Profiling MPI codes with Allinea's MAP

Sergey Mashchenko
McMaster University, SHARCNET

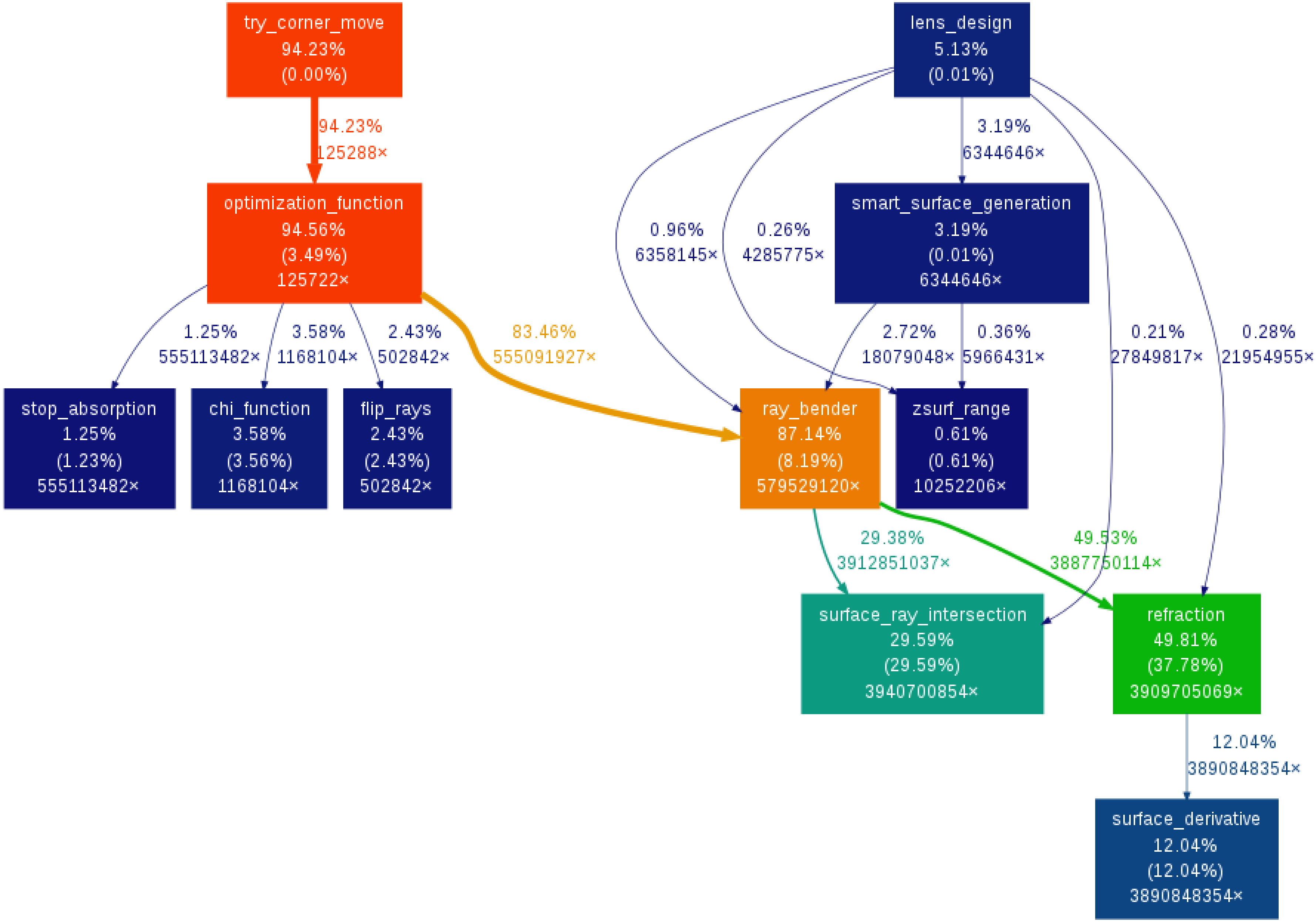
Overview

- Introduction
- Using MAP
- Demonstration

Introduction

- Profiling is an important part of code development
  - Almost as important as debugging; can be considered as “code performance debugging”
  - If writing a new code from scratch, profiling of new code blocks should be done continuously, alongside debugging: “performance bugs” made at the early code development stages will be hard or impossible to fix when the code is finished.
  - If converting a serial code to a parallel one (threads, MPI, CUDA, ...), profiling the serial code can be crucial in guiding the parallelization efforts.

Example: serial code profiling



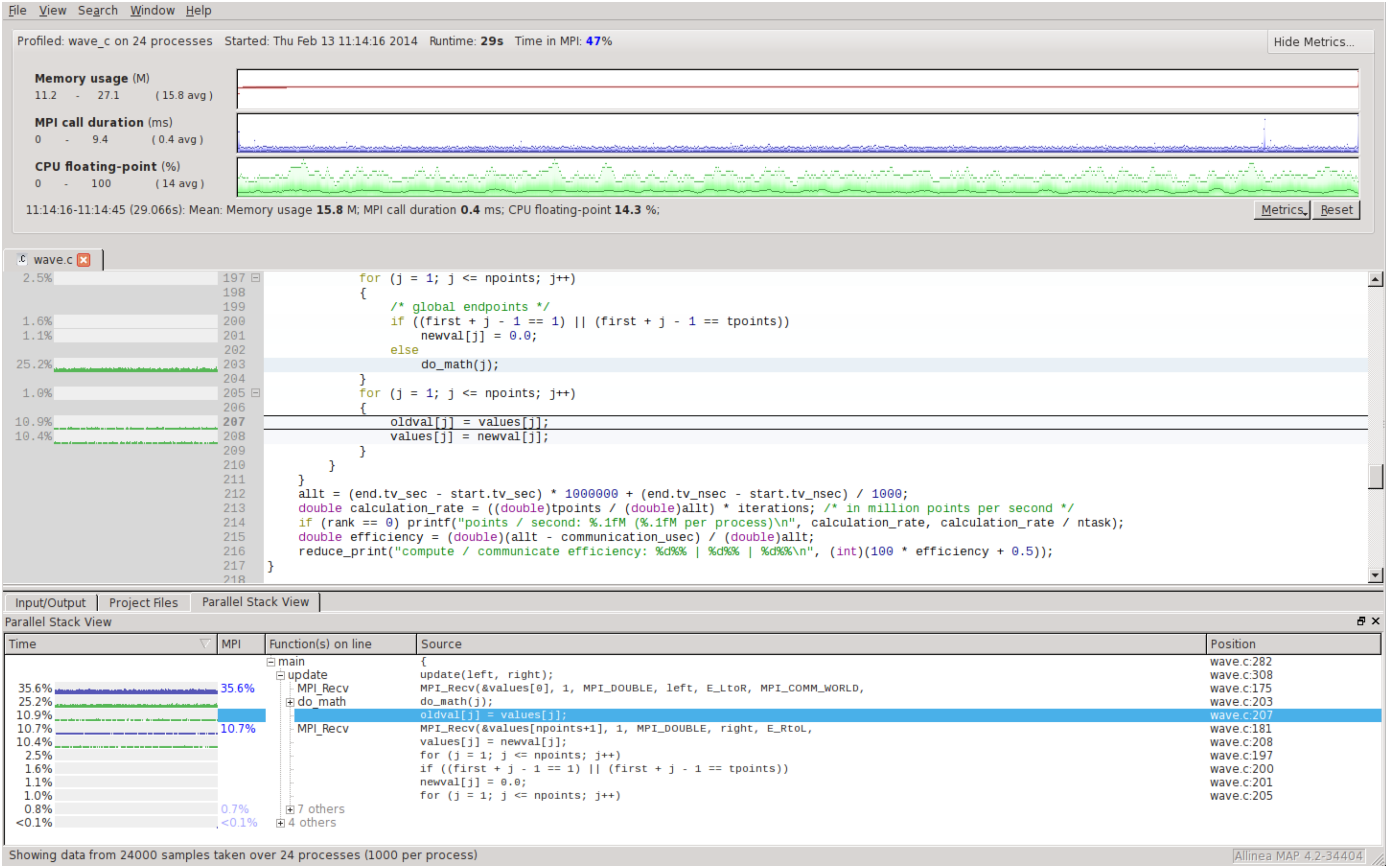
- The serial code was profiled before its conversion to CUDA
- Profiling identified the ~85% of the code readily available for parallelization, and the next 10-14% which could be converted with more efforts
- The plot was generated using these commands:
  - gcc -pg ... -o code
  - ./code
  - gprof ./code | gprof2dot.py | dot -Tpng -o output.png

MPI profiling in SHARCNET

- SHARCNET web portal lists three officially supported MPI profilers:
  - OPT: the old product from Allinea, installed only on requin, not usable for realistically large MPI jobs (say, >8 cores for >30 minutes).
  - IPM: open source profiler
  - MAP: new profiler from Allinea, installed on orca; integrated with their parallel debugger DDT

MAP overview

- Integrated with DDT debugger – makes it easier to go back and forth between profiling and debugging
- Uses statistical sampling (~1000 samples per rank by default) to dramatically accelerate the profiling process
- As a result, the profiler's overhead is <5%
- Polished and convenient to use GUI
- No need to recompile the code (needs to be compiled with “-g” - same as for debugging)
- Installed on orca, license for 512 MPI ranks



Profiled: wave_c on 24 processes   Started: Thu Feb 13 11:14:16 2014   Runtime: **29s**   Time in MPI: **47**%

Memory usage (M)
11.2 - 27.1 (15.8 avg)

MPI call duration (ms)
0 - 9.4 (0.4 avg)

CPU floating-point (%)
0 - 100 (14 avg)

11:14:16-11:14:45 (29.066s): Mean: Memory usage **15.8** M; MPI call duration **0.4** ms; CPU floating-point **14.3** %;

```
 2.5%               for (j = 1; j <= npoints; j++)
                    {
                        /* global endpoints */
 1.6%                   if ((first + j - 1 == 1) || (first + j - 1 == tpoints))
 1.1%                       newval[j] = 0.0;
                        else
25.2%                       do_math(j);
                    }
 1.0%               for (j = 1; j <= npoints; j++)
                    {
10.9%                   oldval[j] = values[j];
10.4%                   values[j] = newval[j];
                    }
                }
            }
            allt = (end.tv_sec - start.tv_sec) * 1000000 + (end.tv_nsec - start.tv_nsec) / 1000;
            double calculation_rate = ((double)tpoints / (double)allt) * iterations; /* in million points per second */
            if (rank == 0) printf("points / second: %.1fM (%.1fM per process)\n", calculation_rate, calculation_rate / ntask);
            double efficiency = (double)(allt - communication_usec) / (double)allt;
            reduce_print("compute / communicate efficiency: %d%% | %d%% | %d%%\n", (int)(100 * efficiency + 0.5));
        }
```

Parallel Stack View

Time	MPI	Function(s) on line	Source	Position
		main	{	wave.c:282
		update	update(left, right);	wave.c:308
35.6%	35.6%	MPI_Recv	MPI_Recv(&values[0], 1, MPI_DOUBLE, left, E_LtoR, MPI_COMM_WORLD,	wave.c:175
25.2%		do_math	do_math(j);	wave.c:203
10.9%			oldval[j] = values[j];	wave.c:207
10.7%	10.7%	MPI_Recv	MPI_Recv(&values[npoints+1], 1, MPI_DOUBLE, right, E_RtoL,	wave.c:181
10.4%			values[j] = newval[j];	wave.c:208
2.5%			for (j = 1; j <= npoints; j++)	wave.c:197
1.6%			if ((first + j - 1 == 1) || (first + j - 1 == tpoints))	wave.c:200
1.1%			newval[j] = 0.0;	wave.c:201
1.0%			for (j = 1; j <= npoints; j++)	wave.c:205
0.8%	0.7%	7 others		
<0.1%	<0.1%	4 others		

Showing data from 24000 samples taken over 24 processes (1000 per process)

Allinea MAP 4.2-34404

Using MAP

- Interactive use instructions (works for up to 24 cores):
  - ssh orca
  - ssh orc-dev1 (or dev2, dev3, dev4)
  - top (check if the node is busy; no point profiling your code on a busy node)
  - module load ddt
  - compile your code with “-O2 -g” switches (or -O3)
  - map ./code [arguments]
- Interactive analysis, plus *.map is written which can be analyzed later.

- Non-interactive use instructions (for up to 512 cores):
  - ssh orca
  - compile your code with “-O2 -g” switches (or -O3)
  - module load ddt
  - sqsub -q mpi -o out -r 1h --nompirun -n 2 map -profile -n 2 ./code [args]
- The *.map file produced during both interactive and non-interactive runs can be later analyzed via
  - map code.map

MAP requires an X window client

- The GUI part of MAP requires an X window client on your computer
  - Already present under Linux and Mac
  - Under Windows, a third party software is required
  - Mobaxterm is a good (and free) solution for Windows, as it combines three applications in one (ssh, sftp, and X window clients):

    http://mobaxterm.mobatek.net

Demo