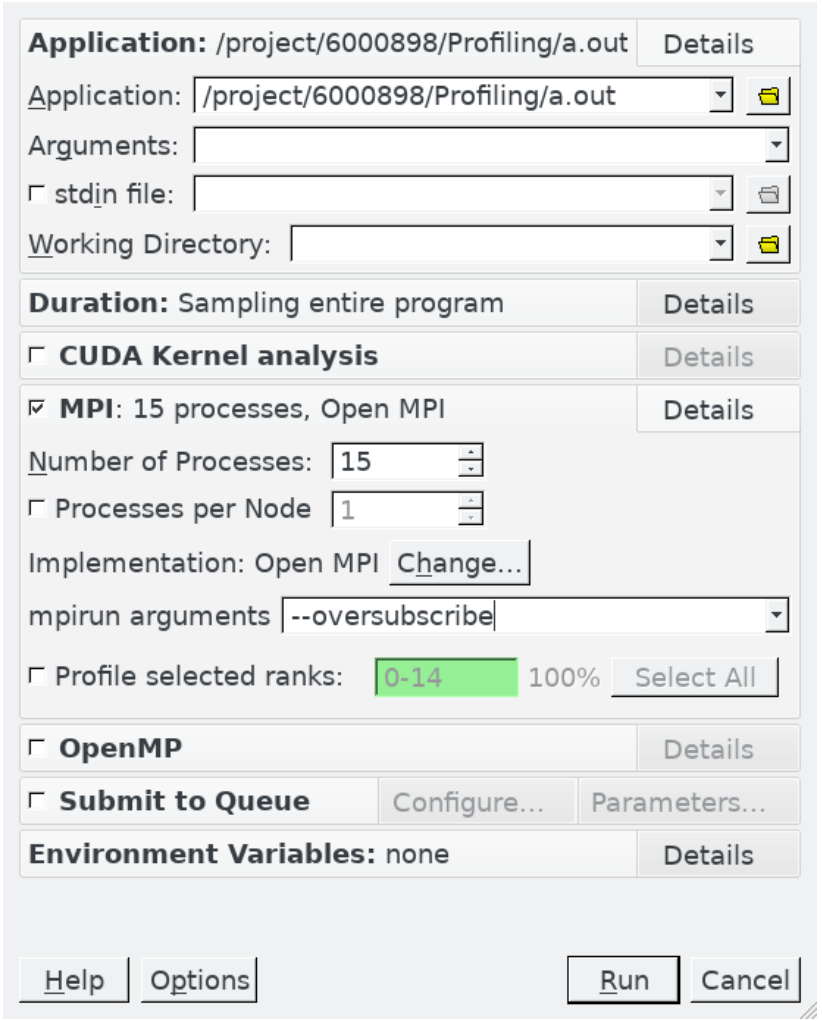
Some details

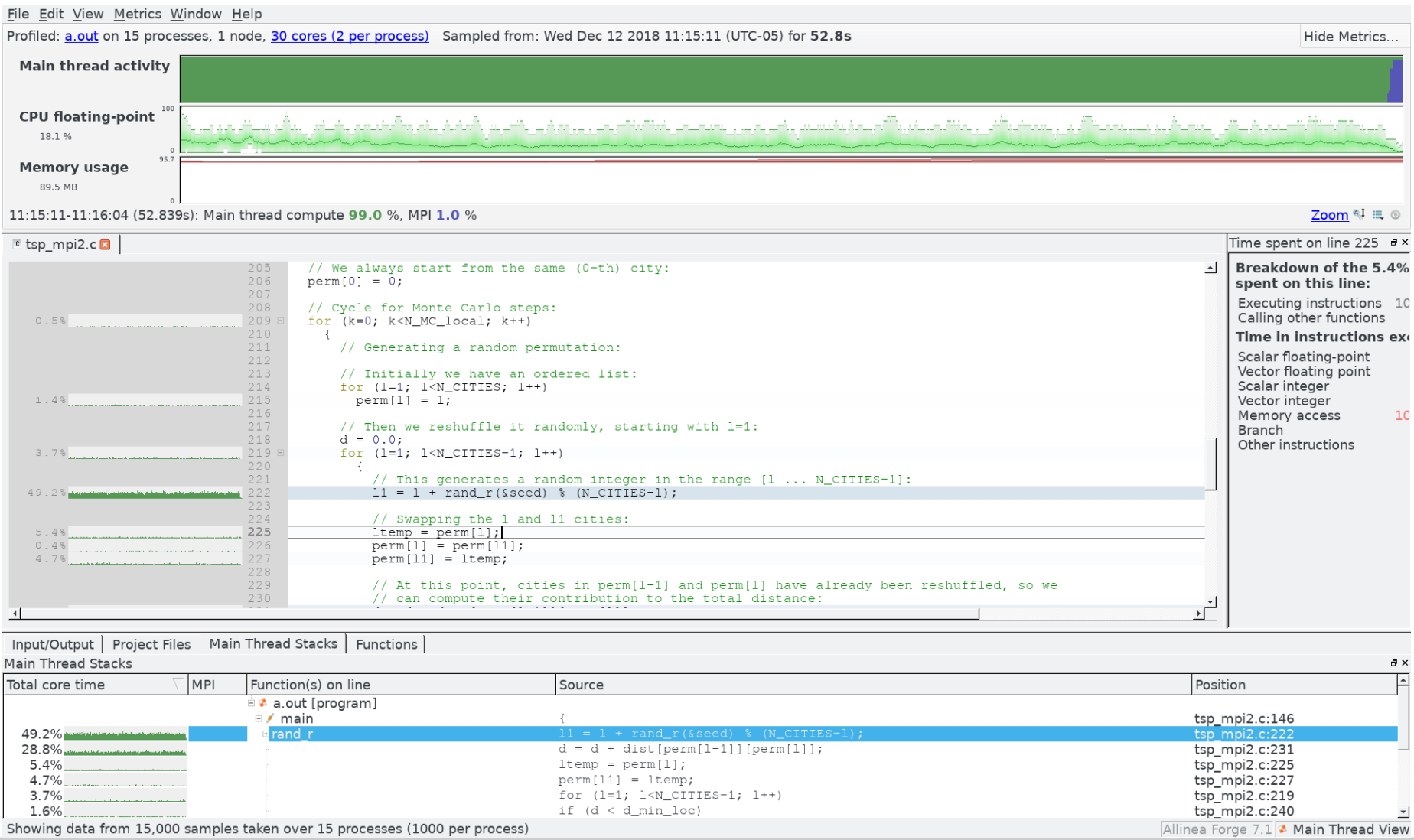
- Use the 7.1 version for now, as the newer one has some issues.
- Request one more cpu core than your code needs with salloc, as MAP uses one cpu core at 100% inside salloc session (it is likely a bug; we'll try to fix it).
- If you need more resources than available with salloc (>3h runtime, or hundreds of cpu cores), submit the MAP session as a job, e.g.

```
module load ddt-cpu/7.1
map  --profile  -n 16  ./code
```

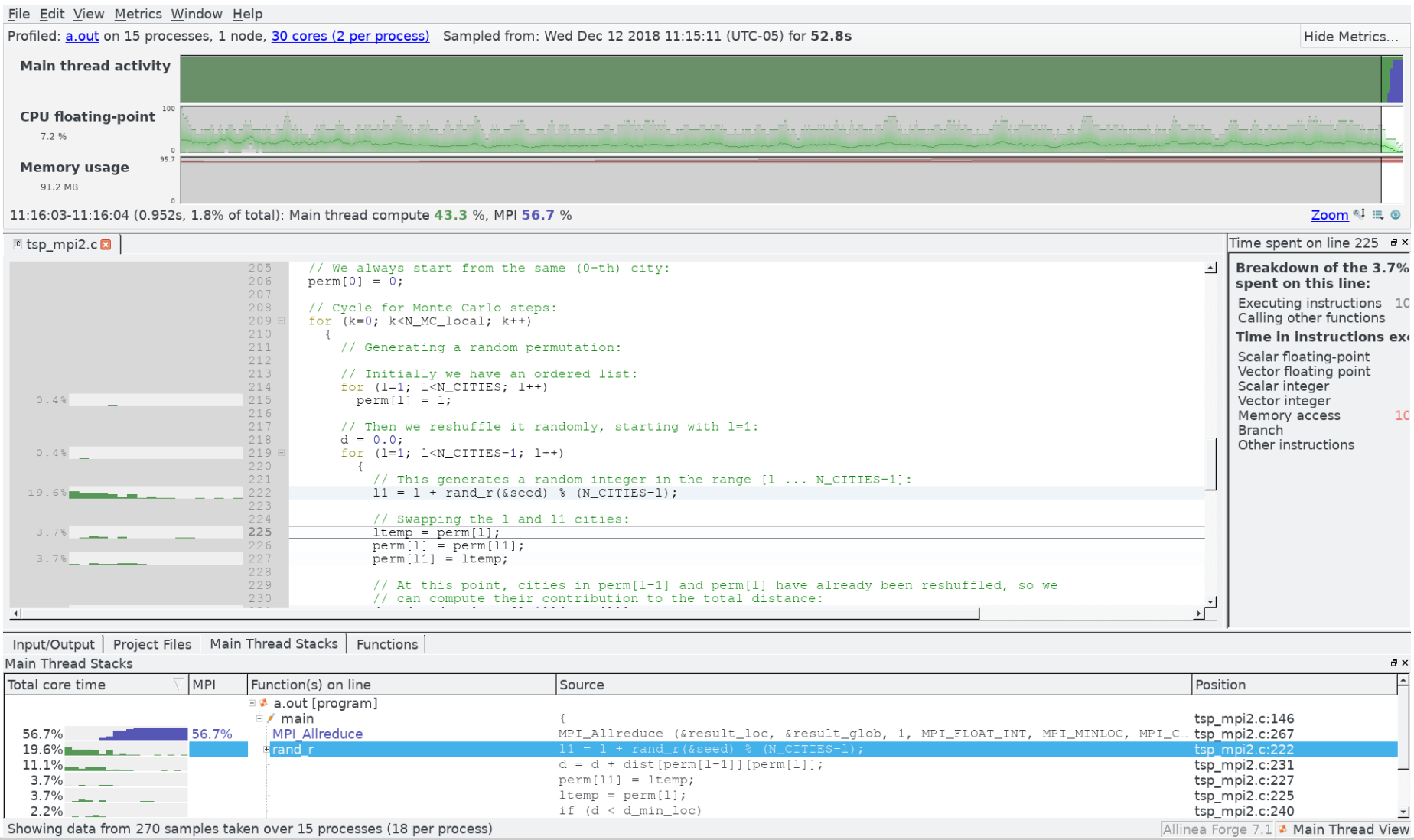
- The profiling results (*.map files) can be analyzed offline with MAP:

```
$ map  results.map
```

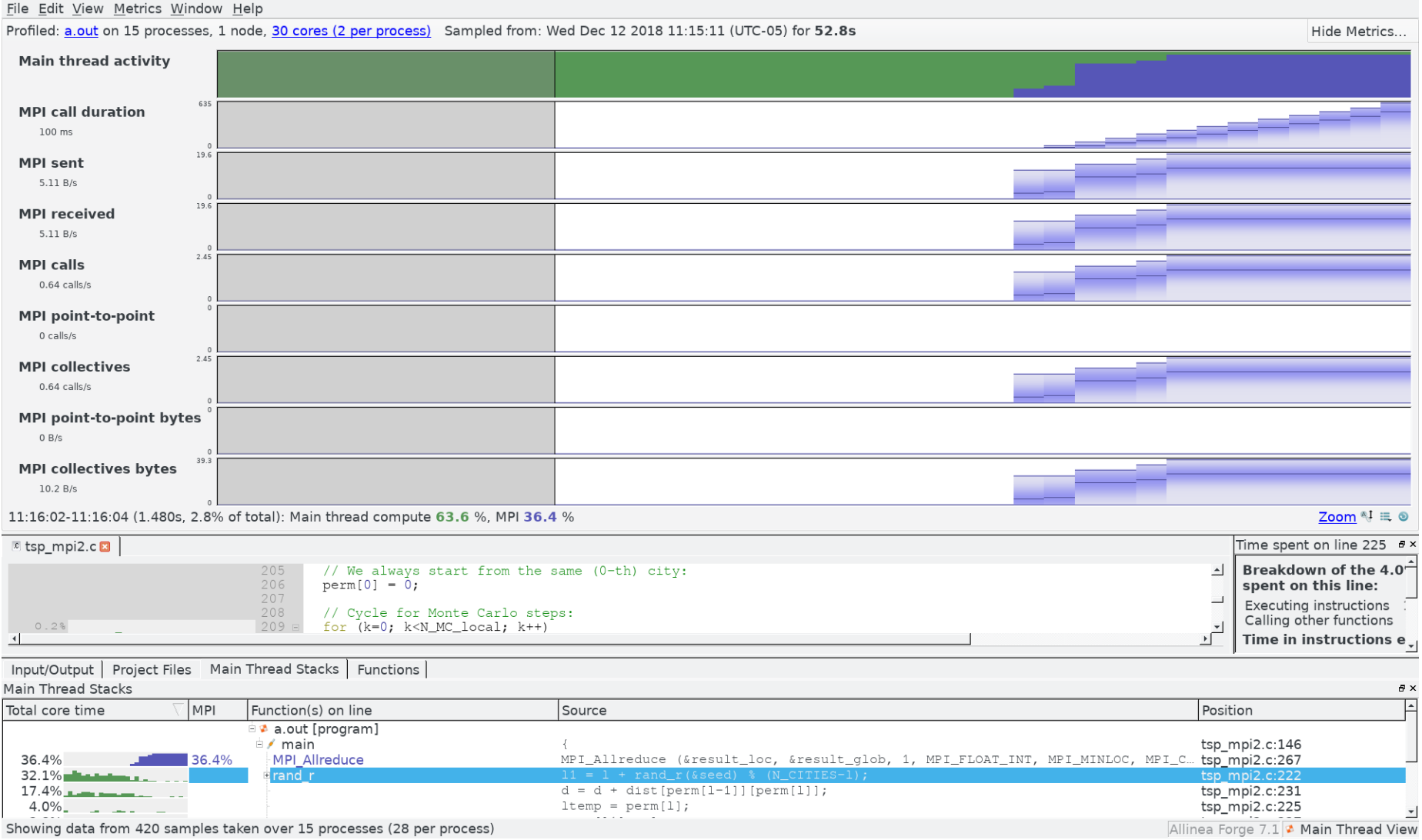

Application: /project/6000898/Profiling/a.out
Details
Application: /project/6000898/Profiling/a.out
Arguments:
stdin file:
Working Directory:
Duration: Sampling entire program
Details
CUDA Kernel analysis
Details
MPI: 15 processes, Open MPI
Details
Number of Processes: 15
Processes per Node 1
Implementation: Open MPI
Change...
mpirun arguments --oversubscribe
Profile selected ranks: 0-14 100%
Select All
OpenMP
Details
Submit to Queue
Configure...
Parameters...
Environment Variables: none
Details
Help
Options
Run
Cancel



Default interface


File Edit View Metrics Window Help
Profiled: a.out on 15 processes, 1 node, 30 cores (2 per process) Sampled from: Wed Dec 12 2018 11:15:11 (UTC-05) for 52.8s
Hide Metrics...
Main thread activity
CPU floating-point
7.2 %
Memory usage
91.2 MB
11:16:03-11:16:04 (0.952s, 1.8% of total): Main thread compute 43.3 %, MPI 56.7 %
Zoom
tsp_mpi2.c
// We always start from the same (0-th) city:
perm[0] = 0;
// Cycle for Monte Carlo steps:
for (k=0; k<N_MC_local; k++)
// Generating a random permutation:
// Initially we have an ordered list:
for (l=1; l<N_CITIES; l++)
perm[l] = l;
// Then we reshuffle it randomly, starting with l=1:
d = 0.0;
for (l=1; l<N_CITIES-1; l++)
// This generates a random integer in the range [l ... N_CITIES-1]:
l1 = l + rand_r(&seed) % (N_CITIES-l);
// Swapping the l and l1 cities:
ltemp = perm[l];
perm[l] = perm[l1];
perm[l1] = ltemp;
// At this point, cities in perm[l-1] and perm[l] have already been reshuffled, so we
// can compute their contribution to the total distance:
Time spent on line 225
Breakdown of the 3.7% spent on this line:
Executing instructions
Calling other functions
Time in instructions ex
Scalar floating-point
Vector floating point
Scalar integer
Vector integer
Memory access
Branch
Other instructions
Input/Output
Project Files
Main Thread Stacks
Functions
Main Thread Stacks
Total core time
MPI
Function(s) on line
Source
Position
a.out [program]
main
tsp_mpi2.c:146
56.7%
MPI_Allreduce
tsp_mpi2.c:267
19.6%
rand_r
tsp_mpi2.c:222
11.1%
tsp_mpi2.c:231
3.7%
tsp_mpi2.c:227
3.7%
tsp_mpi2.c:225
2.2%
tsp_mpi2.c:240
Showing data from 270 samples taken over 15 processes (18 per process)
Allinea Forge 7.1
Main Thread View

Zooming in


File Edit View Metrics Window Help
Profiled: a.out on 15 processes, 1 node, 30 cores (2 per process) Sampled from: Wed Dec 12 2018 11:15:11 (UTC-05) for 52.8s
Hide Metrics...
Main thread activity
MPI call duration
100 ms
MPI sent
5.11 B/s
MPI received
5.11 B/s
MPI calls
0.64 calls/s
MPI point-to-point
0 calls/s
MPI collectives
0.64 calls/s
MPI point-to-point bytes
0 B/s
MPI collectives bytes
10.2 B/s
11:16:02-11:16:04 (1.480s, 2.8% of total): Main thread compute 63.6 %, MPI 36.4 %
Zoom
tsp_mpi2.c
205 // We always start from the same (0-th) city:
206 perm[0] = 0;
207
208 // Cycle for Monte Carlo steps:
209 for (k=0; k<N_MC_local; k++)
Time spent on line 225
Breakdown of the 4.0
spent on this line:
Executing instructions
Calling other functions
Time in instructions e
Input/Output
Project Files
Main Thread Stacks
Functions
Main Thread Stacks
Total core time
MPI
Function(s) on line
Source
Position
a.out [program]
main { tsp_mpi2.c:146
36.4% 36.4% MPI_Allreduce MPI_Allreduce (&result_loc, &result_glob, 1, MPI_FLOAT_INT, MPI_MINLOC, MPI_C... tsp_mpi2.c:267
32.1% rand_r l1 = l + rand_r(&seed) % (N_CITIES-l); tsp_mpi2.c:222
17.4% d = d + dist[perm[l-1]][perm[l]]; tsp_mpi2.c:231
4.0% ltemp = perm[l]; tsp_mpi2.c:225
Showing data from 420 samples taken over 15 processes (28 per process)
Allinea Forge 7.1
Main Thread View

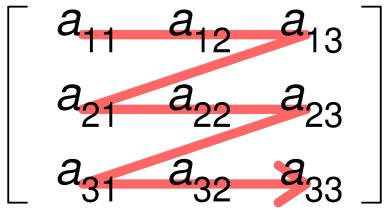
MPI preset

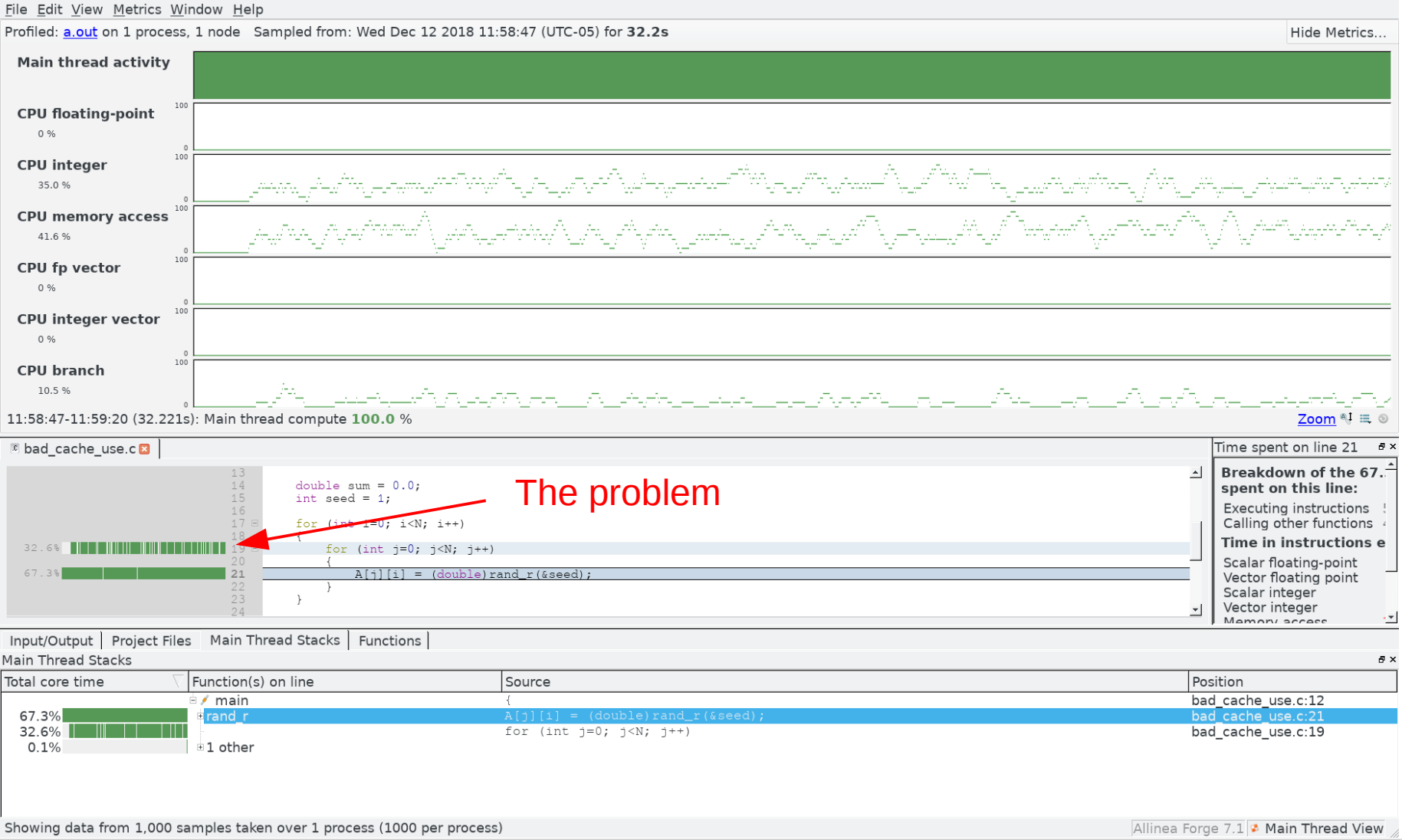
Serial code profiling

- Example: a typical efficiency issue is when a loop in the code reads elements of the vector/array not in the order the data is stored in the memory. This makes CPU-memory caching inefficient.
- For C/C++ codes, data is stored in a row-major order, so it is the last index in multidimensional arrays which should correspond to the inner-most loop:

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        A[i][j] = 0.0;
```

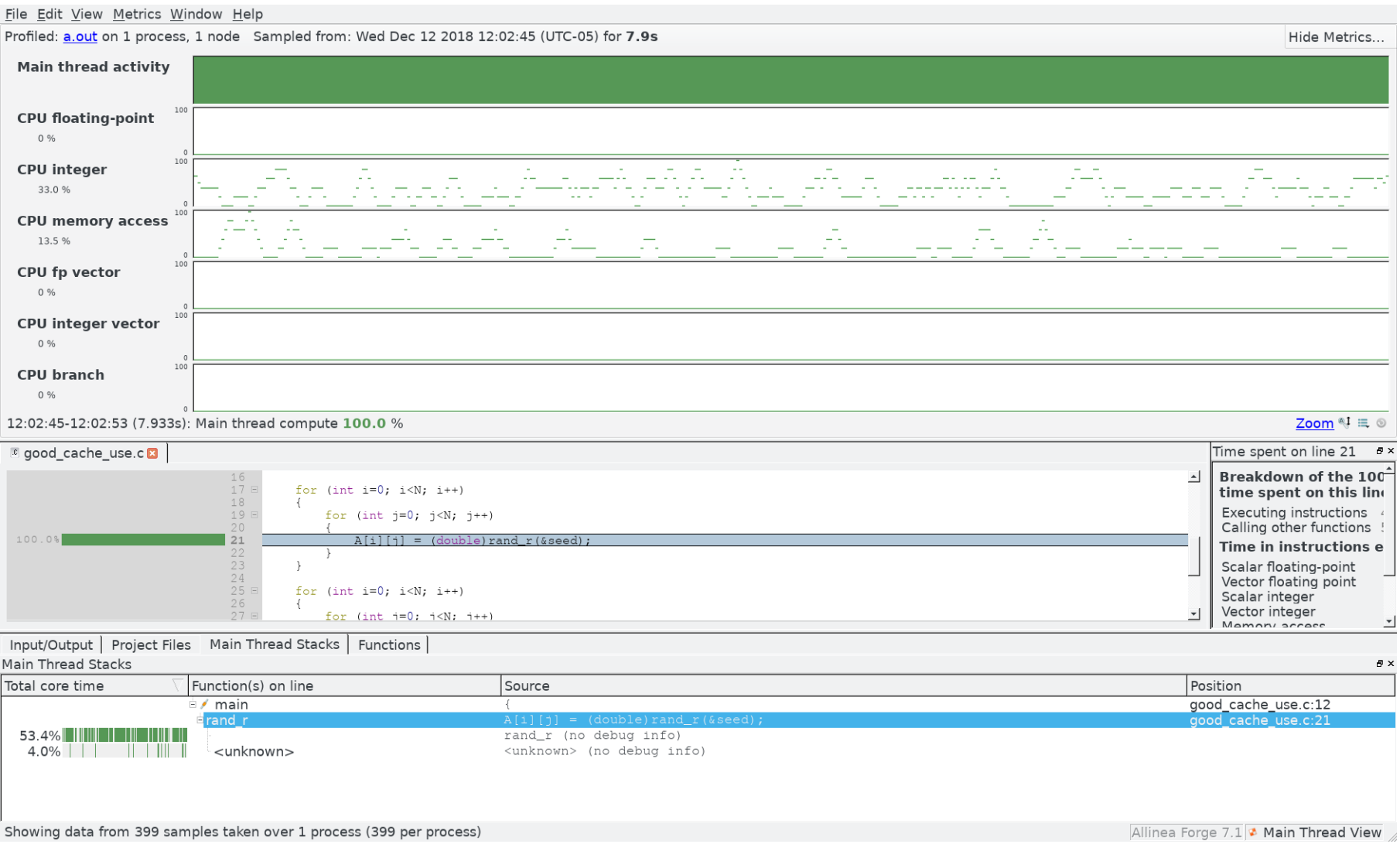
Row-major order





Serial code profiling (CPU instructions preset).

Bad memory access case.



Serial code profiling (CPU instructions preset).

Good memory access case.

Profiling MPI codes

- Dynamic workload balancing (DWB) is frequently used by MPI programs.
- We use it when the length of time spent on computing different parts of a large workload by different MPI ranks is hard or impossible to predict ahead of time.
- Well written DWB code should have a way to adjust the size of the workload quantum. (In other words – number of chunks.)

Dynamic workload balancing example

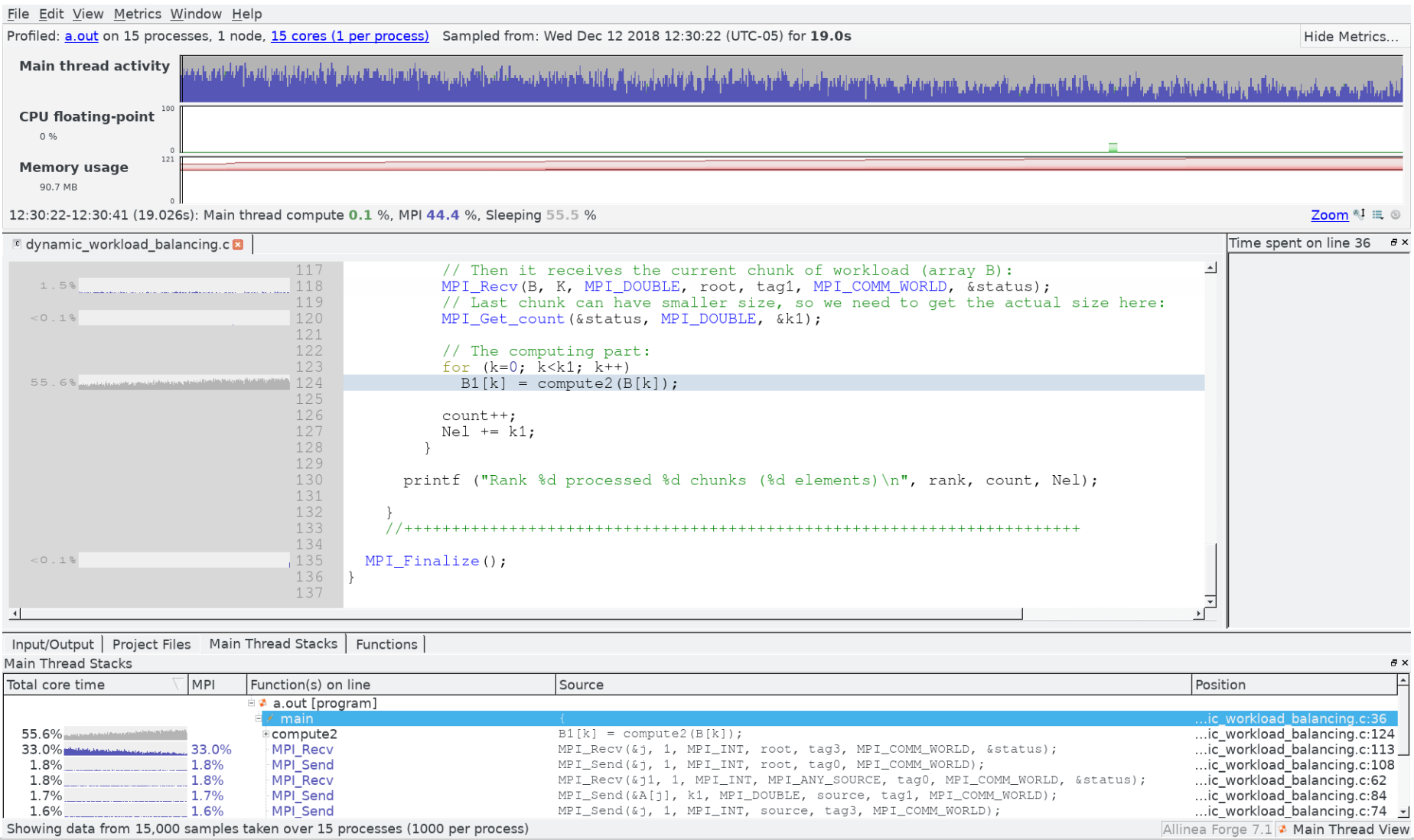
- Example code:
  - dynamic_workload_balancing.c: using “nanosleep” function to emulate different processing time for different elements of a large input array
  - On 15 graham cpu cores, I got the following wall clock times:

N_chunks / N_CPUs	Wall clock time (s)
1	28.8
10	16.7
100	16.2
1000	16.1
200,000	20.6

} Severe workload imbalance

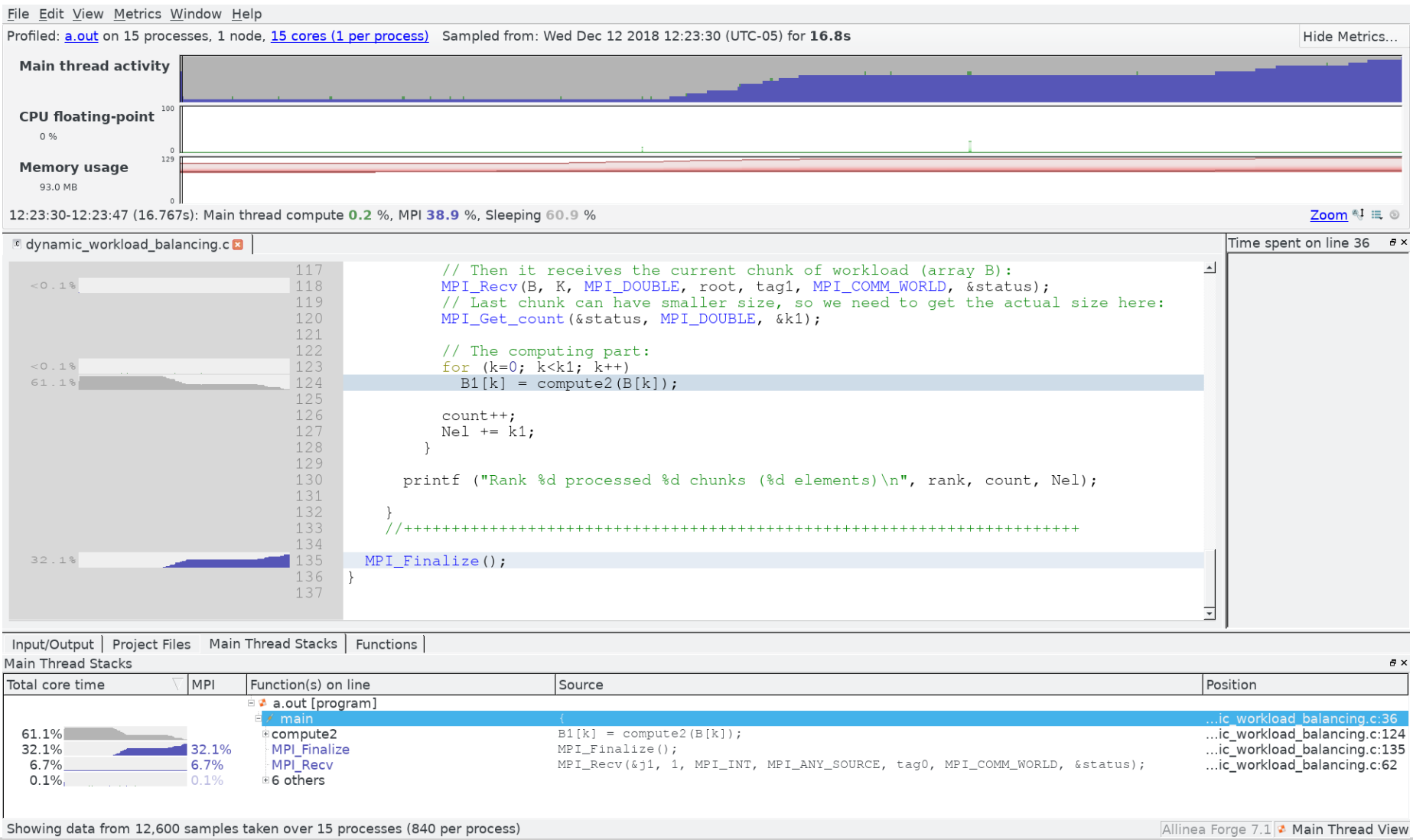
} Optimal performance

} Latency becomes important



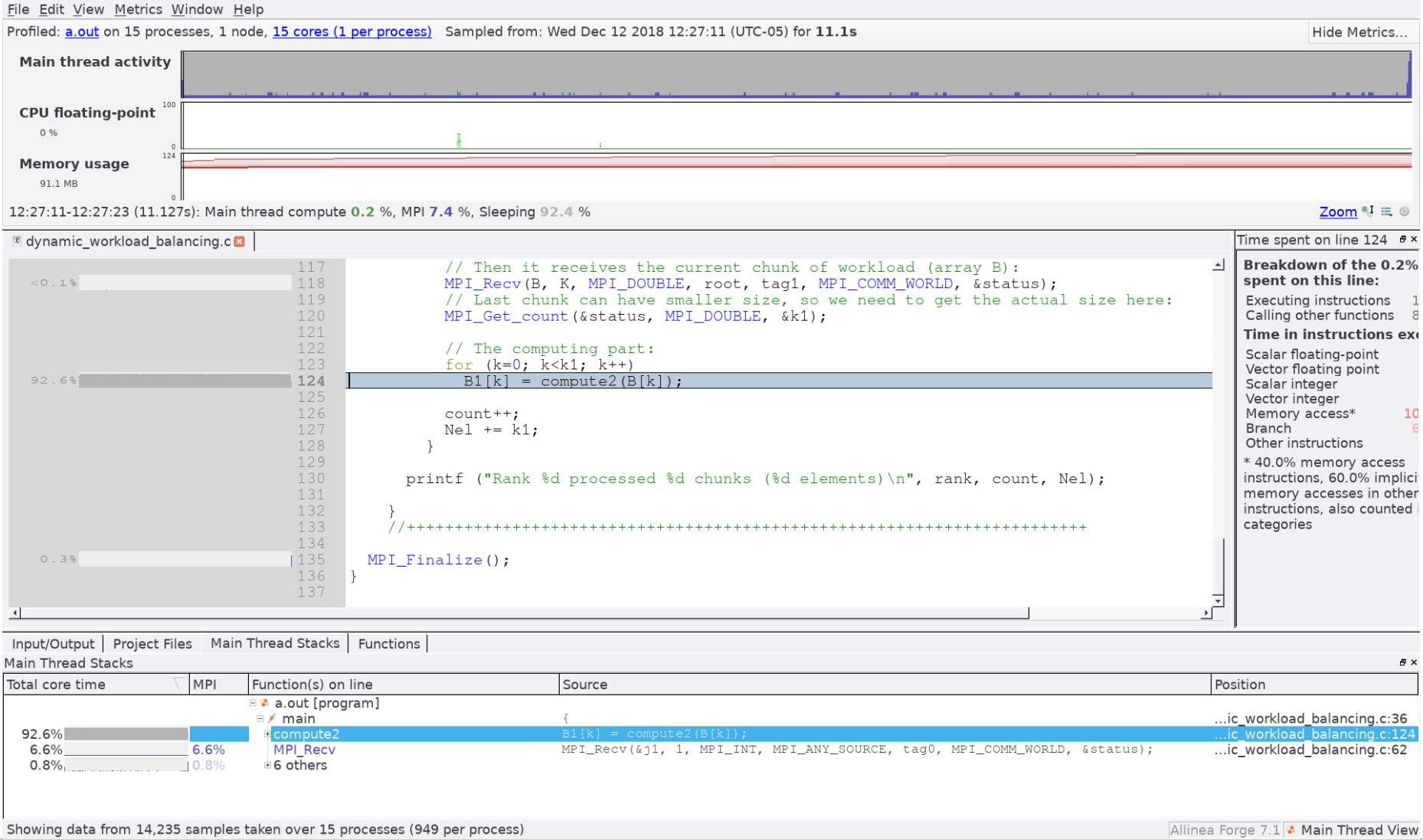
MPI profiling, dynamic workload balancing example.

Chunks are too small = latency dominated.



MPI profiling, dynamic workload balancing example.

Chunks are too large = severe workload imbalance.



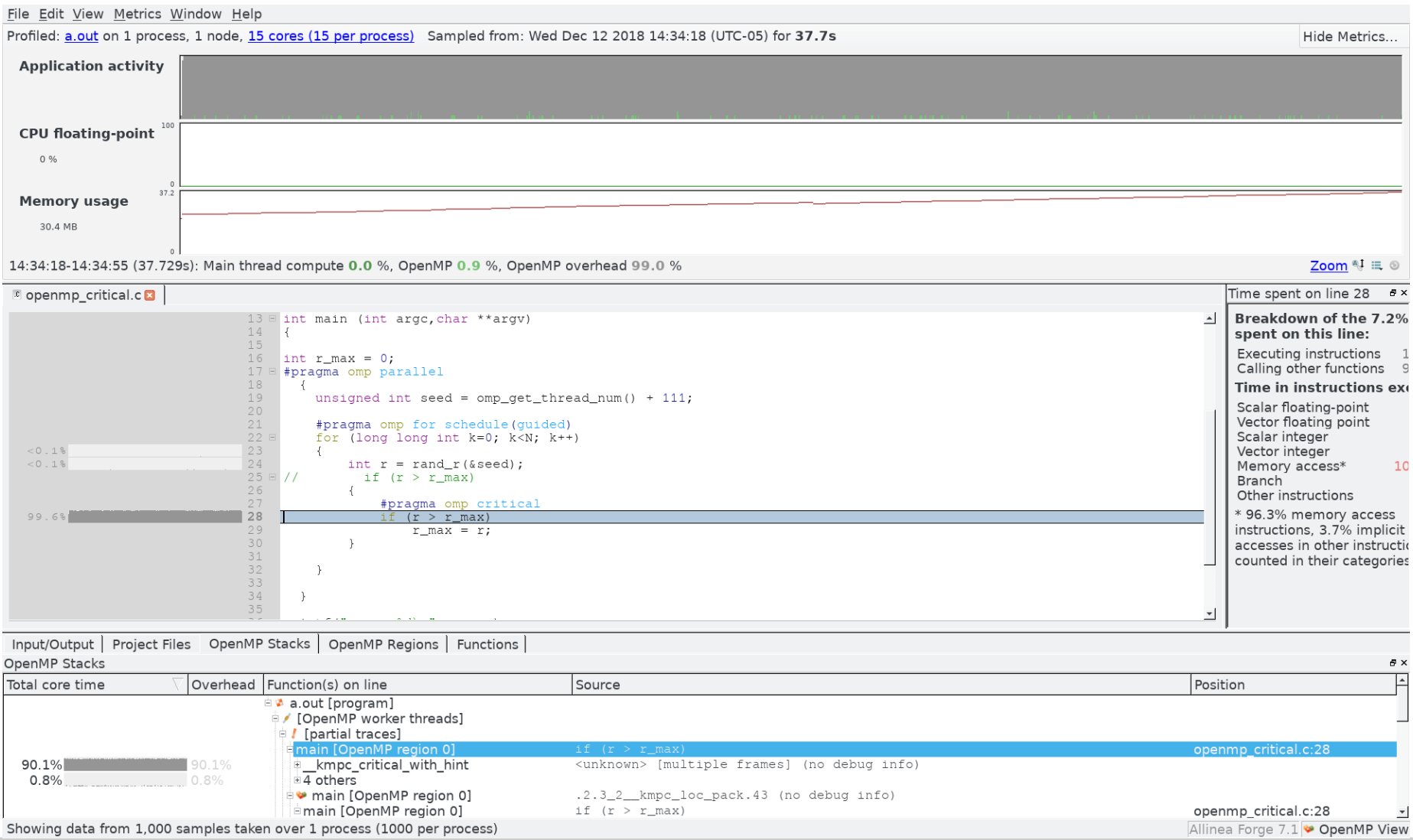
MPI profiling, dynamic workload balancing example.

Chunks have just the right size.

OpenMP profiling

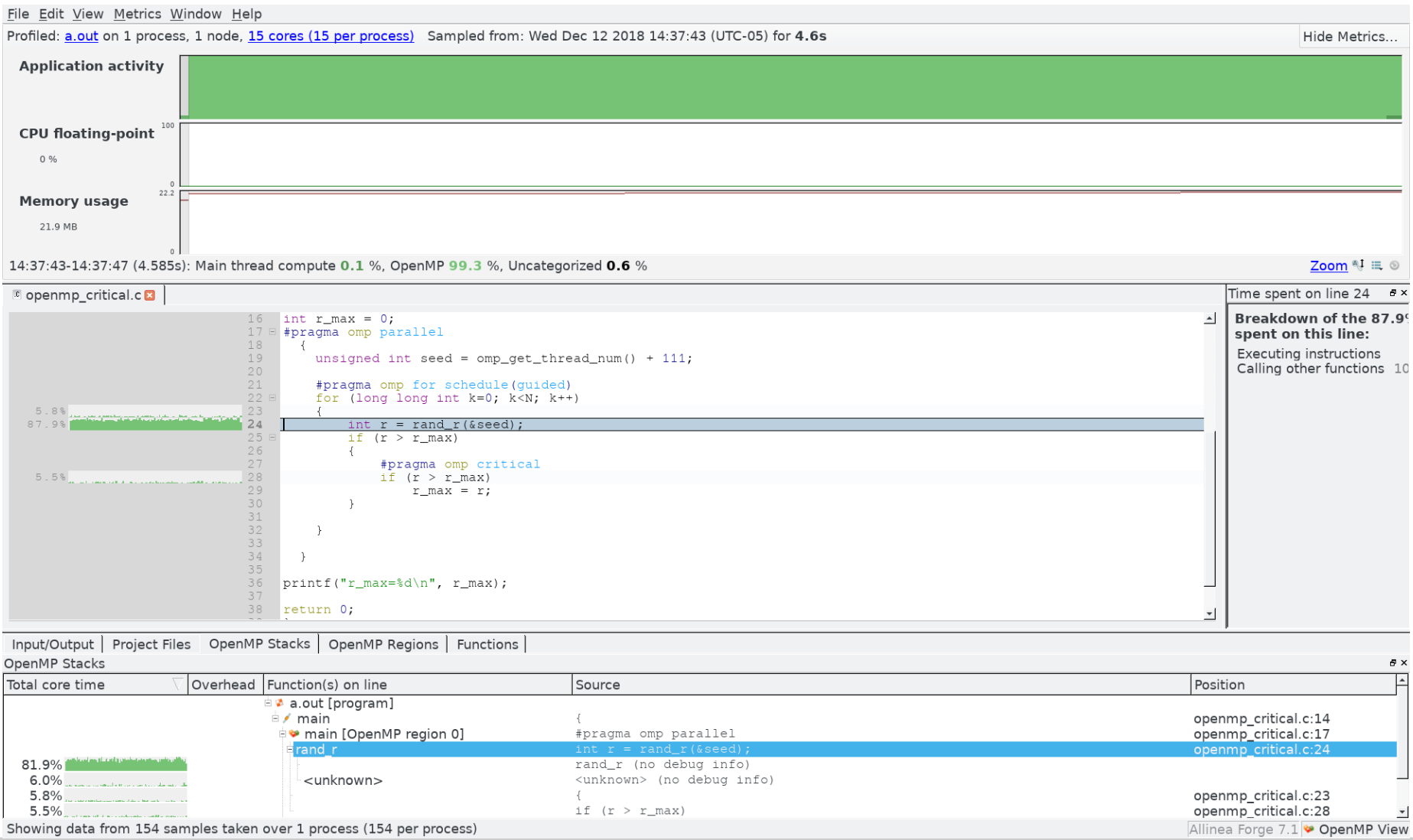
- In OpenMP, critical regions have a significant overhead and should be used sparingly.
- In particular, if used inside a loop for reduction, there should be a pre-selection statement, outside of the critical region:

```
if (x > x_max)   //  Pre-selects only plausible candidates
  #pragma omp critical
    if (x > x_max)   // Very infrequently threads would enter the critical
      x_max = x;   //  region, for the definitive "if" clause application
```



OpenMP profiling, dynamic critical region's impact.

No pre-selection = huge performance hit.



OpenMP profiling, dynamic critical region's impact.

Pre-selection = good performance.

CUDA profiling

- Recently MAP became capable of CUDA (GPGPU) code profiling.
- Unfortunately, SHARCNET's license doesn't cover this feature.
- But we do have Nvidia provided visual profilers for CUDA programs – nvvp and nsight.
- Unfortunately, they don't provide line-by-line kernel analysis (the way MAP does). But they provide plenty of detailed info on kernel performance.
- nvvp and nsight are bundled with cuda modules.

CUDA profiling (cont.)

- No code re-compilation is needed for nvvp profiling.
- Using nvvp interactively (on graham and cedar):

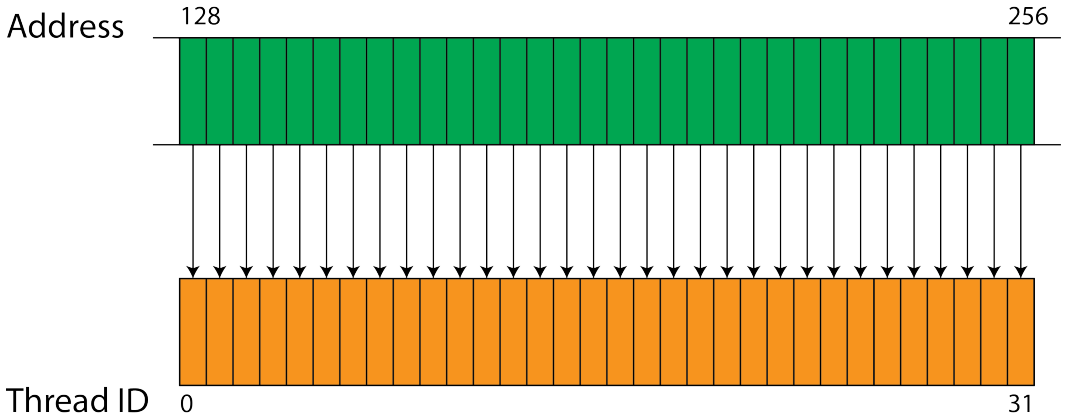
```
$  salloc --x11 --time=0-03:00 --ntasks=1 --gres=gpu:1
-A def-account --mem-per-cpu=4G
$  module  load  cuda/10
$  nvcc  -O2  your_code.c  -o your_code
$  nvvp  ./your_code
```

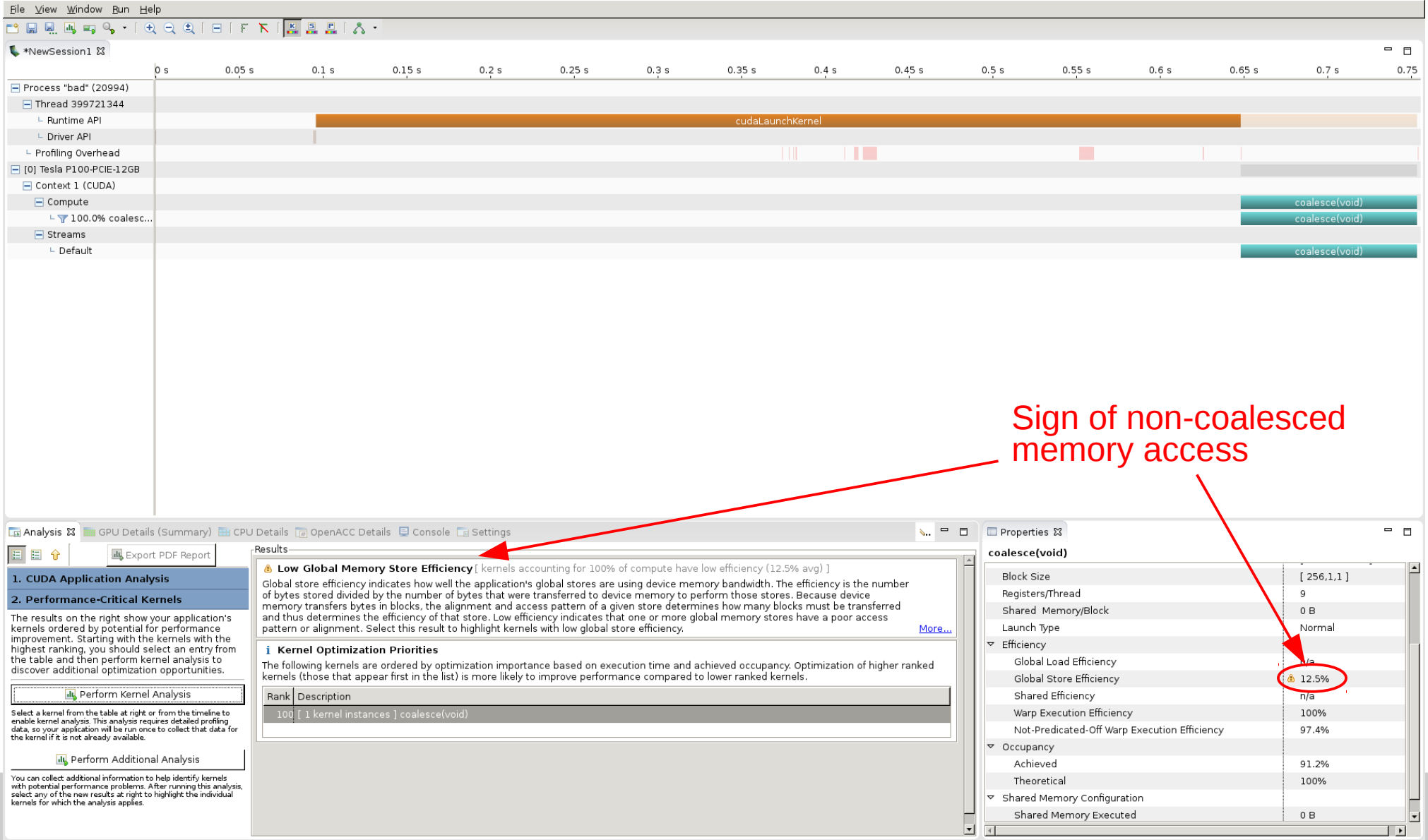
- The app will provide a step-by-step profiling setup. You choose which kernels to profile, and what specific details you need.
- The app will often provide useful descriptive suggestions regarding which parts of your code have efficiency issues.

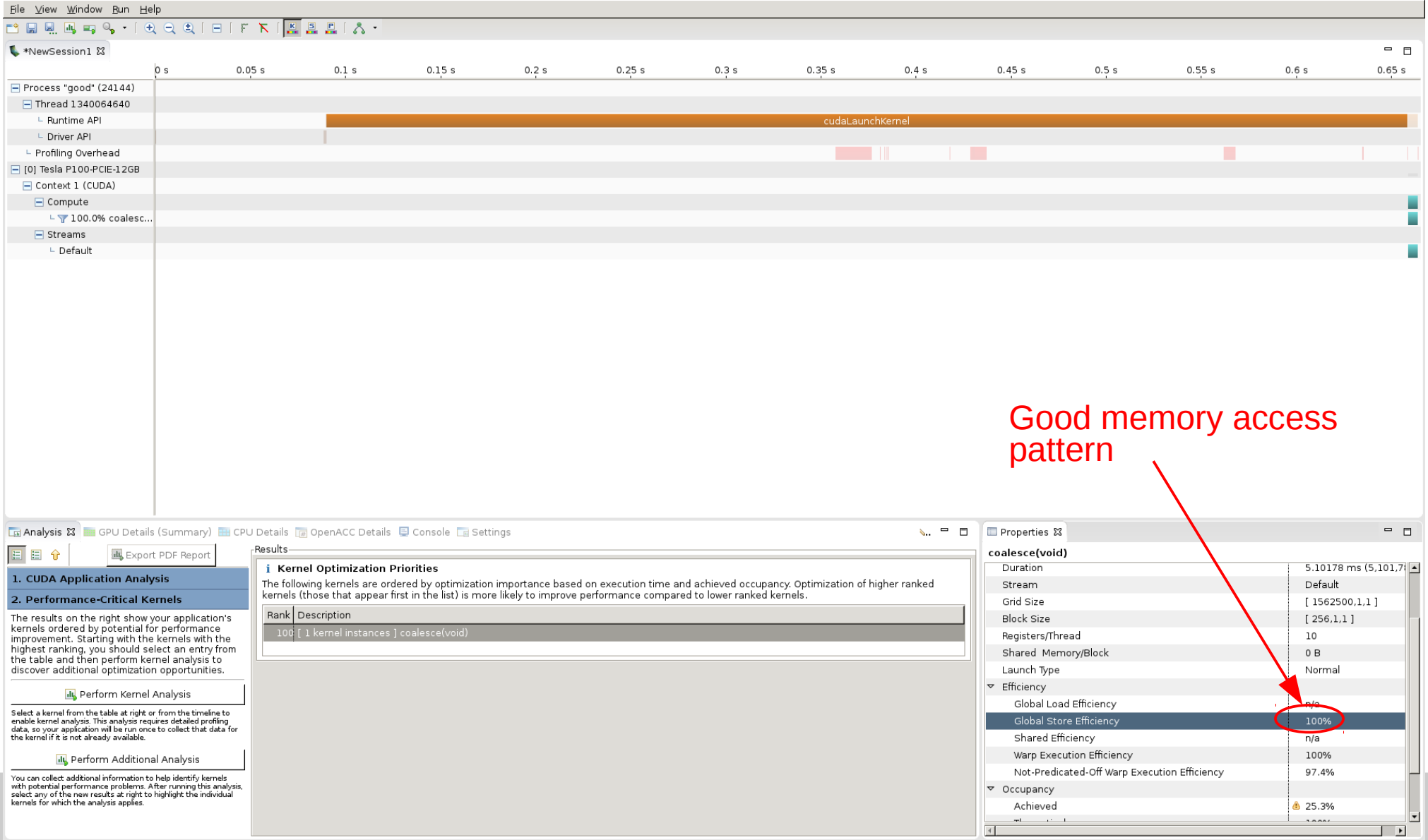
CUDA profiling (cont.)

- Non-coalesced access of the device memory is a significant efficiency issue in GPU programming (similar to the row-major memory access requirement on CPUs).

Good (coalesced) access pattern:




File View Window Run Help
*NewSession1
0 s 0.05 s 0.1 s 0.15 s 0.2 s 0.25 s 0.3 s 0.35 s 0.4 s 0.45 s 0.5 s 0.55 s 0.6 s 0.65 s 0.7 s 0.75
Process "bad" (20994)
Thread 399721344
Runtime API
cudaLaunchKernel
Driver API
Profiling Overhead
[0] Tesla P100-PCIE-12GB
Context 1 (CUDA)
Compute
100.0% coalesc...
Streams
Default
coalesce(void)
coalesce(void)
coalesce(void)
Sign of non-coalesced memory access
Analysis
GPU Details (Summary)
CPU Details
OpenACC Details
Console
Settings
Export PDF Report
1. CUDA Application Analysis
2. Performance-Critical Kernels
The results on the right show your application's kernels ordered by potential for performance improvement. Starting with the kernels with the highest ranking, you should select an entry from the table and then perform kernel analysis to discover additional optimization opportunities.
Perform Kernel Analysis
Select a kernel from the table at right or from the timeline to enable kernel analysis. This analysis requires detailed profiling data, so your application will be run once to collect that data for the kernel if it is not already available.
Perform Additional Analysis
You can collect additional information to help identify kernels with potential performance problems. After running this analysis, select any of the new results at right to highlight the individual kernels for which the analysis applies.
Results
Low Global Memory Store Efficiency [kernels accounting for 100% of compute have low efficiency (12.5% avg)]
Global store efficiency indicates how well the application's global stores are using device memory bandwidth. The efficiency is the number of bytes stored divided by the number of bytes that were transferred to device memory to perform those stores. Because device memory transfers bytes in blocks, the alignment and access pattern of a given store determines how many blocks must be transferred and thus determines the efficiency of that store. Low efficiency indicates that one or more global memory stores have a poor access pattern or alignment. Select this result to highlight kernels with low global store efficiency.
More...
Kernel Optimization Priorities
The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.
Rank
Description
100
[1 kernel instances] coalesce(void)
Properties
coalesce(void)
Block Size
[256,1,1]
Registers/Thread
9
Shared Memory/Block
0 B
Launch Type
Normal
Efficiency
Global Load Efficiency
n/a
Global Store Efficiency
12.5%
Shared Efficiency
n/a
Warp Execution Efficiency
100%
Not-Predicated-Off Warp Execution Efficiency
97.4%
Occupancy
Achieved
91.2%
Theoretical
100%
Shared Memory Configuration
Shared Memory Executed
0 B


File View Window Run Help
*NewSession1
0 s
0.05 s
0.1 s
0.15 s
0.2 s
0.25 s
0.3 s
0.35 s
0.4 s
0.45 s
0.5 s
0.55 s
0.6 s
0.65 s
Process "good" (24144)
Thread 1340064640
Runtime API
cudaLaunchKernel
Driver API
Profiling Overhead
[0] Tesla P100-PCIE-12GB
Context 1 (CUDA)
Compute
100.0% coalesc...
Streams
Default
Good memory access pattern
Analysis
GPU Details (Summary)
CPU Details
OpenACC Details
Console
Settings
Export PDF Report
1. CUDA Application Analysis
2. Performance-Critical Kernels
The results on the right show your application's kernels ordered by potential for performance improvement. Starting with the kernels with the highest ranking, you should select an entry from the table and then perform kernel analysis to discover additional optimization opportunities.
Perform Kernel Analysis
Select a kernel from the table at right or from the timeline to enable kernel analysis. This analysis requires detailed profiling data, so your application will be run once to collect that data for the kernel if it is not already available.
Perform Additional Analysis
You can collect additional information to help identify kernels with potential performance problems. After running this analysis, select any of the new results at right to highlight the individual kernels for which the analysis applies.
Results
Kernel Optimization Priorities
The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.
Rank
Description
100
[1 kernel instances] coalesce(void)
Properties
coalesce(void)
Duration
5.10178 ms (5,101,7
Stream
Default
Grid Size
[1562500,1,1]
Block Size
[256,1,1]
Registers/Thread
10
Shared Memory/Block
0 B
Launch Type
Normal
Efficiency
Global Load Efficiency
n/a
Global Store Efficiency
100%
Shared Efficiency
n/a
Warp Execution Efficiency
100%
Not-Predicated-Off Warp Execution Efficiency
97.4%
Occupancy
Achieved
25.3%

Questions?

- You can always contact me directly (syam@sharcnet.ca) or send an email to help@sharcnet.ca .