Vlaanderen
is supercomputing

Code Profiling Workshop

Tier-2 HPC User Day 2025

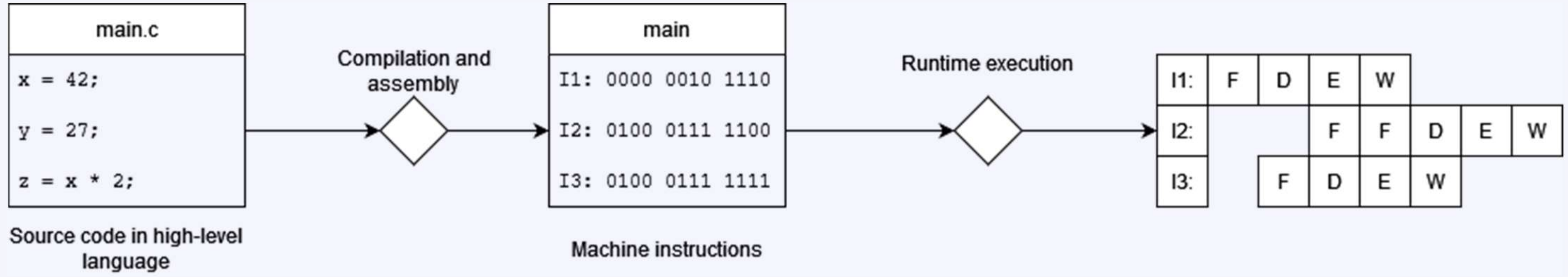
VLAAMS
SUPERCOMPUTER
CENTRUM

*Innovative Computing
for A Smarter Flanders*

vscentrum.be

The role of a code profiler in code optimization

- Large gap between high-level code and processor execution
  - Compiler transforms code into machine instructions
  - Processor executes instructions pipelined and out-of-order



main.c

```
x = 42;
y = 27;
z = x * 2;
```

Source code in high-level language

main

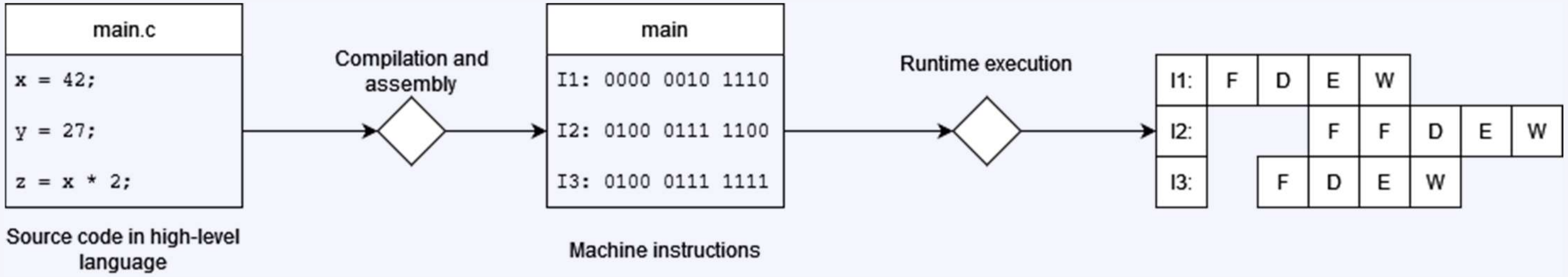
```
I1: 0000 0010 1110
I2: 0100 0111 1100
I3: 0100 0111 1111
```

Machine instructions

I1:	F	D	E	W			
I2:			F	F	D	E	W
I3:		F	D	E	W		

The role of a code profiler in code optimization

- Large gap between high-level code and processor execution
  - Compiler transforms code into machine instructions
  - Processor executes instructions pipelined and out-of-order



```
x = 42;

y = 27;

z = x * 2;
```

```
I1: 0000 0010 1110
I2: 0100 0111 1100
I3: 0100 0111 1111
```

I1:	F	D	E	W			
I2:			F	F	D	E	W
I3:		F	D	E	W		

- Profiler bridges hardware utilization and source code

VLAAMS
SUPERCOMPUTER
CENTRUM

The role of a code profiler in code optimization

Profiling helps you understand code performance characteristics

by collecting hardware performance counters

- identify hotspots and bottlenecks
- provide memory, I/O, accelerator usage statistics
- analyze MPI communication and other parallel approaches

Success stories

Laser ablation code in C (mechanical engineering)

- Calls to GSL routines gave a lot overhead

  => replacing with simple, pure C gave a 4x speedup
- Nested loop structure was bottleneck

  => code transformations and microarchitecture optimizations

  gave another 4x speedup

Success stories

Crop yield simulation code (AqauCrop) in Fortran

- Majority of time spent in I/O because of re-reading same file

  => 6x speedup by reading once and storing in memory
- Temporary files written to slow file system

  => 2x speedup by using local disk for temporary storage

Success stories

SP-Wind: Navier-Stokes solver in Fortran+MPI

- Strided memory access caused cache thrashing

  => reordering loops gives 2x speedup in specific case
- Inter-node communication bottleneck in large-scale runs

  => rethinking domain decomposition gives 30% speedup

Agenda

- **Introduction to and overview of profilers**
- Examples using Linaro MAP
  - Serial CPU code
  - MPI parallel CPU code
  - CUDA GPU code

Tracing vs sampling

Tracing

- Insert function calls for counters
- Generally, requires code changes (instrumenting can be automated)
- Should produce detailed and accurate results
- Can introduce overhead and output can be overwhelming
- Good for in-depth analysis (if you know the code)

Tracing vs sampling

Tracing

- Insert function calls for counters
- Generally, requires code changes (instrumenting can be automated)
- Should produce detailed and accurate results
- Can introduce overhead and output can be overwhelming
- Good for in-depth analysis (if you know the code)

Sampling

- Statistical average of snapshots
- Requires compilation with debug symbols (`-g` option)
- *Might* lack details or be inaccurate, especially for very short runs
- Usually small overhead, easy to interpret (with a *good* profiler)
- Get a quick overview without knowledge of the code

The HPC profiler landscape

- TAU Performance System® is a portable profiling and tracing toolkit for performance analysis of parallel programs written in Fortran, C, C++, UPC, Java, Python.
- Scalasca is a software tool that supports the performance optimization of parallel programs by measuring and analyzing their runtime behavior.
- BSC Tools Paraver, a performance analyzer based on traces with a great flexibility to explore the collected data (also see https://pop-coe.eu/) `[Paraver/4.11.1-foss-2022a]`
- HPCToolkit is an integrated suite of tools for measurement and analysis of program performance on computers ranging
- Likwid is a simple to install and use toolsuite of command line applications and a library for performance oriented programmers. `[likwid/5.3.0-GCC-10.3.0]`
- Intel® VTune™ Profiler optimizes application performance, system performance, and system configuration for AI, HPC, cloud, IoT, media, storage, and more `[VTune/2022.2.0]`
- NVIDIA Nsight Systems is a performance analysis tool for visualizing app algorithms and scaling optimization across CPUs and GPUs

The HPC profiler landscape

- … and many more
- Spend *some* time choosing the right tools for you
- Get comfortable with the tools you choose!
- Combine insights from different tools

The Linaro MAP profiler

- Sampling profiler from LinaroForge (previously ArmForge/Allinea)
- Easy to use, attractive GUI
- Supports many HPC use cases
  - C, C++, Fortran, Python
  - MPI, OpenMP, pthreads, CUDA
- Similar interface as Linaro DDT debugger
- Expensive; we offer 272 license tokens on Tier-2@KU Leuven/UHasselt

Agenda

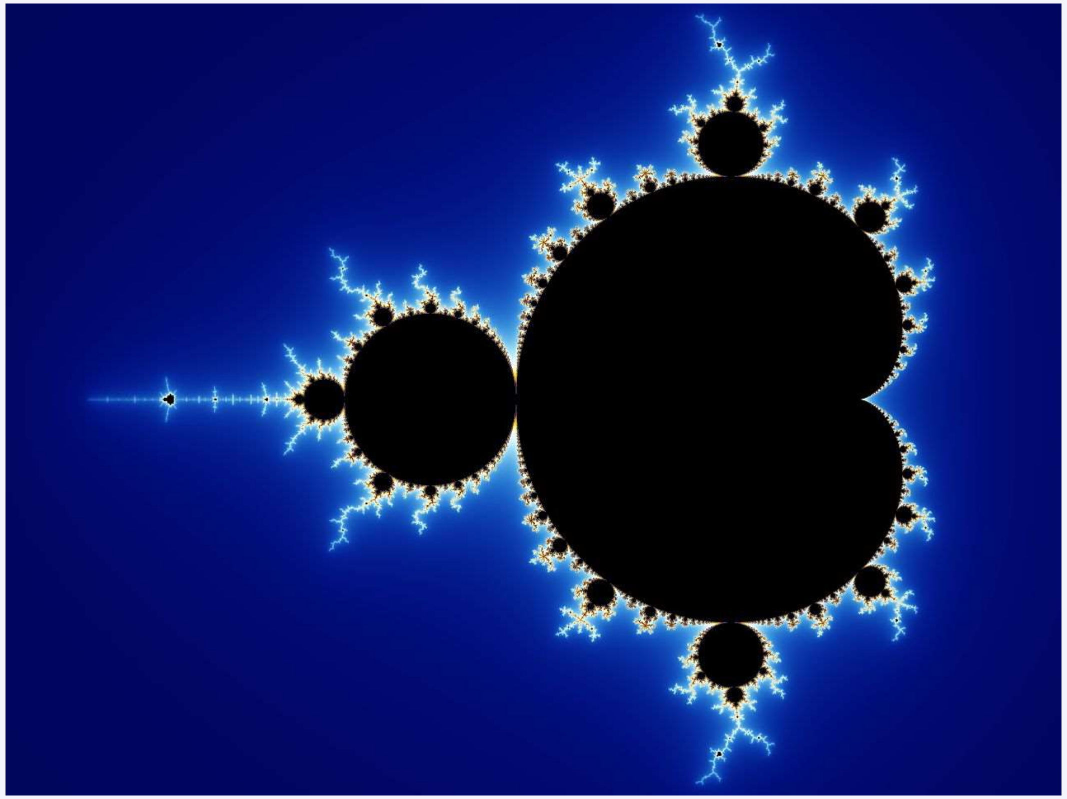
- Introduction to and overview of profilers
- Examples using Linaro MAP
  - **Serial CPU code**
  - MPI parallel CPU code
  - CUDA GPU code

The Mandelbrot set: mathematics

- Fractal in complex plane defined as complex numbers c for which mapping:

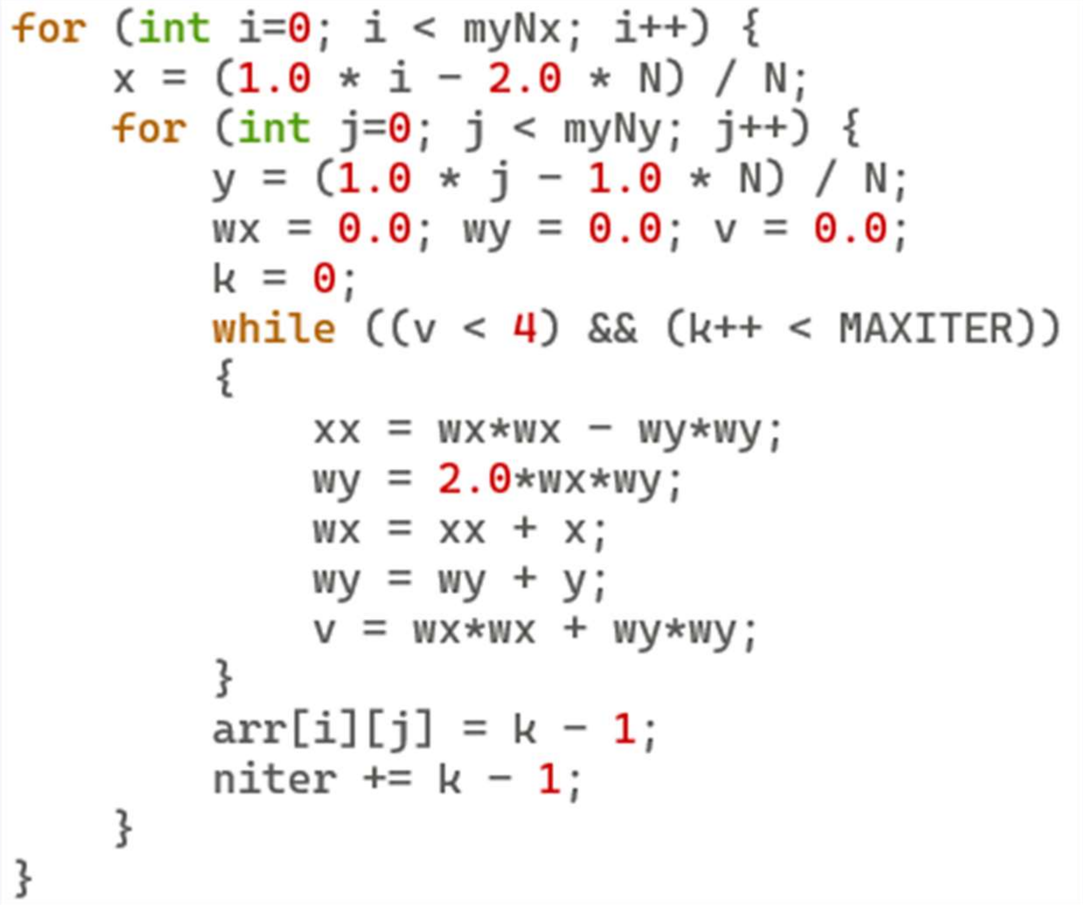
  $$z_{n+1} = z_n^2 + c$$

  remains bounded as n increases $(z_0 = 0)$
- For each pixel $c = x + \ iy$, apply mapping until it converges or maximal number of iterations is reached
- Number of iterations indicates color of pixel

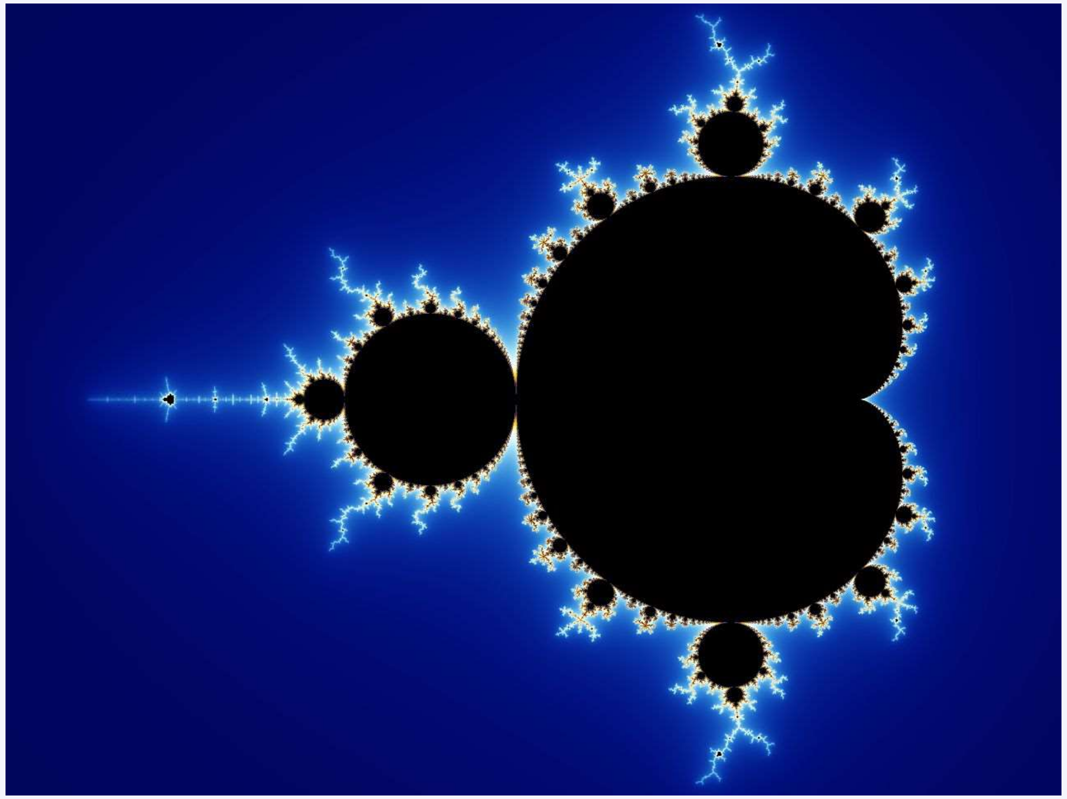


VLAAMS
SUPERCOMPUTER
CENTRUM

The Mandelbrot set: code



```
for (int i=0; i < myNx; i++) {
    x = (1.0 * i - 2.0 * N) / N;
    for (int j=0; j < myNy; j++) {
        y = (1.0 * j - 1.0 * N) / N;
        wx = 0.0; wy = 0.0; v = 0.0;
        k = 0;
        while ((v < 4) && (k++ < MAXITER))
        {
            xx = wx*wx - wy*wy;
            wy = 2.0*wx*wy;
            wx = xx + x;
            wy = wy + y;
            v = wx*wx + wy*wy;
        }
        arr[i][j] = k - 1;
        niter += k - 1;
    }
}
```



In all example runs, the grid size is 51840 x 34560 (1.8B grid points)

Linaro MAP basics: batch mode

- Compile your code with debug symbols (`-g1`) [Optional: disable inlining]

Linaro MAP basics: batch mode

- Compile your code with debug symbols (`-g1`) [Optional: disable inlining]
- Load the LinaroForge module, before other modules:

```
module load LinaroForge
```

Linaro MAP basics: batch mode

- Compile your code with debug symbols (`-g1`) [Optional: disable inlining]
- Load the LinaroForge module, before other modules:

```
module load LinaroForge
```

- Profile your run, inside your **job script**:

```
map --profile mpirun <your_executable> <options>
```

```
map --profile python <your_script.py>
```

Linaro MAP basics: batch mode

- Compile your code with debug symbols (`-g1`) [Optional: disable inlining]
- Load the LinaroForge module, before other modules:

```
module load LinaroForge
```

- Profile your run, inside your **job script**:

```
map --profile mpirun <your_executable> <options>
map --profile python <your_script.py>
```

- Open the generated map file (requires graphical connection):

```
map <your_executable>_<timestamp>.map
```

Linaro MAP basics: batch mode

- Compile your code with debug symbols (`-g1`) [Optional: disable inlining]
- Load the LinaroForge module, before other modules:

```
module load LinaroForge
```

- Profile your run, inside your **job script**:

```
map --profile mpirun <your_executable> <options>
map --profile python <your_script.py>
```

- Open the generated map file (requires graphical connection):

```
map <your_executable>_<timestamp>.map
```

- [Optional] Launch profiling run interactively from `map` GUI
- [Optional] Run client locally and use remote launch + job submission

Profiling serial CPU code

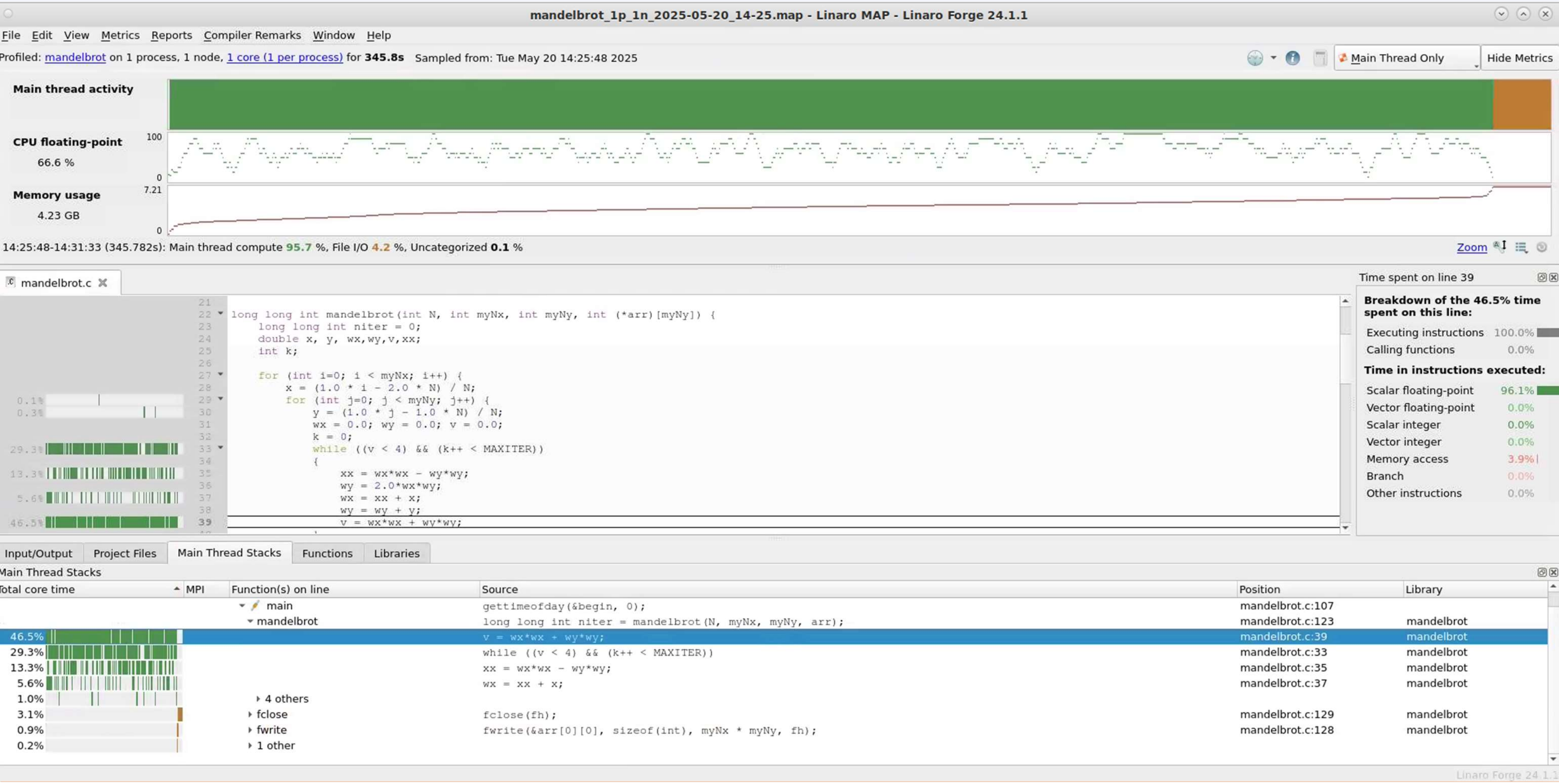
Top tip 1:

start by optimizing serial performance

Top tip 2:

profile a run that is representative for your actual workload

Visualization of the MAP file



mandelbrot_1p_1n_2025-05-20_14-25.map - Linaro MAP - Linaro Forge 24.1.1

Profiled: mandelbrot on 1 process, 1 node, 1 core (1 per process) for **345.8s** Sampled from: Tue May 20 14:25:48 2025

CPU floating-point 66.6 %

Memory usage 4.23 GB

14:25:48-14:31:33 (345.782s): Main thread compute 95.7 %, File I/O 4.2 %, Uncategorized 0.1 %

```
long long int mandelbrot(int N, int myNx, int myNy, int (*arr)[myNy]) {
    long long int niter = 0;
    double x, y, wx,wy,v,xx;
    int k;

    for (int i=0; i < myNx; i++) {
        x = (1.0 * i - 2.0 * N) / N;
        for (int j=0; j < myNy; j++) {
            y = (1.0 * j - 1.0 * N) / N;
            wx = 0.0; wy = 0.0; v = 0.0;
            k = 0;
            while ((v < 4) && (k++ < MAXITER))
            {
                xx = wx*wx - wy*wy;
                wy = 2.0*wx*wy;
                wx = xx + x;
                wy = wy + y;
                v = wx*wx + wy*wy;
```

Line	Time
29	0.1%
30	0.3%
33	29.3%
35	13.3%
37	5.6%
39	46.5%

Time spent on line 39

Breakdown of the 46.5% time spent on this line:

Executing instructions	100.0%
Calling functions	0.0%

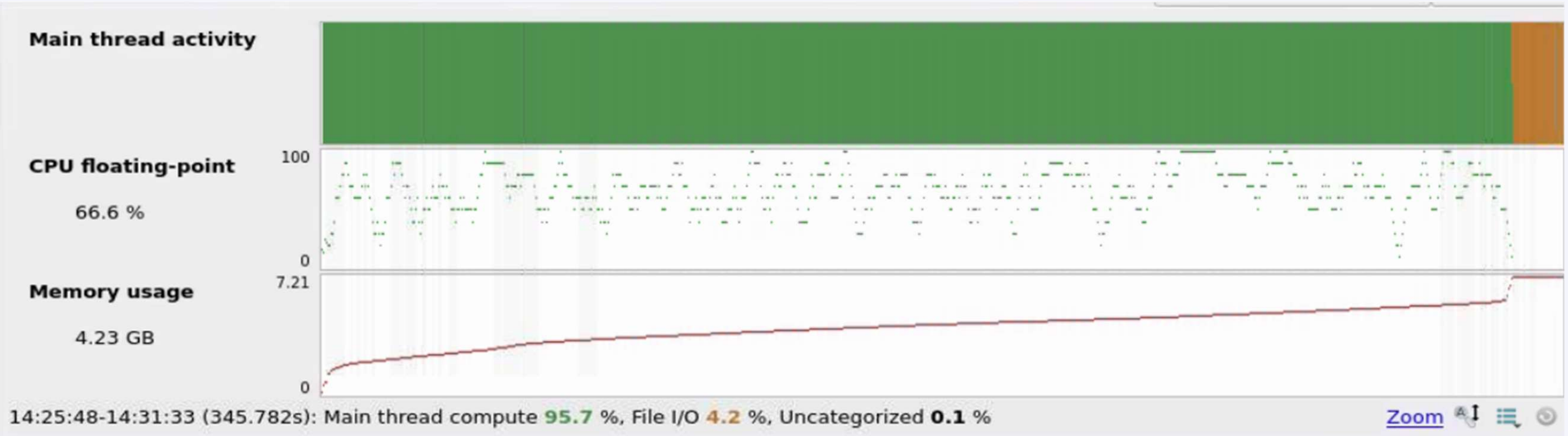
Time in instructions executed:

Scalar floating-point	96.1%
Vector floating-point	0.0%
Scalar integer	0.0%
Vector integer	0.0%
Memory access	3.9%
Branch	0.0%
Other instructions	0.0%

Main Thread Stacks

Total core time	MPI	Function(s) on line	Source	Position	Library
		main	gettimeofday(&begin, 0);	mandelbrot.c:107	
		mandelbrot	long long int niter = mandelbrot(N, myNx, myNy, arr);	mandelbrot.c:123	mandelbrot
46.5%			v = wx*wx + wy*wy;	mandelbrot.c:39	mandelbrot
29.3%			while ((v < 4) && (k++ < MAXITER))	mandelbrot.c:33	mandelbrot
13.3%			xx = wx*wx - wy*wy;	mandelbrot.c:35	mandelbrot
5.6%			wx = xx + x;	mandelbrot.c:37	mandelbrot
1.0%		4 others			
3.1%		fclose	fclose(fh);	mandelbrot.c:129	mandelbrot
0.9%		fwrite	fwrite(&arr[0][0], sizeof(int), myNx * myNy, fh);	mandelbrot.c:128	mandelbrot
0.2%		1 other			

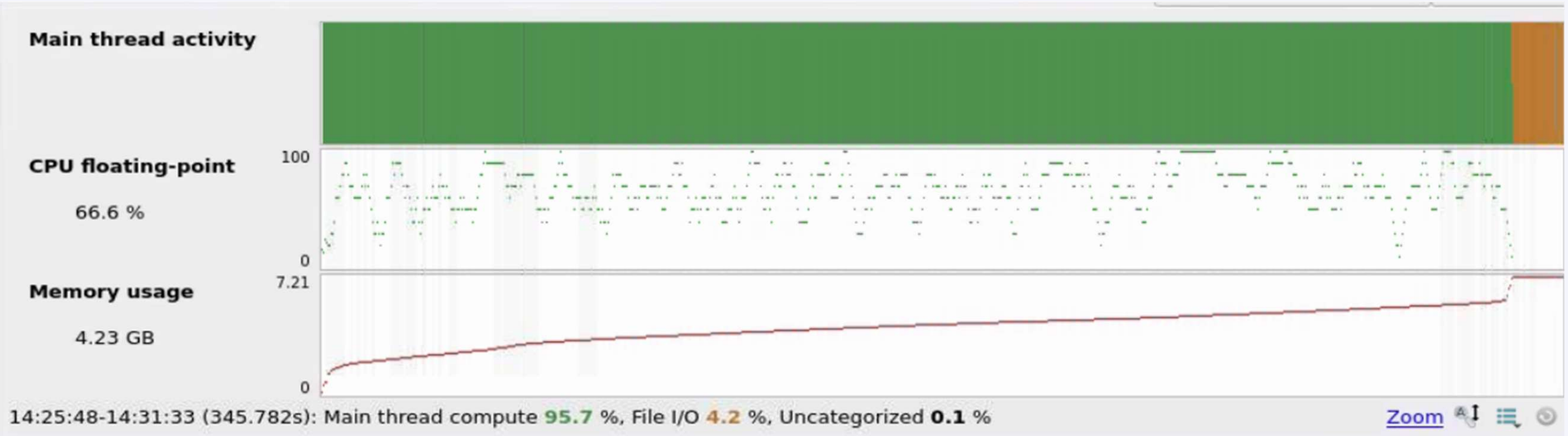
Timeline view



14:25:48-14:31:33 (345.782s): Main thread compute 95.7 %, File I/O 4.2 %, Uncategorized **0.1** %

- Horizontal axis indicates time
- **Main thread activity:**
  - vertical axis indicates different processes (single process in this example)
  - color code: green -> compute – orange -> I/O – blue -> MPI – purple -> accelerator
- **Other metrics can be chosen from taskbar**

Timeline view

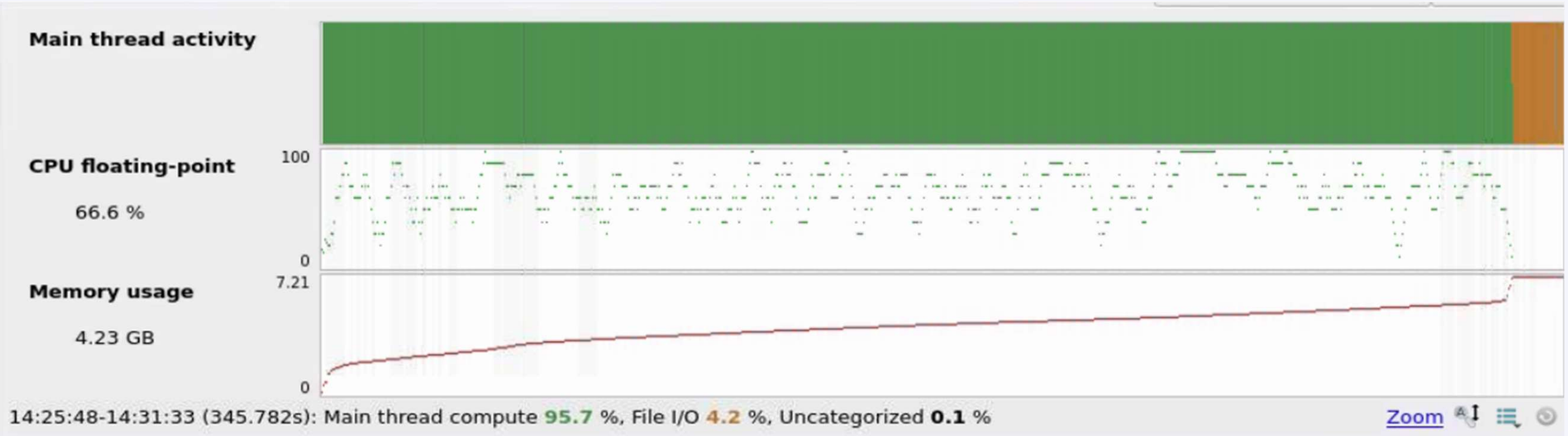


14:25:48-14:31:33 (345.782s): Main thread compute 95.7 %, File I/O 4.2 %, Uncategorized 0.1 %

Top tip 3: check overhead introduced by profiler

Without profiler: 345s - With profiler: 348s - Difference: <1%

Timeline view



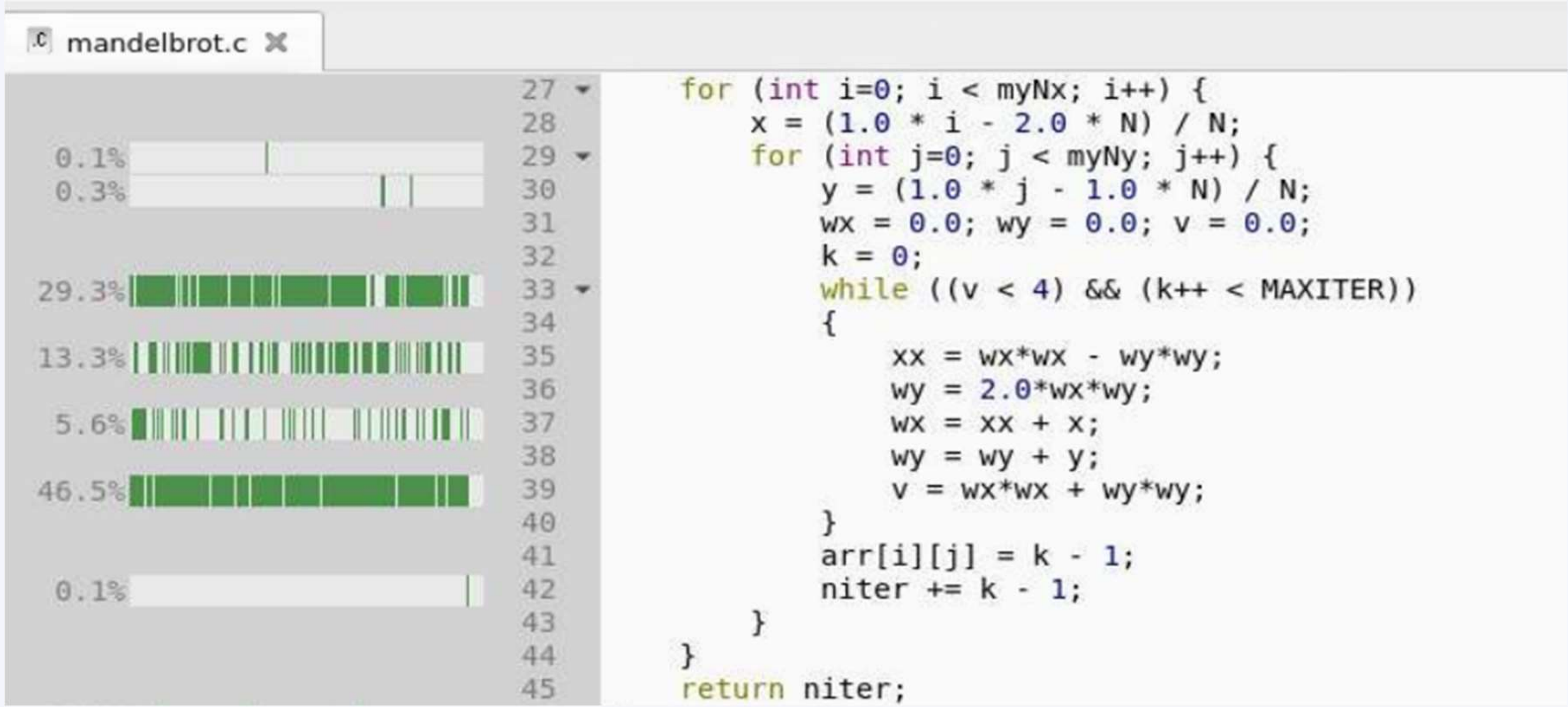
14:25:48-14:31:33 (345.782s): Main thread compute 95.7 %, File I/O 4.2 %, Uncategorized 0.1 %

Top tip 3: check overhead introduced by profiler

Without profiler: 345s - With profiler: 348s - Difference: <1%

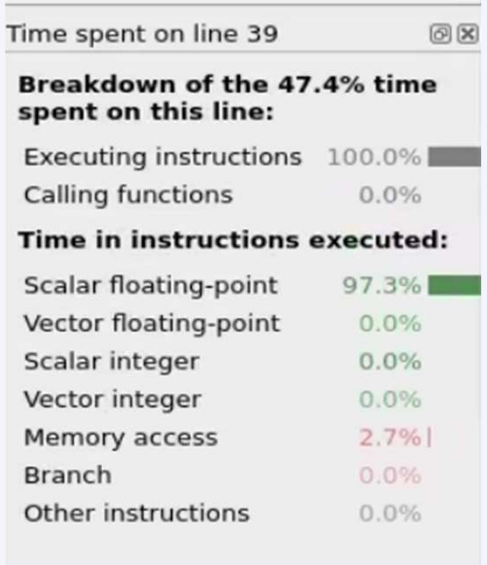
Top tip 4: zoom in on specific phases of the run in the timeline view

Source code view



mandelbrot.c

```
        27  for (int i=0; i < myNx; i++) {
        28      x = (1.0 * i - 2.0 * N) / N;
 0.1%   29      for (int j=0; j < myNy; j++) {
 0.3%   30          y = (1.0 * j - 1.0 * N) / N;
        31          wx = 0.0; wy = 0.0; v = 0.0;
        32          k = 0;
29.3%   33          while ((v < 4) && (k++ < MAXITER))
        34          {
13.3%   35              xx = wx*wx - wy*wy;
        36              wy = 2.0*wx*wy;
 5.6%   37              wx = xx + x;
        38              wy = wy + y;
46.5%   39              v = wx*wx + wy*wy;
        40          }
        41          arr[i][j] = k - 1;
 0.1%   42          niter += k - 1;
        43      }
        44  }
        45  return niter;
```



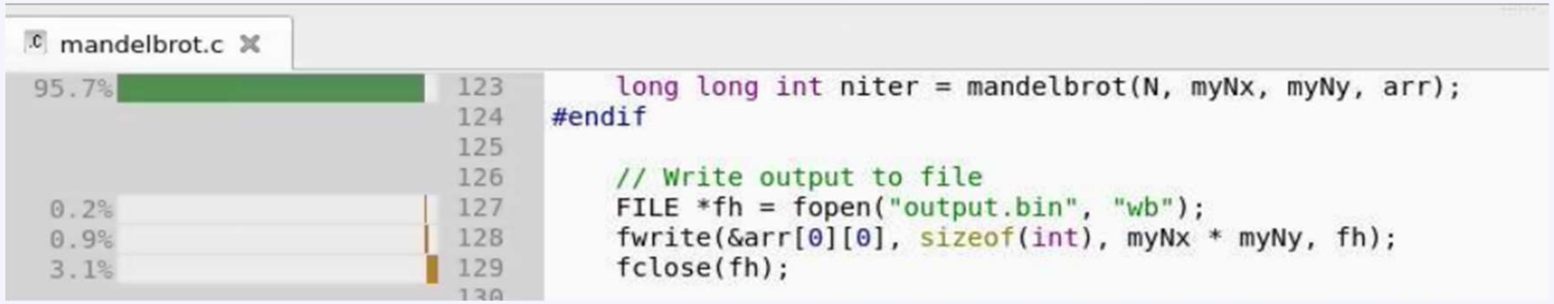
Time spent on line 39

Breakdown of the 47.4% time spent on this line:

Executing instructions	100.0%
Calling functions	0.0%

Time in instructions executed:

Scalar floating-point	97.3%
Vector floating-point	0.0%
Scalar integer	0.0%
Vector integer	0.0%
Memory access	2.7%
Branch	0.0%
Other instructions	0.0%

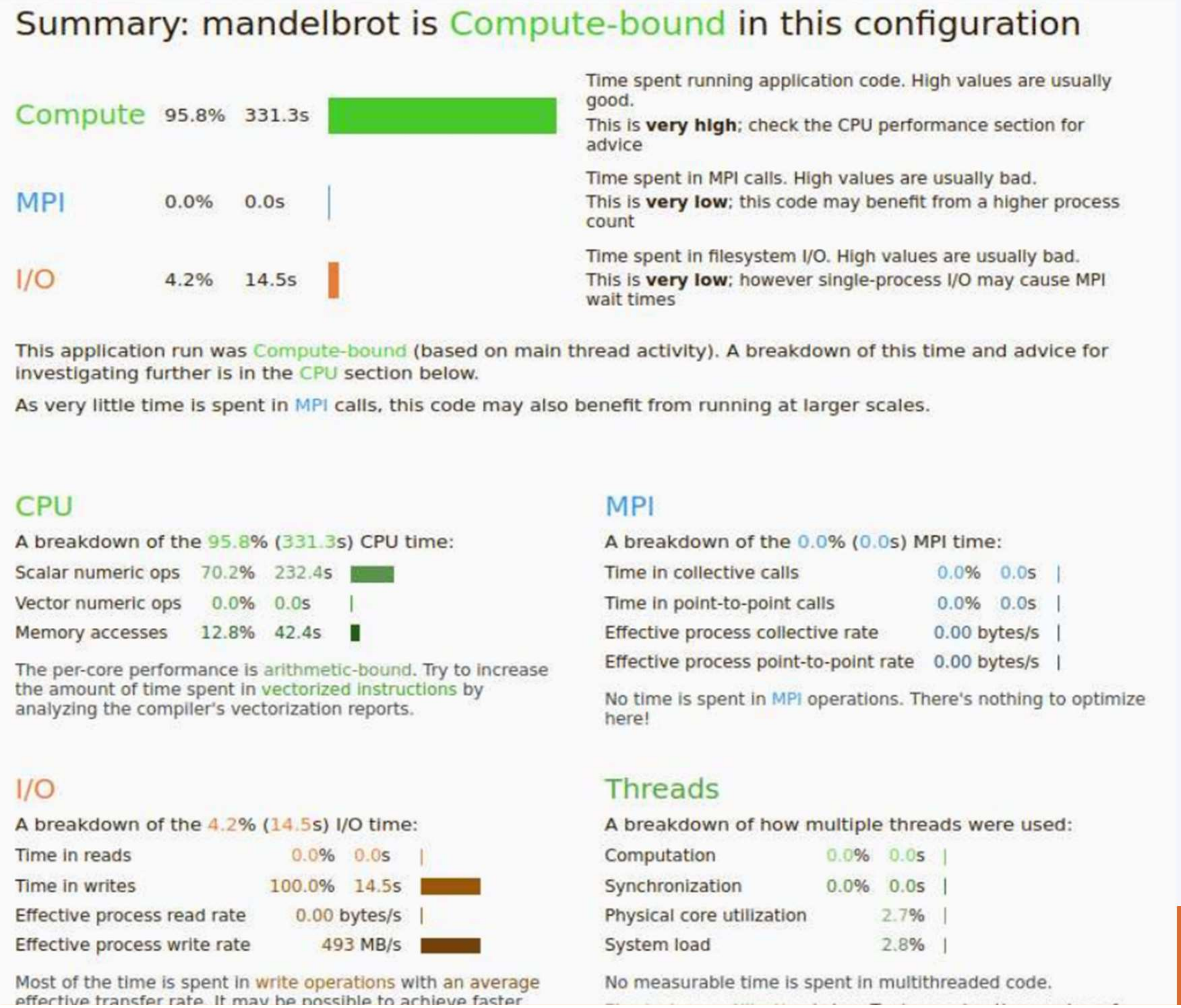


mandelbrot.c

```
95.7%   123      long long int niter = mandelbrot(N, myNx, myNy, arr);
        124  #endif
        125
        126      // Write output to file
 0.2%   127      FILE *fh = fopen("output.bin", "wb");
 0.9%   128      fwrite(&arr[0][0], sizeof(int), myNx * myNy, fh);
 3.1%   129      fclose(fh);
        130
```

Indicates how much time is spent in each line and how it is spent

Performance report



Summary: mandelbrot is Compute-bound in this configuration

Compute	95.8%	331.3s	Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for advice
MPI	0.0%	0.0s	Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from a higher process count
I/O	4.2%	14.5s	Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times

This application run was Compute-bound (based on main thread activity). A breakdown of this time and advice for investigating further is in the CPU section below.

As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 95.8% (331.3s) CPU time:

Scalar numeric ops	70.2%	232.4s
Vector numeric ops	0.0%	0.0s
Memory accesses	12.8%	42.4s

The per-core performance is arithmetic-bound. Try to increase the amount of time spent in vectorized instructions by analyzing the compiler's vectorization reports.

MPI

A breakdown of the 0.0% (0.0s) MPI time:

Time in collective calls	0.0%	0.0s
Time in point-to-point calls	0.0%	0.0s
Effective process collective rate	0.00 bytes/s	
Effective process point-to-point rate	0.00 bytes/s	

No time is spent in MPI operations. There's nothing to optimize here!

I/O

A breakdown of the 4.2% (14.5s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	14.5s
Effective process read rate	0.00 bytes/s	
Effective process write rate	493 MB/s	

Most of the time is spent in write operations with an average effective transfer rate. It may be possible to achieve faster

Threads

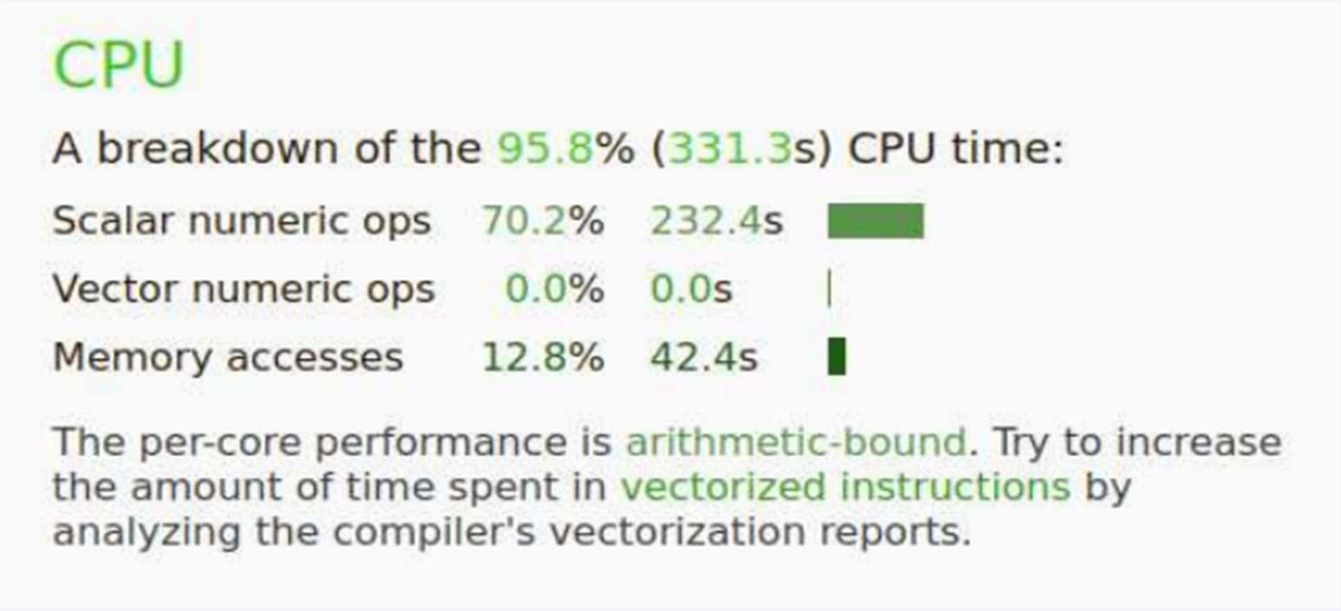
A breakdown of how multiple threads were used:

Computation	0.0%	0.0s
Synchronization	0.0%	0.0s
Physical core utilization	2.7%	
System load	2.8%	

No measurable time is spent in multithreaded code.

- Click on “reports” in toolbar to open performance report
- Quick glance of performance and suggestions for improvement
  => recommended as starting point

Optimizing serial performance



CPU

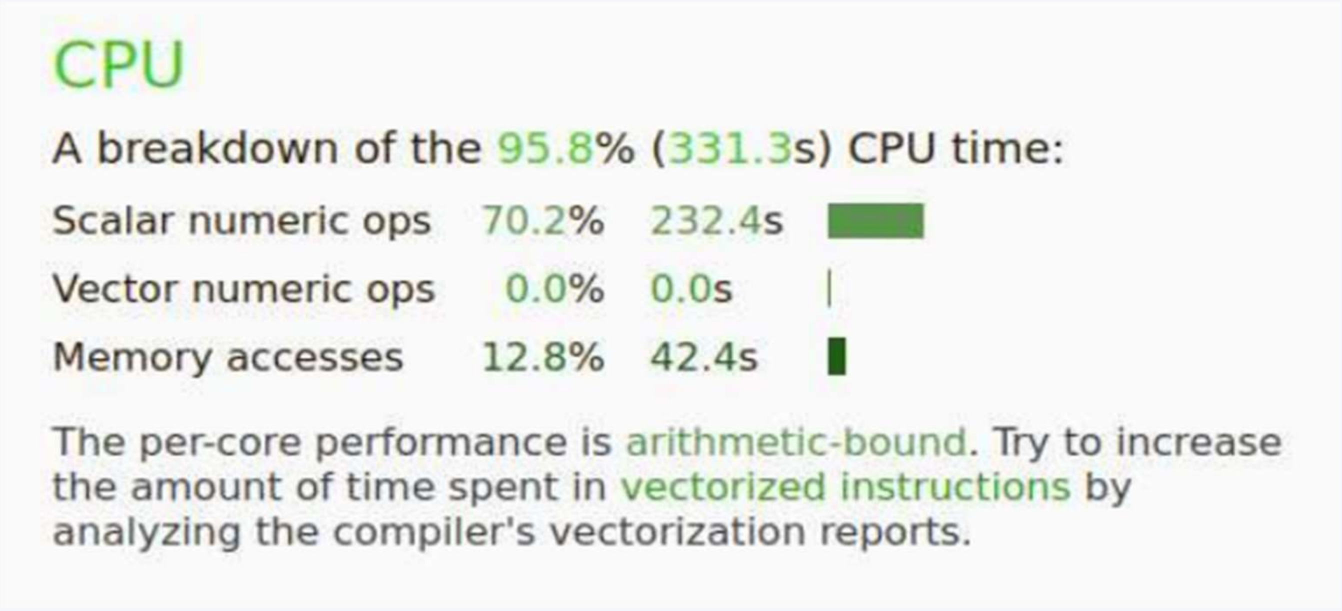
A breakdown of the 95.8% (331.3s) CPU time:

Scalar numeric ops	70.2%	232.4s	█████
Vector numeric ops	0.0%	0.0s	|
Memory accesses	12.8%	42.4s	█

The per-core performance is arithmetic-bound. Try to increase the amount of time spent in vectorized instructions by analyzing the compiler's vectorization reports.

Top tip 5: focus on sections of the code consuming most resources

Optimizing serial performance



CPU

A breakdown of the 95.8% (331.3s) CPU time:

Scalar numeric ops	70.2%	232.4s	▬
Vector numeric ops	0.0%	0.0s	|
Memory accesses	12.8%	42.4s	▮

The per-core performance is arithmetic-bound. Try to increase the amount of time spent in vectorized instructions by analyzing the compiler's vectorization reports.

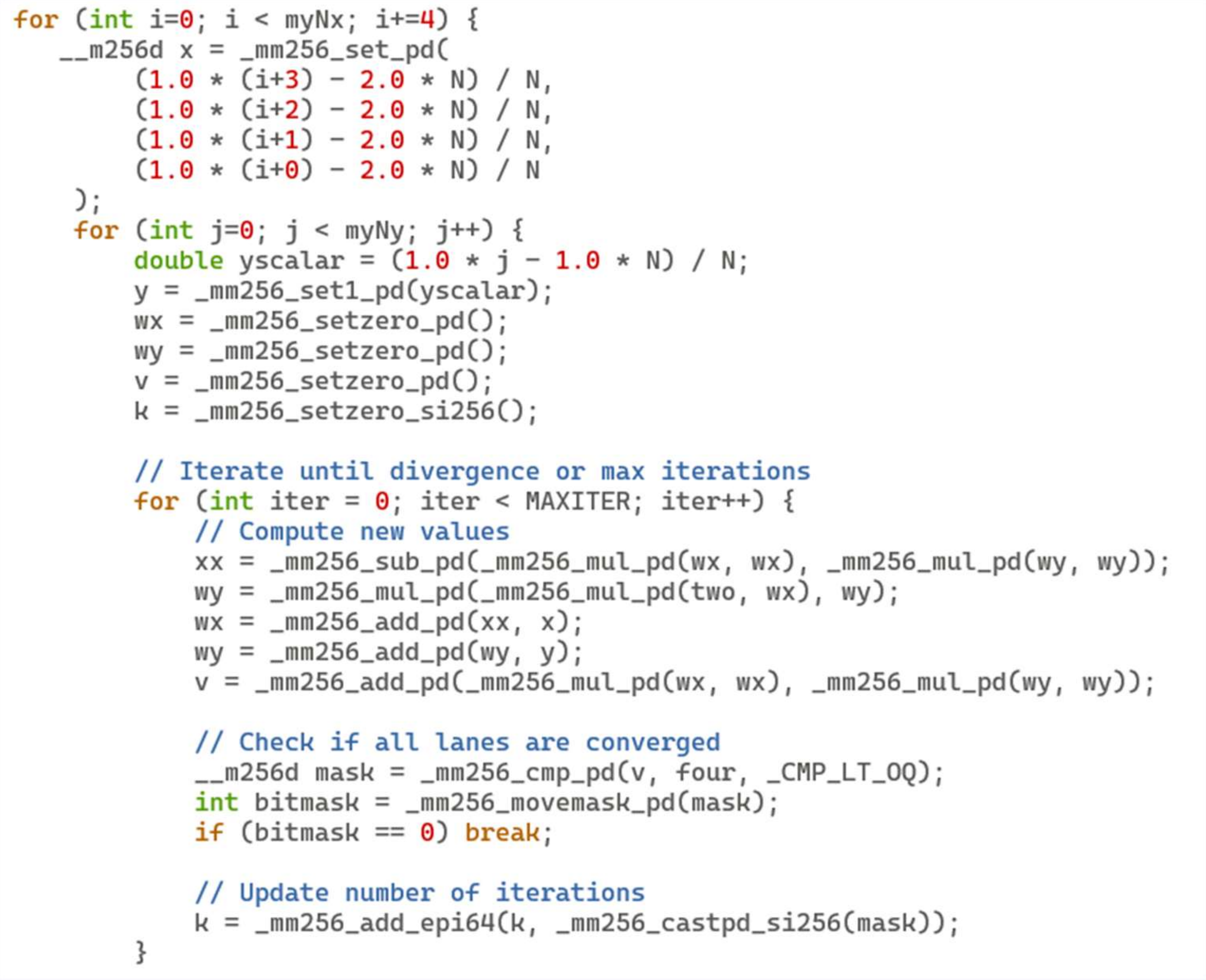
Top tip 5: focus on sections of the code consuming most resources

- Code does not make use of SIMD capabilities
- Compiler fails to vectorize loop:

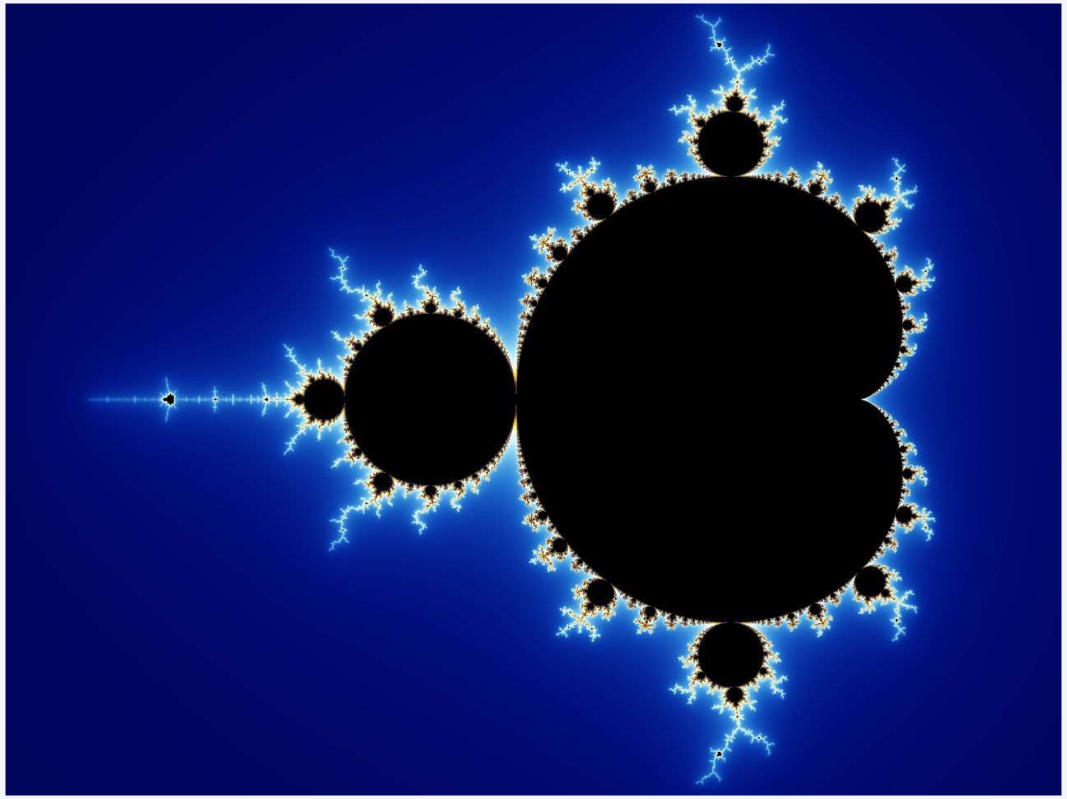
```
$ gcc -fopt-info-vec-missed
mandelbrot.c:33:33: missed: couldn't vectorize loop
mandelbrot.c:33:33: missed: not vectorized: control flow in loop
```

- Let’s write a version with vector intrinsics ourself!

The Mandelbrot set: code with vector instructions

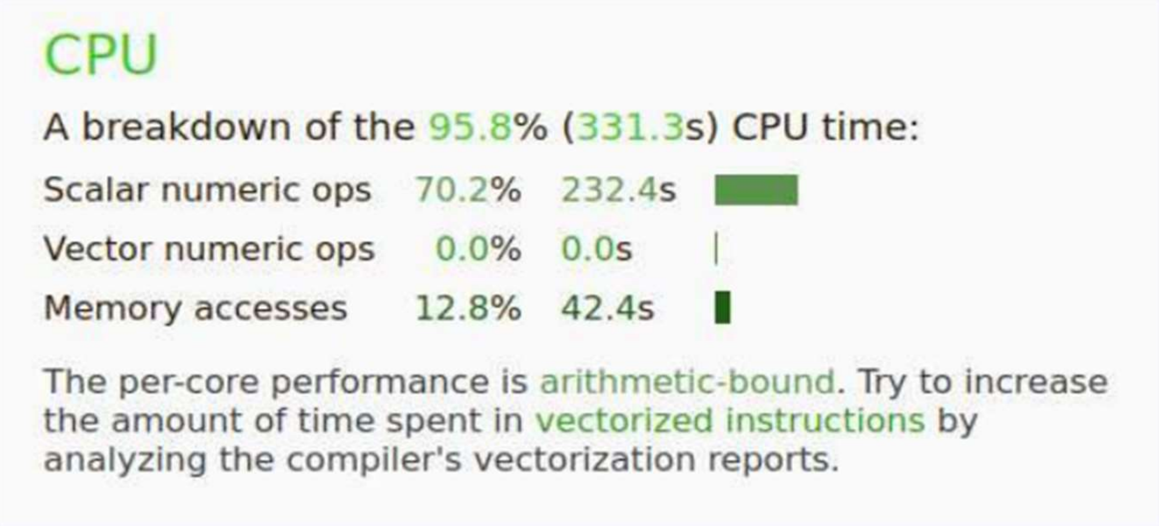


```
for (int i=0; i < myNx; i+=4) {
   __m256d x = _mm256_set_pd(
        (1.0 * (i+3) - 2.0 * N) / N,
        (1.0 * (i+2) - 2.0 * N) / N,
        (1.0 * (i+1) - 2.0 * N) / N,
        (1.0 * (i+0) - 2.0 * N) / N
    );
    for (int j=0; j < myNy; j++) {
        double yscalar = (1.0 * j - 1.0 * N) / N;
        y = _mm256_set1_pd(yscalar);
        wx = _mm256_setzero_pd();
        wy = _mm256_setzero_pd();
        v = _mm256_setzero_pd();
        k = _mm256_setzero_si256();

        // Iterate until divergence or max iterations
        for (int iter = 0; iter < MAXITER; iter++) {
            // Compute new values
            xx = _mm256_sub_pd(_mm256_mul_pd(wx, wx), _mm256_mul_pd(wy, wy));
            wy = _mm256_mul_pd(_mm256_mul_pd(two, wx), wy);
            wx = _mm256_add_pd(xx, x);
            wy = _mm256_add_pd(wy, y);
            v = _mm256_add_pd(_mm256_mul_pd(wx, wx), _mm256_mul_pd(wy, wy));

            // Check if all lanes are converged
            __m256d mask = _mm256_cmp_pd(v, four, _CMP_LT_OQ);
            int bitmask = _mm256_movemask_pd(mask);
            if (bitmask == 0) break;

            // Update number of iterations
            k = _mm256_add_epi64(k, _mm256_castpd_si256(mask));
        }
```



Improved performance with AVX2

Without vectorization



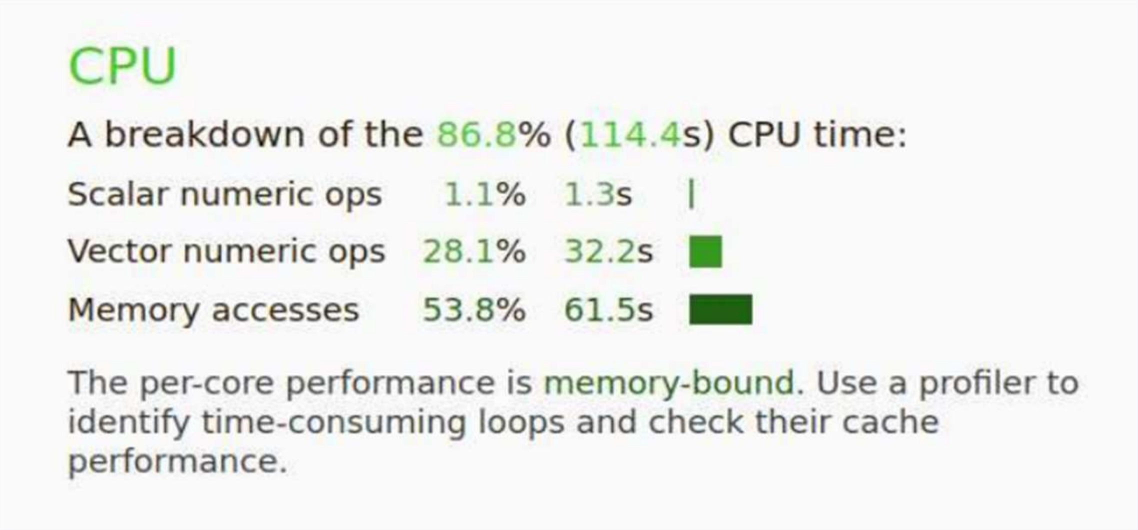
CPU

A breakdown of the 95.8% (331.3s) CPU time:

Scalar numeric ops	70.2%	232.4s
Vector numeric ops	0.0%	0.0s
Memory accesses	12.8%	42.4s

The per-core performance is arithmetic-bound. Try to increase the amount of time spent in vectorized instructions by analyzing the compiler's vectorization reports.

With vectorization



CPU

A breakdown of the 86.8% (114.4s) CPU time:

Scalar numeric ops	1.1%	1.3s
Vector numeric ops	28.1%	32.2s
Memory accesses	53.8%	61.5s

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

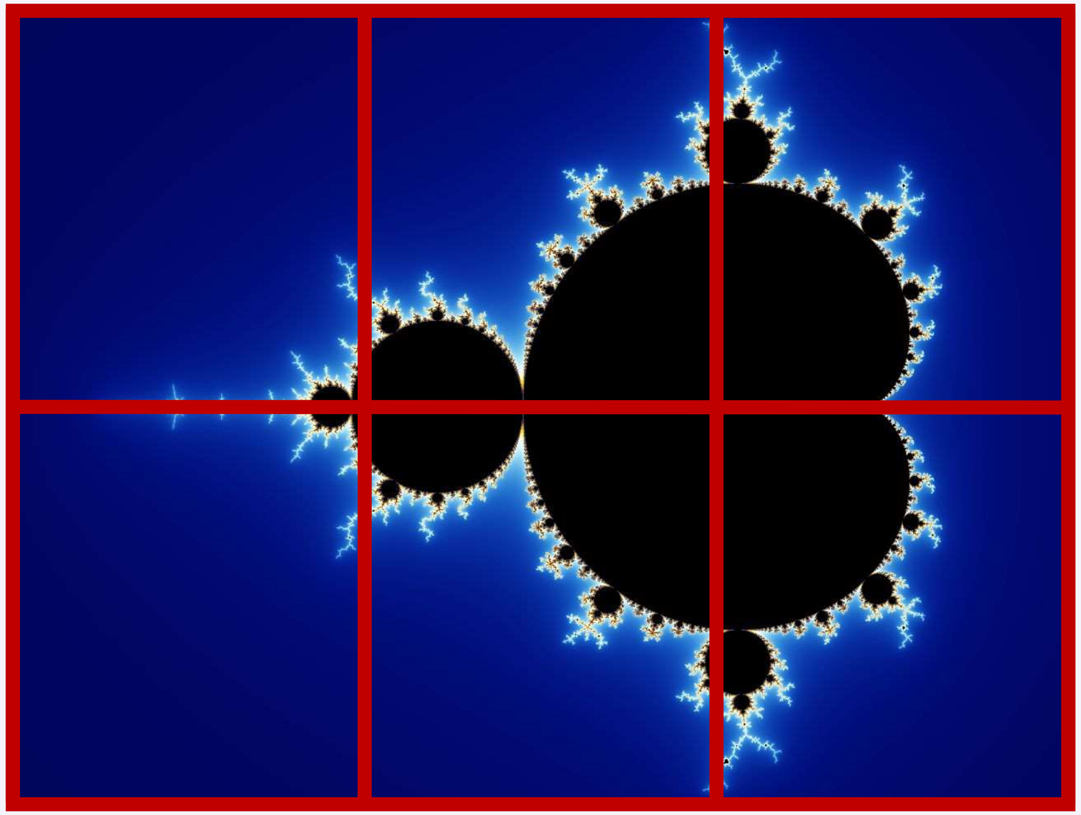
- Total runtime decreases from 348s to 128s (2.7x speedup)
- AVX2 lanes contain 4 double-precision numbers, but vectorizing loop introduces overhead, so no 4x speedup
- Code now becomes memory-bound, might be the next thing to work on

Agenda

- Introduction to and overview of profilers
- Examples using Linaro MAP
    - Serial CPU code
    - **MPI parallel CPU code**
    - CUDA GPU code

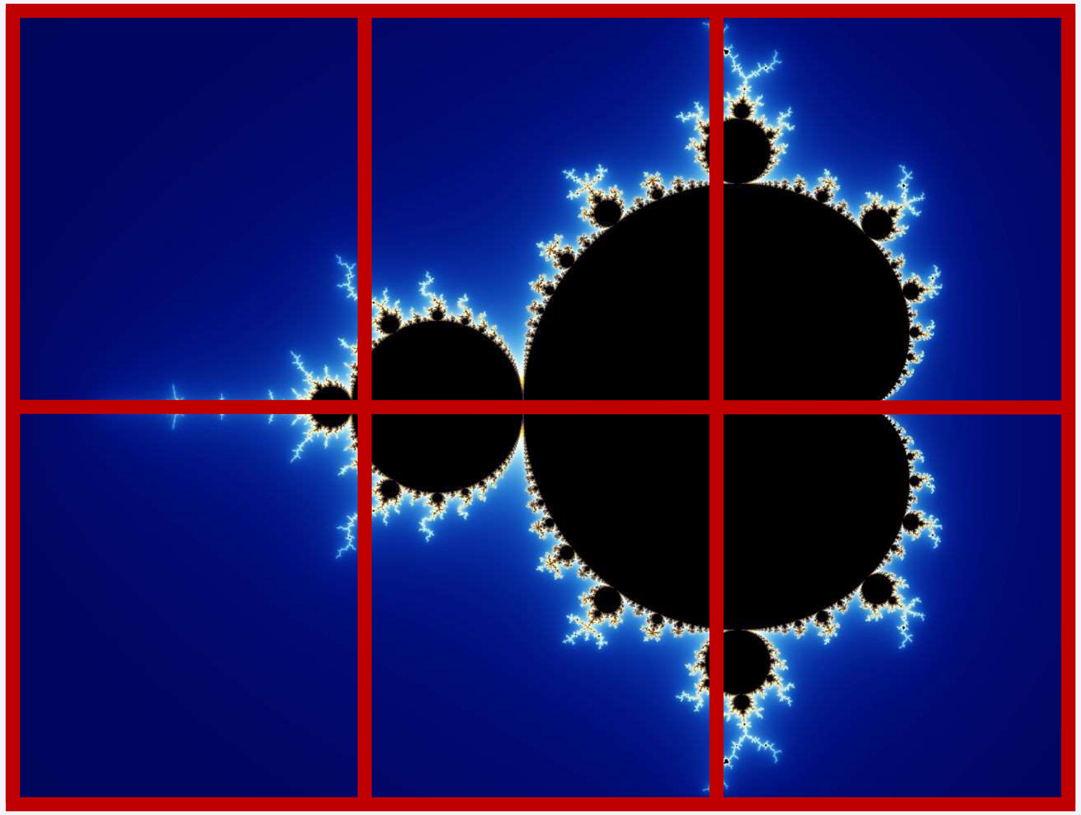
Parallelization by domain decomposition

- Observe that each pixel (xy coordinate) is independent of the others
- Divide all pixels over processes and let each process work on its pixels independently
  => domain decomposition

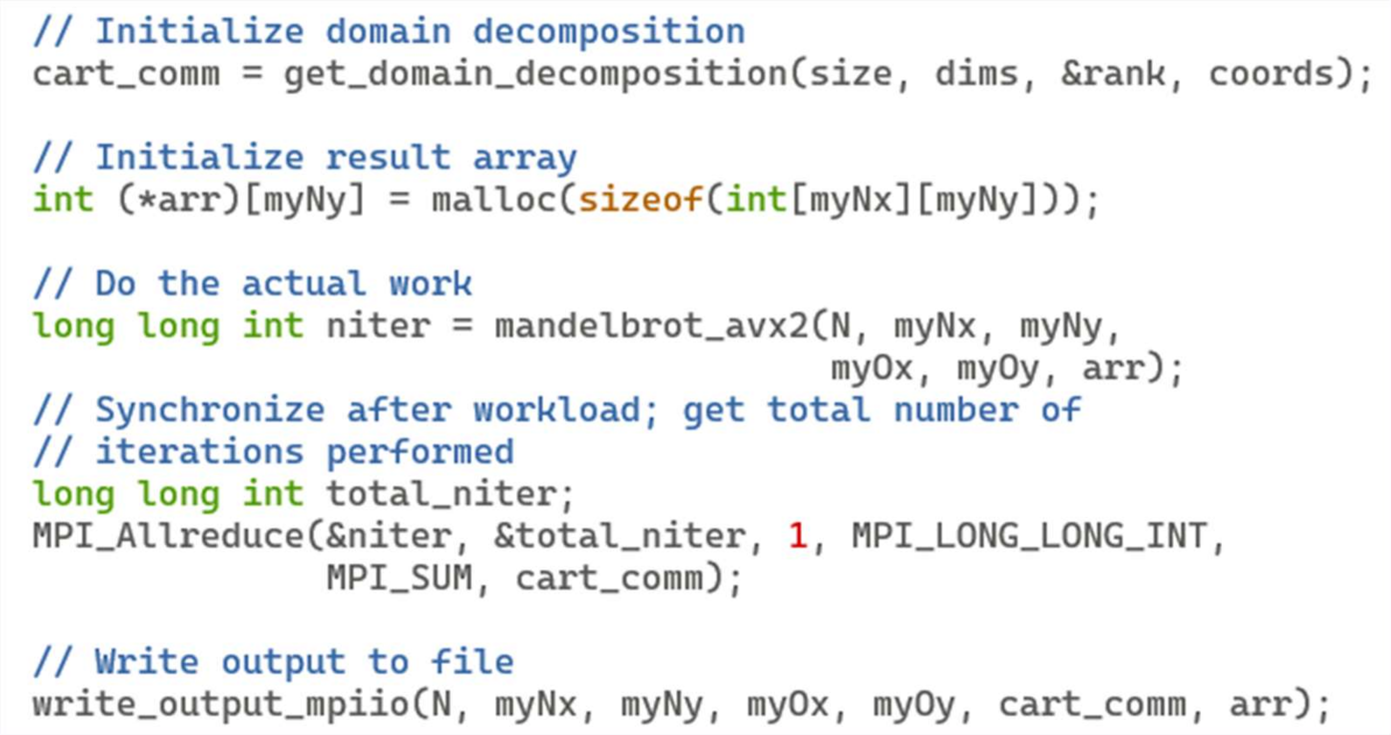


Parallelization by domain decomposition

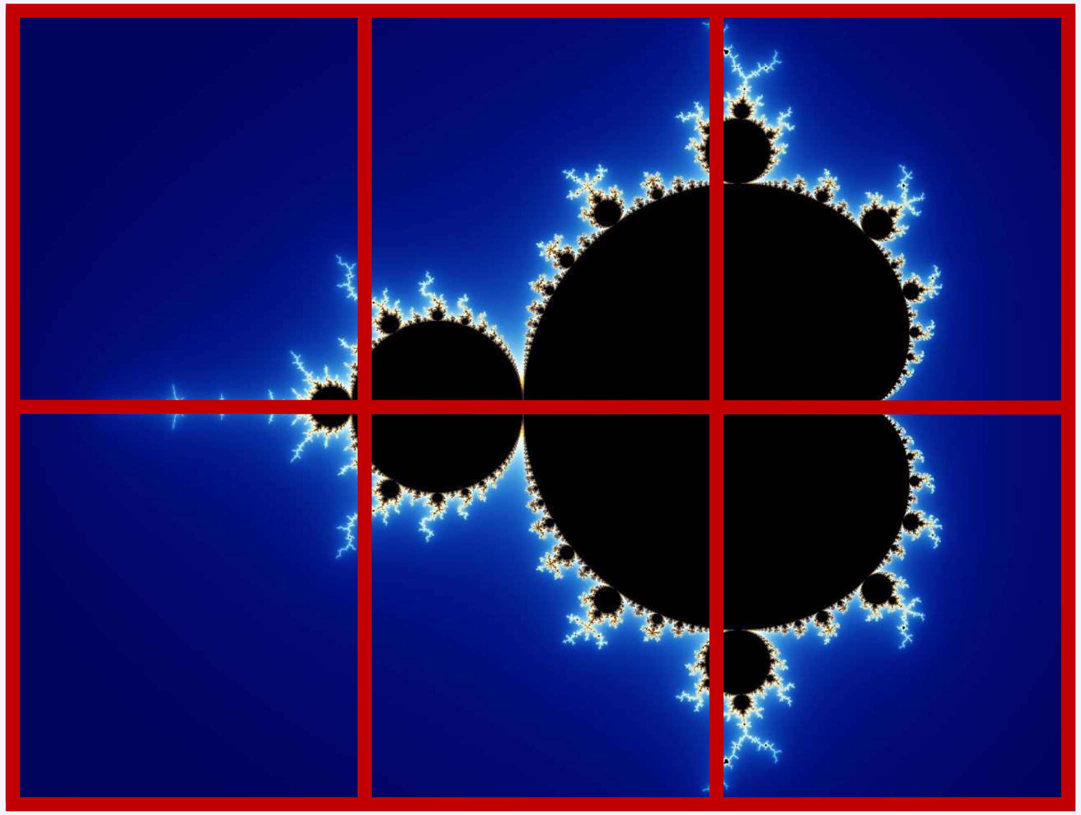
- Observe that each pixel (xy coordinate) is independent of the others
- Divide all pixels over processes and let each process work on its pixels independently => domain decomposition
- Does not require a lot of code changes:
  - Initialize domain decomposition
  - Re-use existing Mandelbrot code
  - Dump file with MPI-IO



Parallelization by domain decomposition



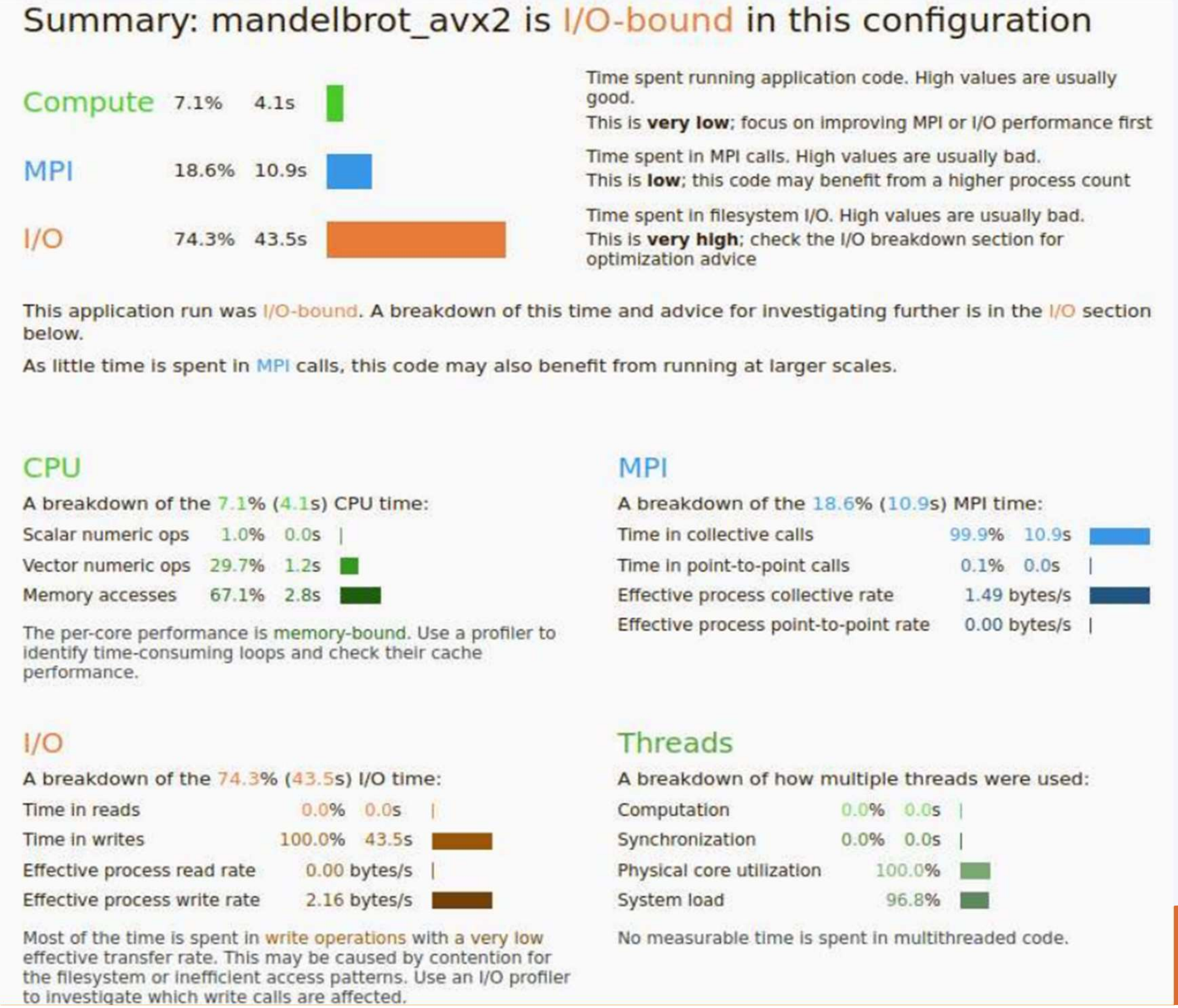
```
// Initialize domain decomposition
cart_comm = get_domain_decomposition(size, dims, &rank, coords);

// Initialize result array
int (*arr)[myNy] = malloc(sizeof(int[myNx][myNy]));

// Do the actual work
long long int niter = mandelbrot_avx2(N, myNx, myNy,
                                      myOx, myOy, arr);
// Synchronize after workload; get total number of
// iterations performed
long long int total_niter;
MPI_Allreduce(&niter, &total_niter, 1, MPI_LONG_LONG_INT,
              MPI_SUM, cart_comm);

// Write output to file
write_output_mpiio(N, myNx, myNy, myOx, myOy, cart_comm, arr);
```



Parallel scaling results: part 1

- Serial run takes 128s, parallel run on 36 cores 48s => speedup of only 2.6x, what is going on?

Parallel scaling results: part 1



Summary: mandelbrot_avx2 is I/O-bound in this configuration

Compute	7.1%	4.1s	Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first
MPI	18.6%	10.9s	Time spent in MPI calls. High values are usually bad. This is **low**; this code may benefit from a higher process count
I/O	74.3%	43.5s	Time spent in filesystem I/O. High values are usually bad. This is **very high**; check the I/O breakdown section for optimization advice

This application run was I/O-bound. A breakdown of this time and advice for investigating further is in the I/O section below.

As little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 7.1% (4.1s) CPU time:

Scalar numeric ops	1.0%	0.0s
Vector numeric ops	29.7%	1.2s
Memory accesses	67.1%	2.8s

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the 18.6% (10.9s) MPI time:

Time in collective calls	99.9%	10.9s
Time in point-to-point calls	0.1%	0.0s
Effective process collective rate	1.49 bytes/s	
Effective process point-to-point rate	0.00 bytes/s	

I/O

A breakdown of the 74.3% (43.5s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	43.5s
Effective process read rate	0.00 bytes/s	
Effective process write rate	2.16 bytes/s	

Most of the time is spent in write operations with a very low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

Threads

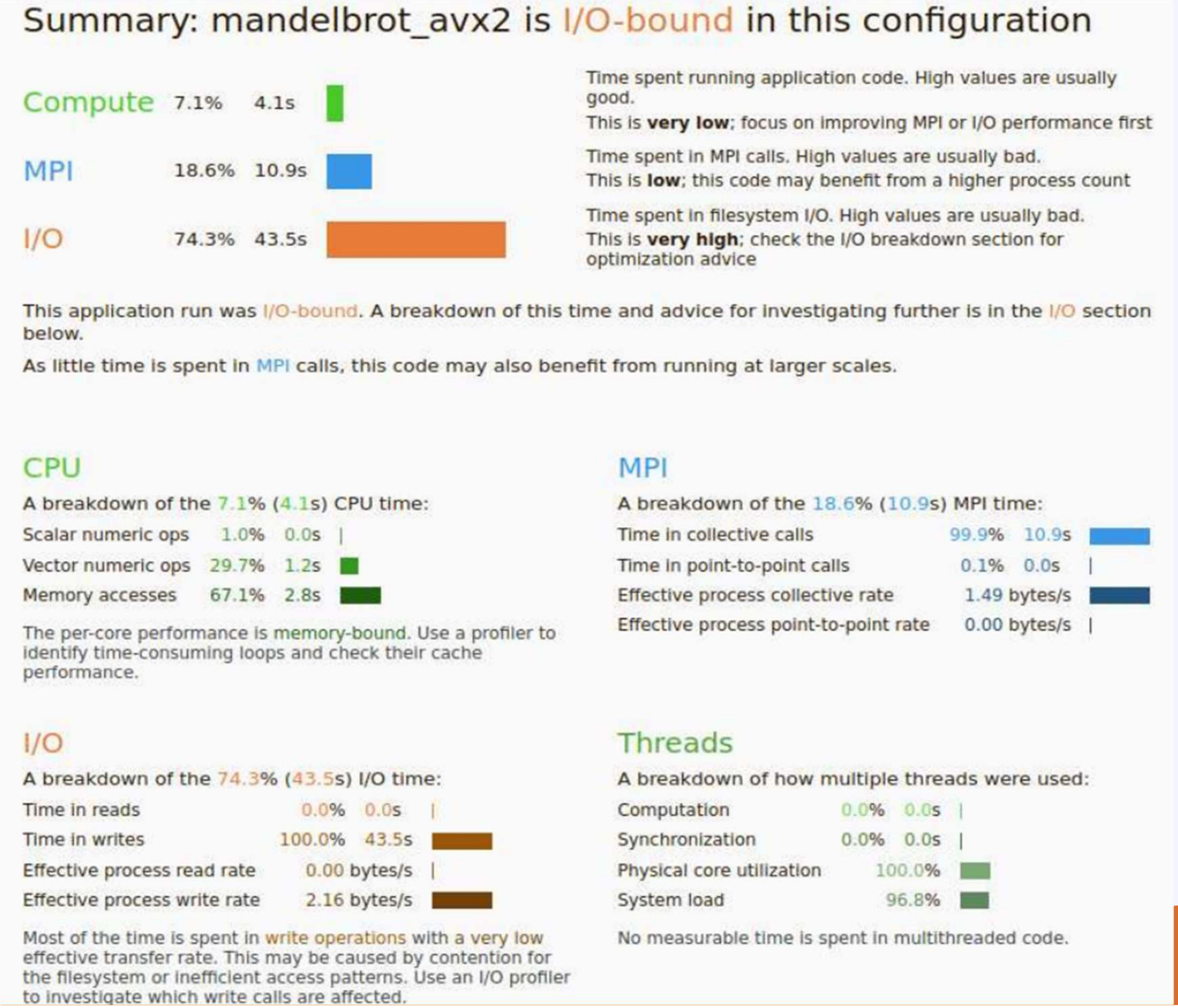
A breakdown of how multiple threads were used:

Computation	0.0%	0.0s
Synchronization	0.0%	0.0s
Physical core utilization	100.0%	
System load	96.8%	

No measurable time is spent in multithreaded code.

- Serial run takes 128s, parallel run on 36 cores 48s => speedup of only 2.6x, what is going on?

Parallel scaling results: part 1



Summary: mandelbrot_avx2 is I/O-bound in this configuration

Compute	7.1%	4.1s	Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first
MPI	18.6%	10.9s	Time spent in MPI calls. High values are usually bad. This is **low**; this code may benefit from a higher process count
I/O	74.3%	43.5s	Time spent in filesystem I/O. High values are usually bad. This is **very high**; check the I/O breakdown section for optimization advice

This application run was I/O-bound. A breakdown of this time and advice for investigating further is in the I/O section below.

As little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 7.1% (4.1s) CPU time:

Scalar numeric ops	1.0%	0.0s
Vector numeric ops	29.7%	1.2s
Memory accesses	67.1%	2.8s

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the 18.6% (10.9s) MPI time:

Time in collective calls	99.9%	10.9s
Time in point-to-point calls	0.1%	0.0s
Effective process collective rate	1.49 bytes/s	
Effective process point-to-point rate	0.00 bytes/s	

I/O

A breakdown of the 74.3% (43.5s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	43.5s
Effective process read rate	0.00 bytes/s	
Effective process write rate	2.16 bytes/s	

Most of the time is spent in write operations with a very low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

Threads

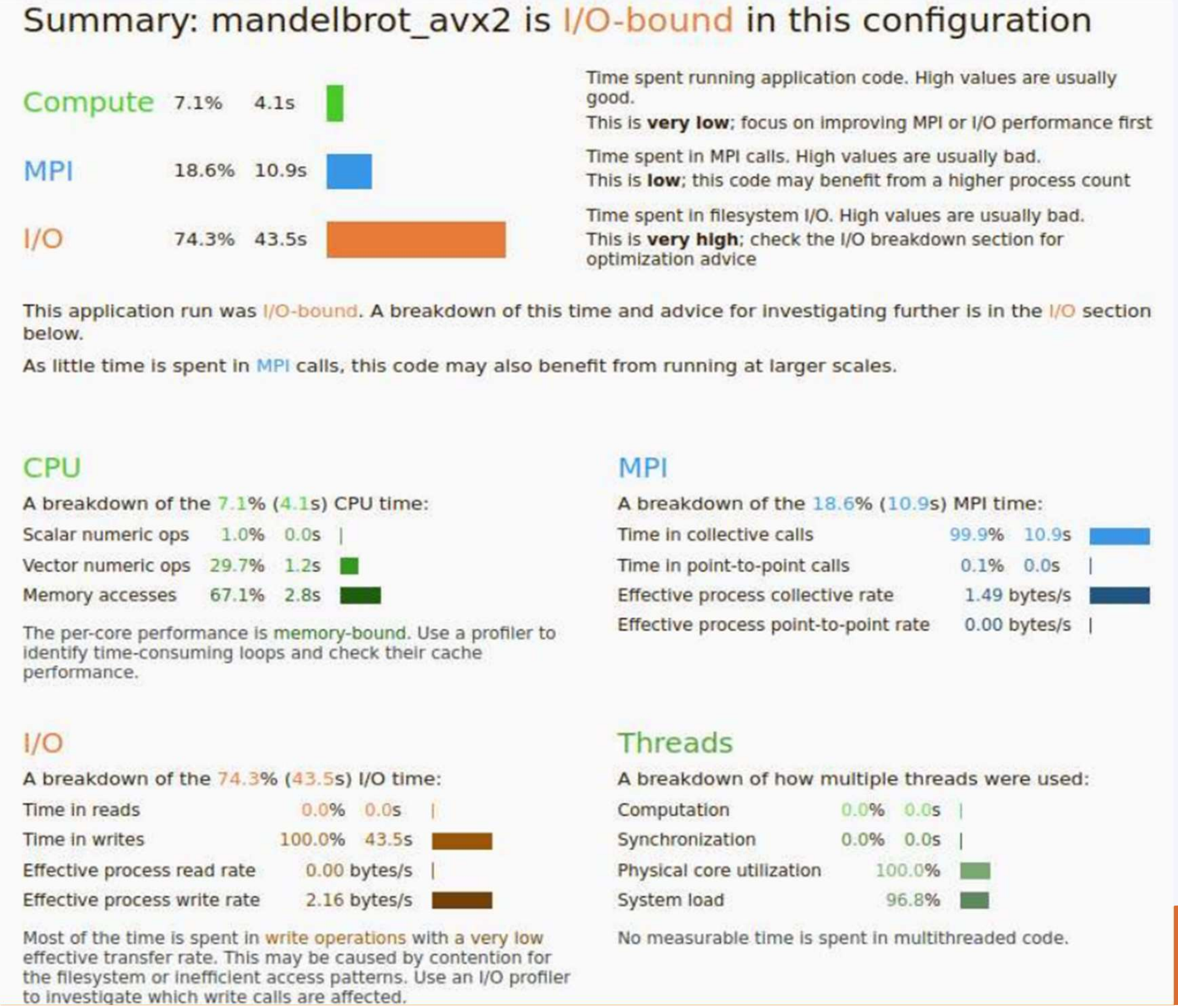
A breakdown of how multiple threads were used:

Computation	0.0%	0.0s
Synchronization	0.0%	0.0s
Physical core utilization	100.0%	
System load	96.8%	

No measurable time is spent in multithreaded code.

- Serial run takes 128s, parallel run on 36 cores 48s => speedup of only 2.6x, what is going on?

Parallel scaling results: part 1



Summary: mandelbrot_avx2 is I/O-bound in this configuration

Compute	7.1%	4.1s	Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first
MPI	18.6%	10.9s	Time spent in MPI calls. High values are usually bad. This is **low**; this code may benefit from a higher process count
I/O	74.3%	43.5s	Time spent in filesystem I/O. High values are usually bad. This is **very high**; check the I/O breakdown section for optimization advice

This application run was I/O-bound. A breakdown of this time and advice for investigating further is in the I/O section below.

As little time is spent in MPI calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the 7.1% (4.1s) CPU time:

Scalar numeric ops	1.0%	0.0s
Vector numeric ops	29.7%	1.2s
Memory accesses	67.1%	2.8s

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the 18.6% (10.9s) MPI time:

Time in collective calls	99.9%	10.9s
Time in point-to-point calls	0.1%	0.0s
Effective process collective rate	1.49 bytes/s	
Effective process point-to-point rate	0.00 bytes/s	

I/O

A breakdown of the 74.3% (43.5s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	43.5s
Effective process read rate	0.00 bytes/s	
Effective process write rate	2.16 bytes/s	

Most of the time is spent in write operations with a very low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

Threads

A breakdown of how multiple threads were used:

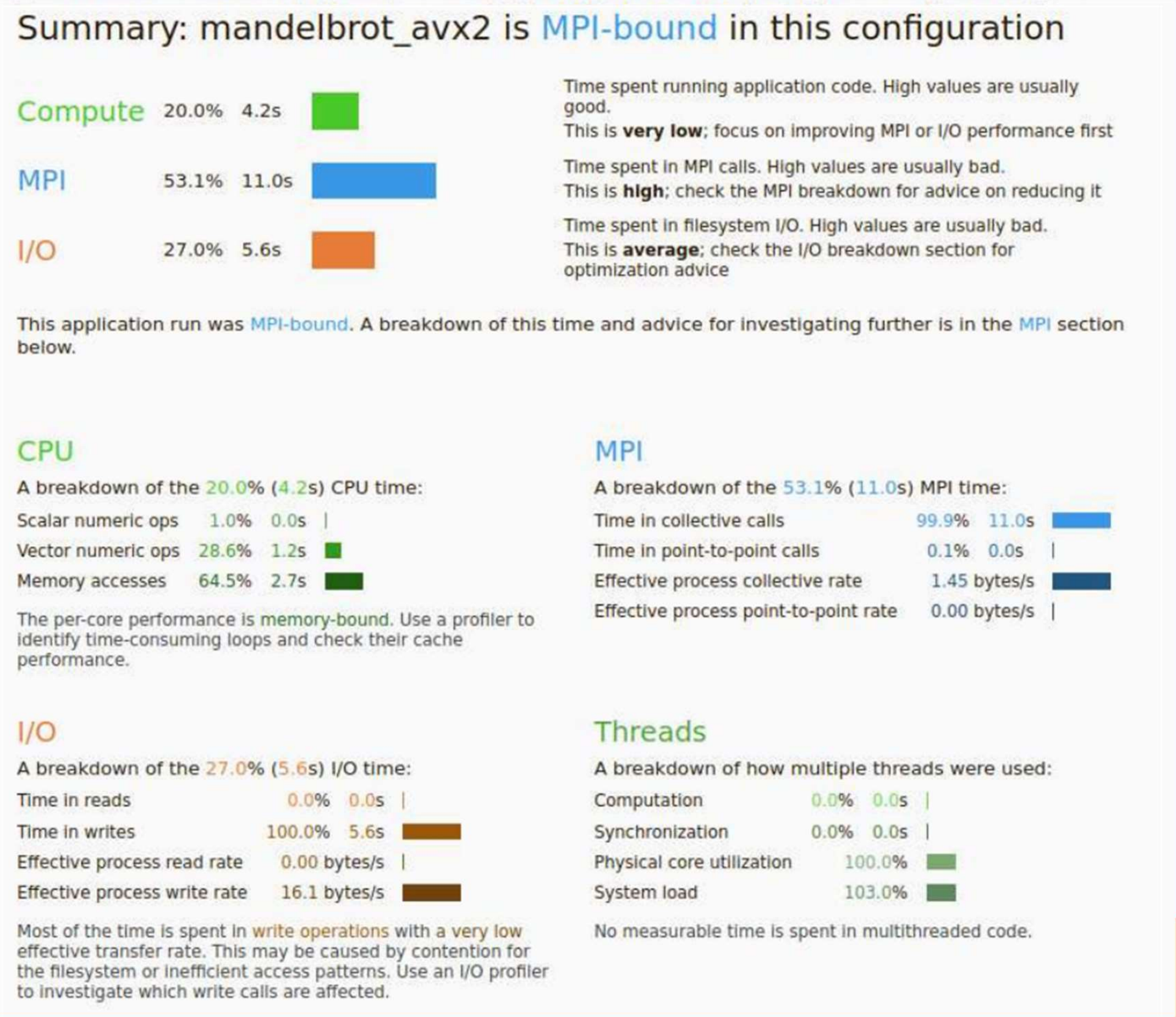
Computation	0.0%	0.0s
Synchronization	0.0%	0.0s
Physical core utilization	100.0%	
System load	96.8%	

No measurable time is spent in multithreaded code.

- Serial run takes 128s,
  parallel run on 36 cores 48s
  => speedup of only 2.6x,
  what is going on?
- I used ${VSC_DATA} to write output, this is not suited for parallel I/O
- Using ${VSC_SCRATCH} instead brings the runtime down to 23s

VLAAMS
SUPERCOMPUTER
CENTRUM

Parallel scaling results: part 1



Summary: mandelbrot_avx2 is MPI-bound in this configuration

Compute	20.0%	4.2s	Time spent running application code. High values are usually good. This is **very low**; focus on improving MPI or I/O performance first
MPI	53.1%	11.0s	Time spent in MPI calls. High values are usually bad. This is **high**; check the MPI breakdown for advice on reducing it
I/O	27.0%	5.6s	Time spent in filesystem I/O. High values are usually bad. This is **average**; check the I/O breakdown section for optimization advice

This application run was MPI-bound. A breakdown of this time and advice for investigating further is in the MPI section below.

CPU

A breakdown of the 20.0% (4.2s) CPU time:

Scalar numeric ops	1.0%	0.0s
Vector numeric ops	28.6%	1.2s
Memory accesses	64.5%	2.7s

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the 53.1% (11.0s) MPI time:

Time in collective calls	99.9%	11.0s
Time in point-to-point calls	0.1%	0.0s
Effective process collective rate		1.45 bytes/s
Effective process point-to-point rate		0.00 bytes/s

I/O

A breakdown of the 27.0% (5.6s) I/O time:

Time in reads	0.0%	0.0s
Time in writes	100.0%	5.6s
Effective process read rate		0.00 bytes/s
Effective process write rate		16.1 bytes/s

Most of the time is spent in write operations with a very low effective transfer rate. This may be caused by contention for the filesystem or inefficient access patterns. Use an I/O profiler to investigate which write calls are affected.

Threads

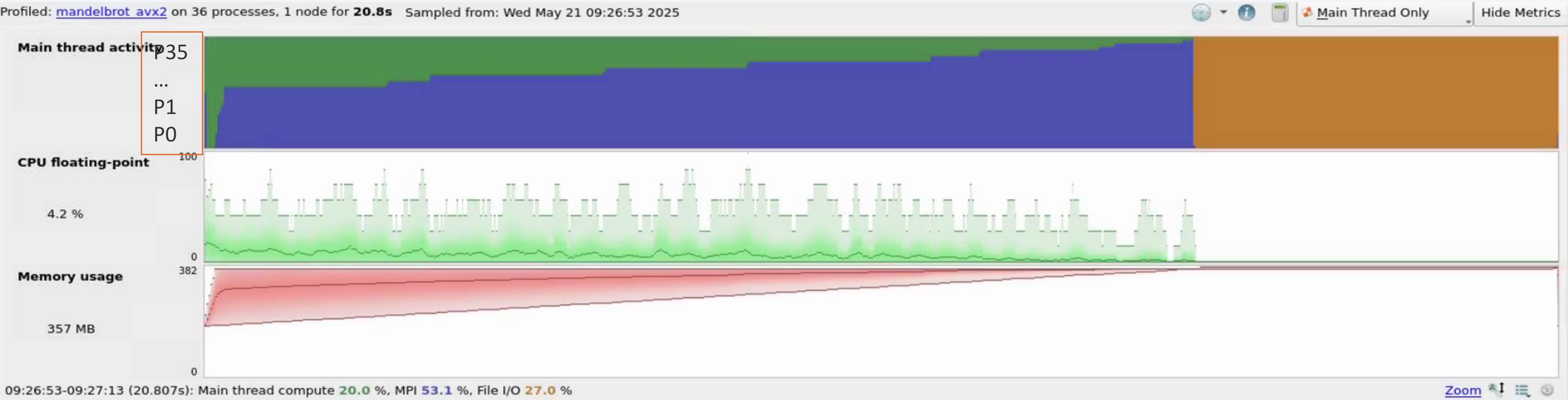
A breakdown of how multiple threads were used:

Computation	0.0%	0.0s
Synchronization	0.0%	0.0s
Physical core utilization	100.0%	
System load	103.0%	

No measurable time is spent in multithreaded code.

- Serial run takes 128s,
  parallel run on 36 cores 48s
  => speedup of only 2.6x,
  what is going on?
- I used ${VSC_DATA} to write output, this is not suited for parallel I/O
- Using ${VSC_SCRATCH} instead brings the runtime down to 23s
- Code now becomes MPI-bound

Parallel scaling results: part 2

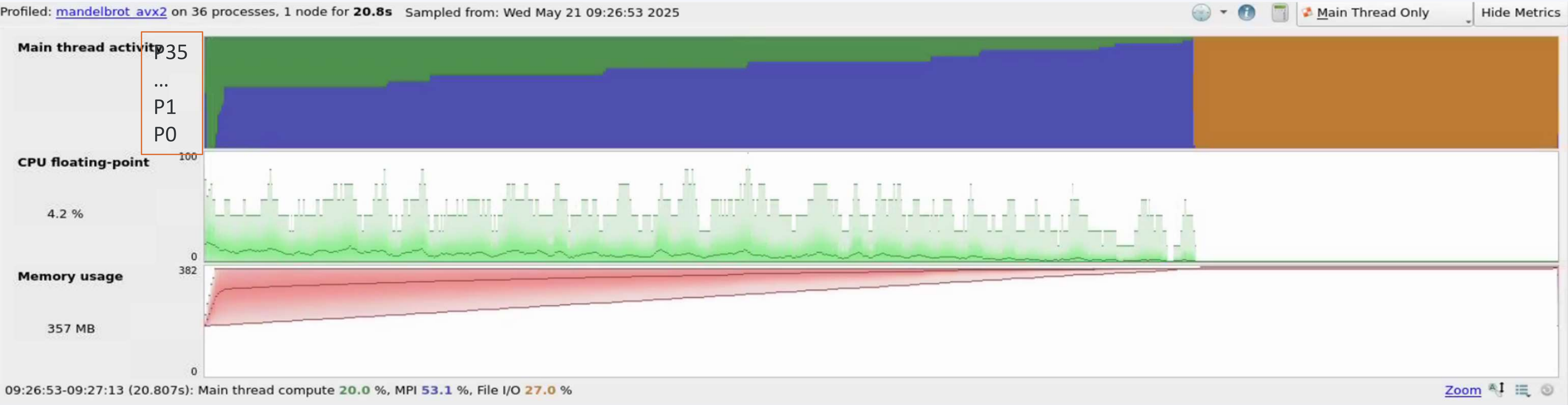


Profiled: mandelbrot_avx2 on 36 processes, 1 node for **20.8s** Sampled from: Wed May 21 09:26:53 2025

09:26:53-09:27:13 (20.807s): Main thread compute 20.0 %, MPI 53.1 %, File I/O 27.0 %

- Some processes are spending most time in “compute” (green), other processes are spending most time in “MPI” (blue):
  => load imbalance

Parallel scaling results: part 2

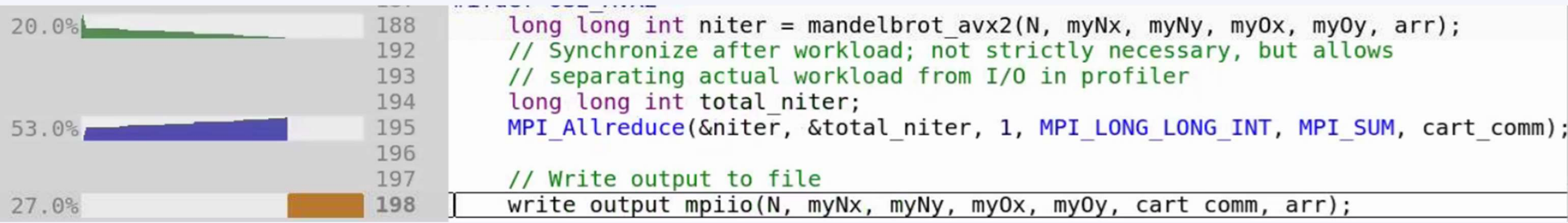


Profiled: mandelbrot_avx2 on 36 processes, 1 node for **20.8s** Sampled from: Wed May 21 09:26:53 2025

09:26:53-09:27:13 (20.807s): Main thread compute 20.0 %, MPI 53.1 %, File I/O 27.0 %

- Some processes are spending most time in “compute” (green), other processes are spending most time in “MPI” (blue):
  => load imbalance
- Some processes do more iterations than others

Load imbalance in the source code view

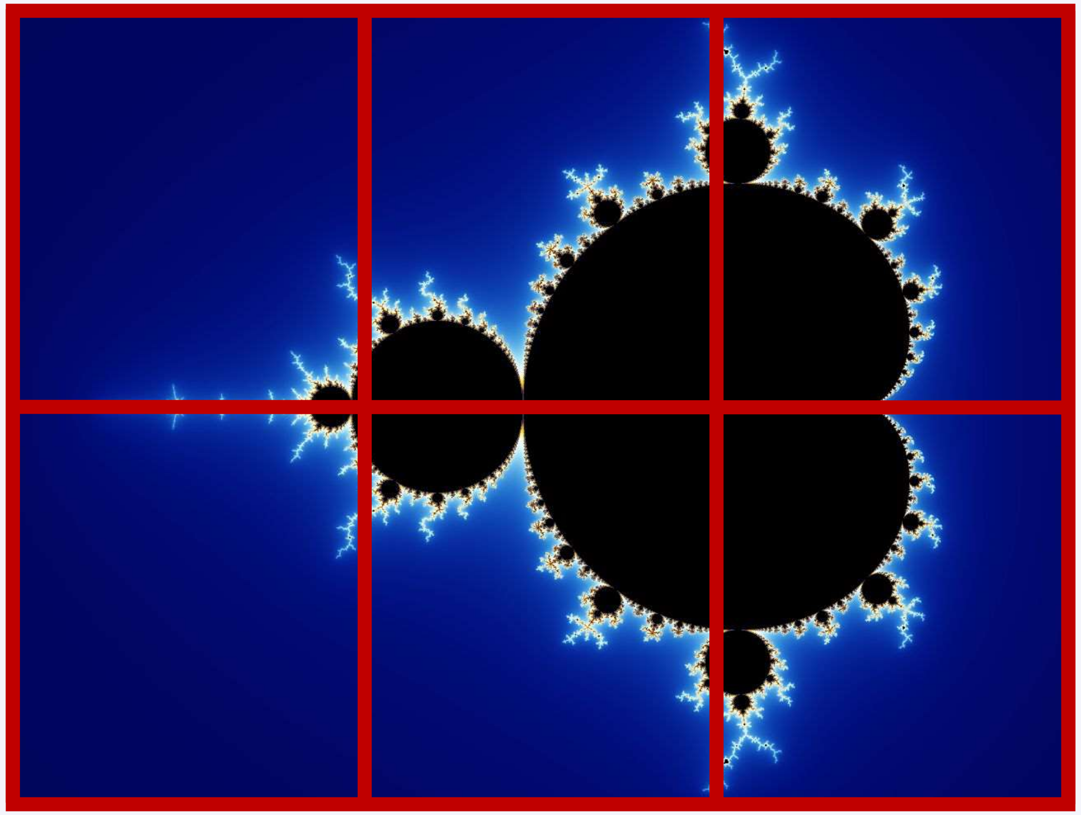


```
    long long int niter = mandelbrot_avx2(N, myNx, myNy, myOx, myOy, arr);
    // Synchronize after workload; not strictly necessary, but allows
    // separating actual workload from I/O in profiler
    long long int total_niter;
    MPI_Allreduce(&niter, &total_niter, 1, MPI_LONG_LONG_INT, MPI_SUM, cart_comm);

    // Write output to file
    write output mpiio(N, myNx, myNy, myOx, myOy, cart comm, arr);
```

- Some processes are spending most time in “compute” (green), other processes are spending most time in “MPI” (blue): => load imbalance
- Some processes do more iterations than others

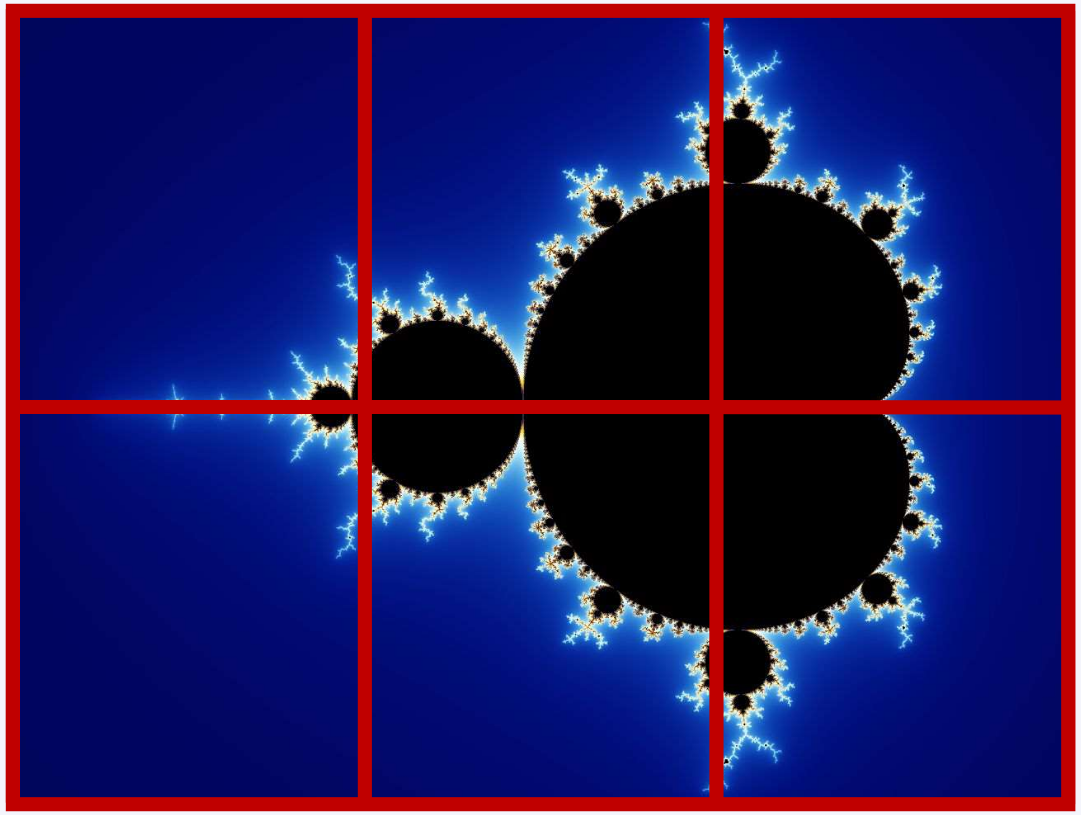
Domain decomposition causes load imbalance

- Color indicates number of iterations
- Total number of iterations per domain has a high variability: this decomposition has a built-in load imbalance



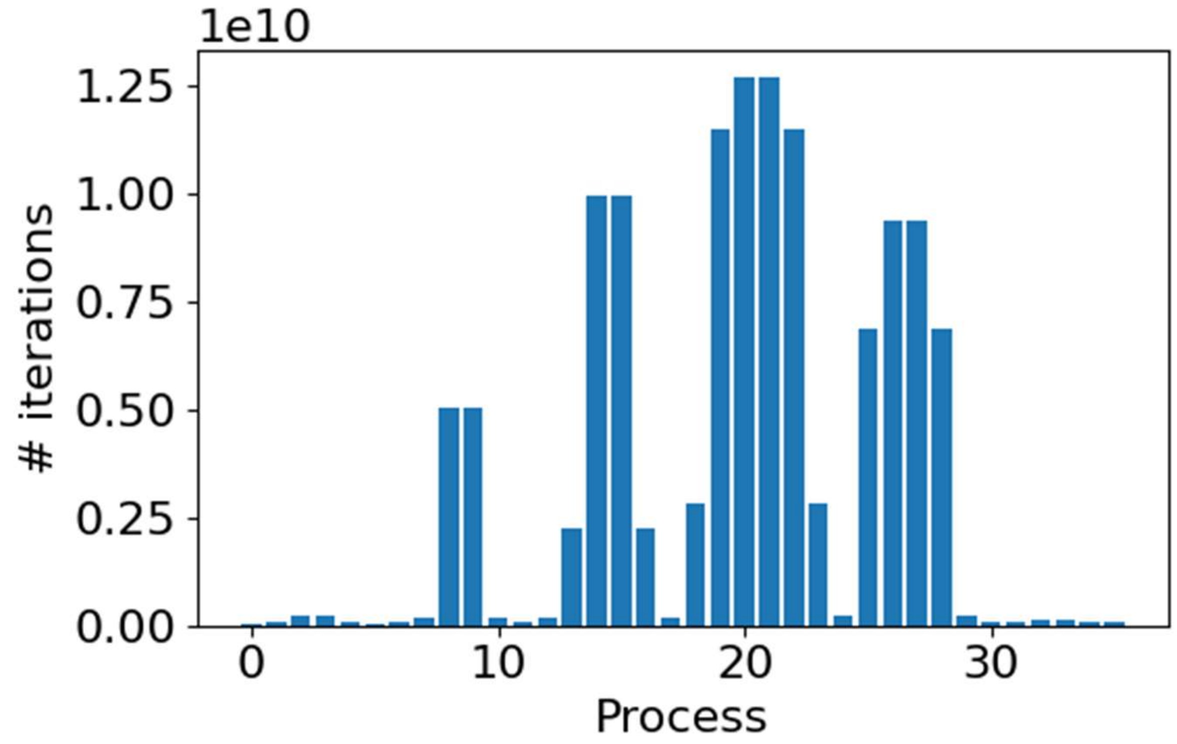
Domain decomposition causes load imbalance

- Color indicates number of iterations
- Total number of iterations per domain has a high variability: this decomposition has a built-in load imbalance
- Possible solution:
  Use smaller domains; one process acts as broker to assign domains to processes



Domain decomposition causes load imbalance

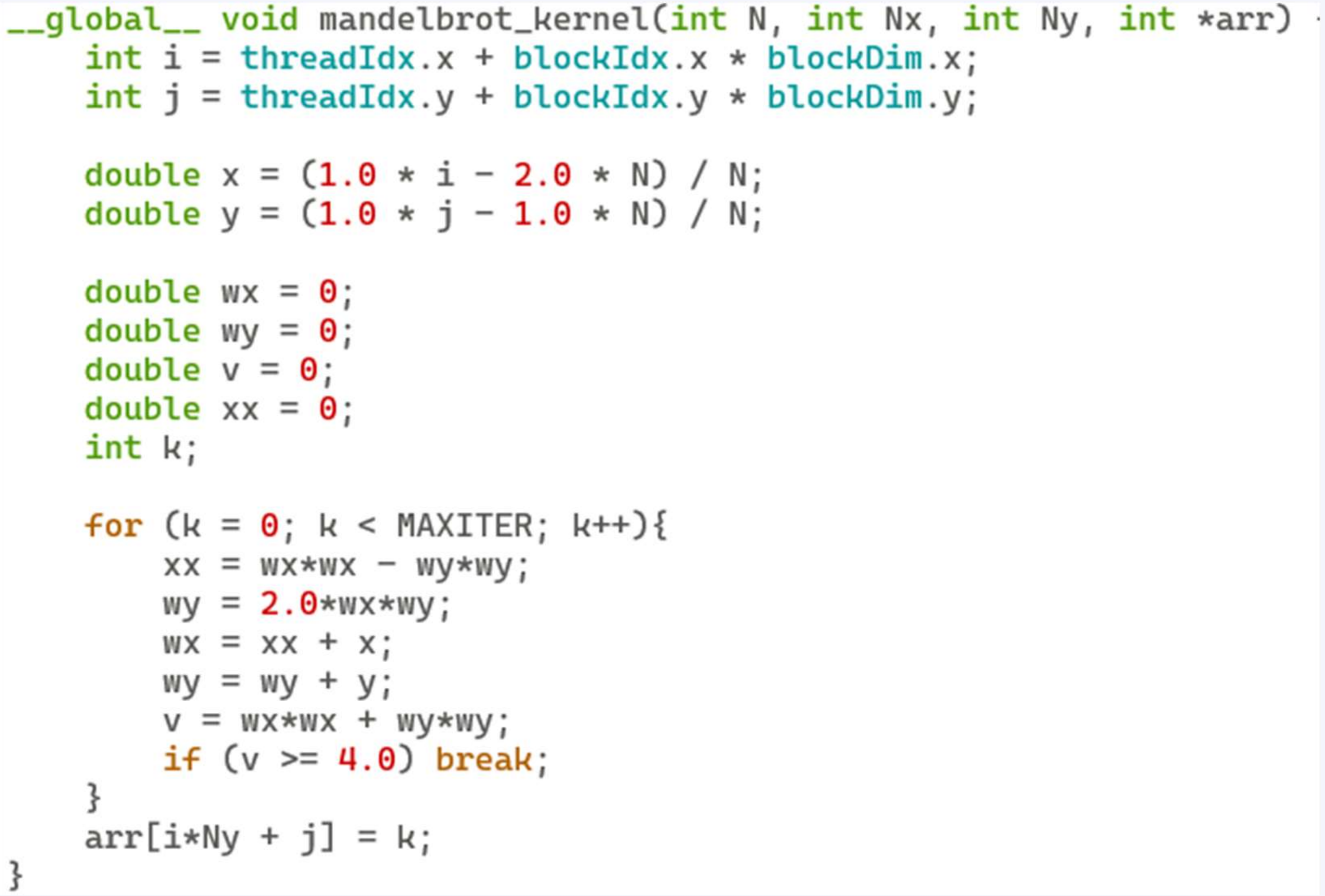
- Color indicates number of iterations
- Total number of iterations per domain has a high variability: this decomposition has a built-in load imbalance
- Possible solution:
  Use smaller domains; one process acts as broker to assign domains to processes



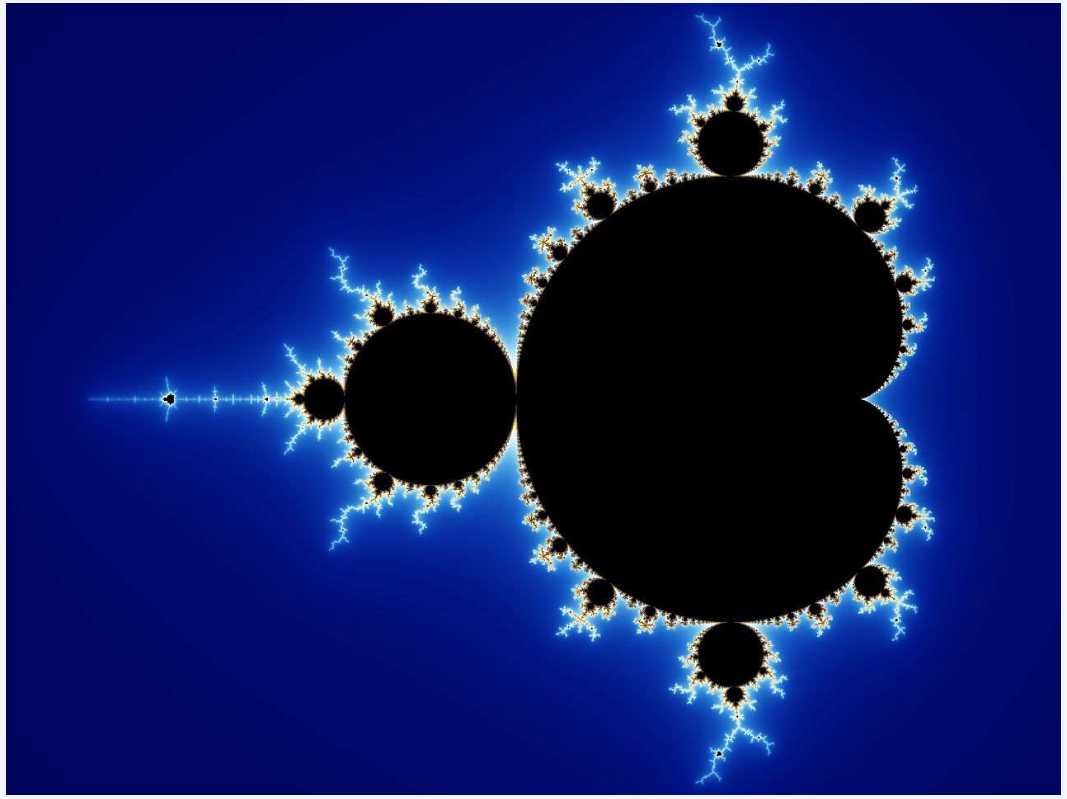
Agenda

- Introduction to and overview of profilers
- Examples using Linaro MAP
  - Serial CPU code
  - MPI parallel CPU code
  - **CUDA GPU code**

The Mandelbrot set in CUDA

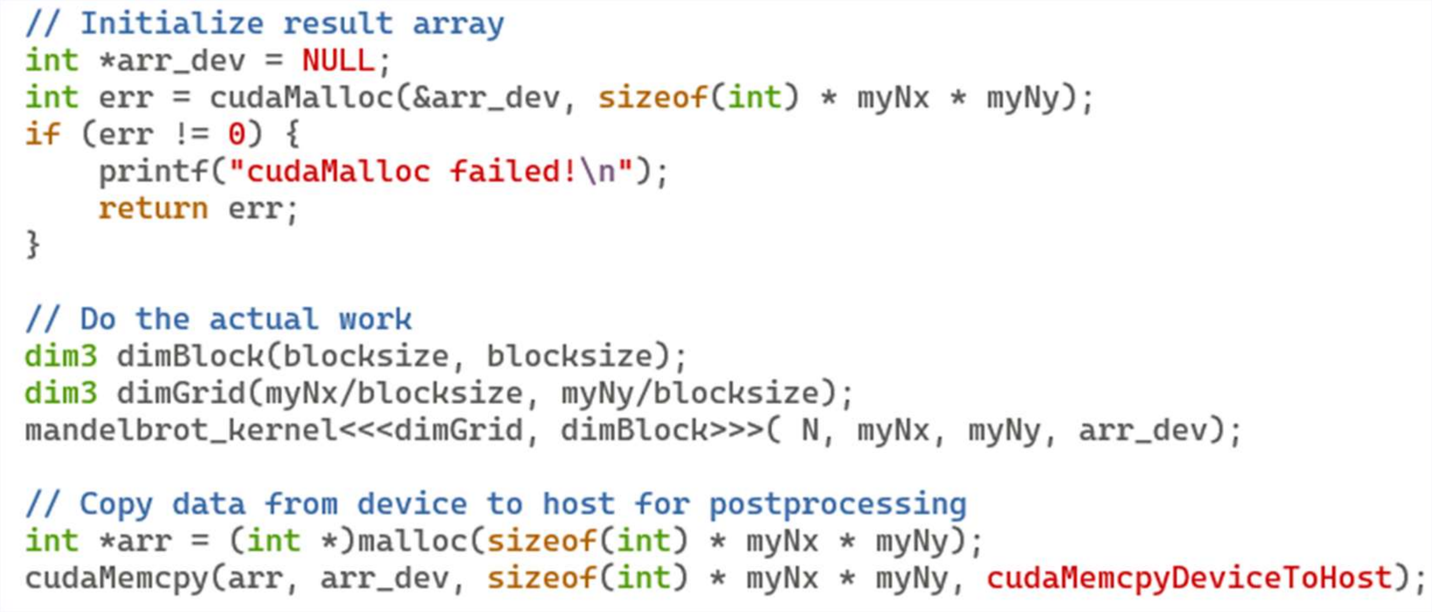


```
__global__ void mandelbrot_kernel(int N, int Nx, int Ny, int *arr)
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int j = threadIdx.y + blockIdx.y * blockDim.y;

    double x = (1.0 * i - 2.0 * N) / N;
    double y = (1.0 * j - 1.0 * N) / N;

    double wx = 0;
    double wy = 0;
    double v = 0;
    double xx = 0;
    int k;

    for (k = 0; k < MAXITER; k++){
        xx = wx*wx - wy*wy;
        wy = 2.0*wx*wy;
        wx = xx + x;
        wy = wy + y;
        v = wx*wx + wy*wy;
        if (v >= 4.0) break;
    }
    arr[i*Ny + j] = k;
}
```

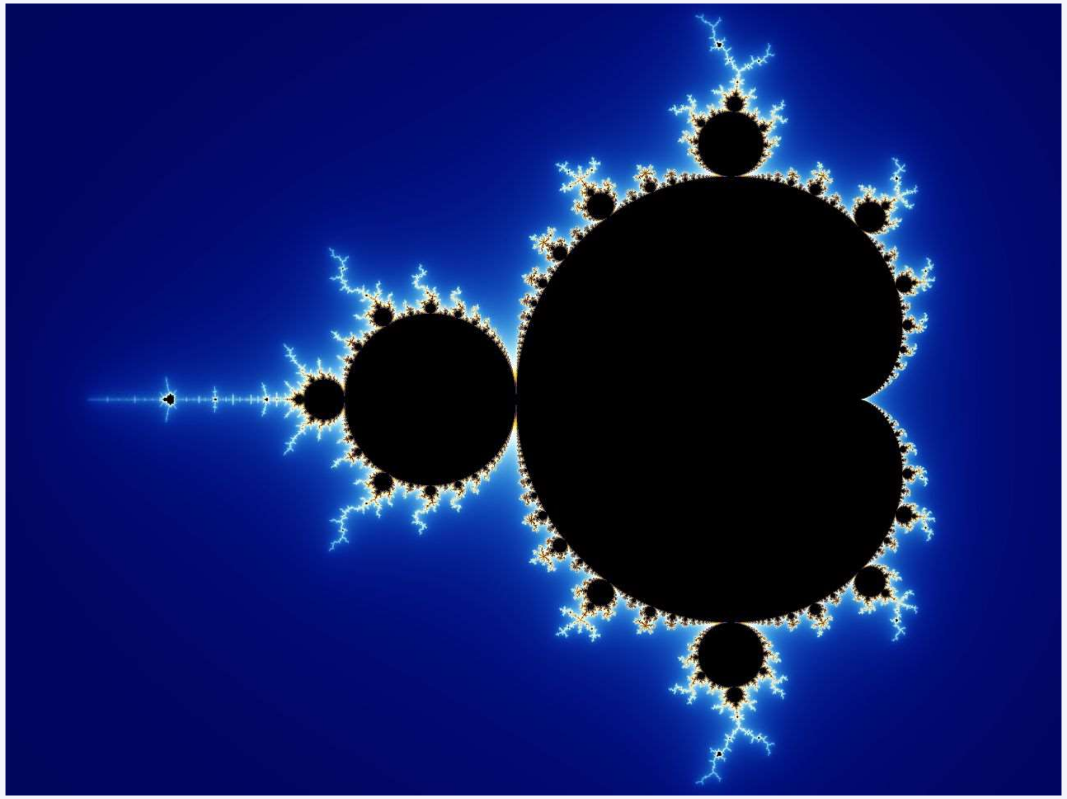


- CUDA kernel code looks quite similar to C code

The Mandelbrot set in CUDA

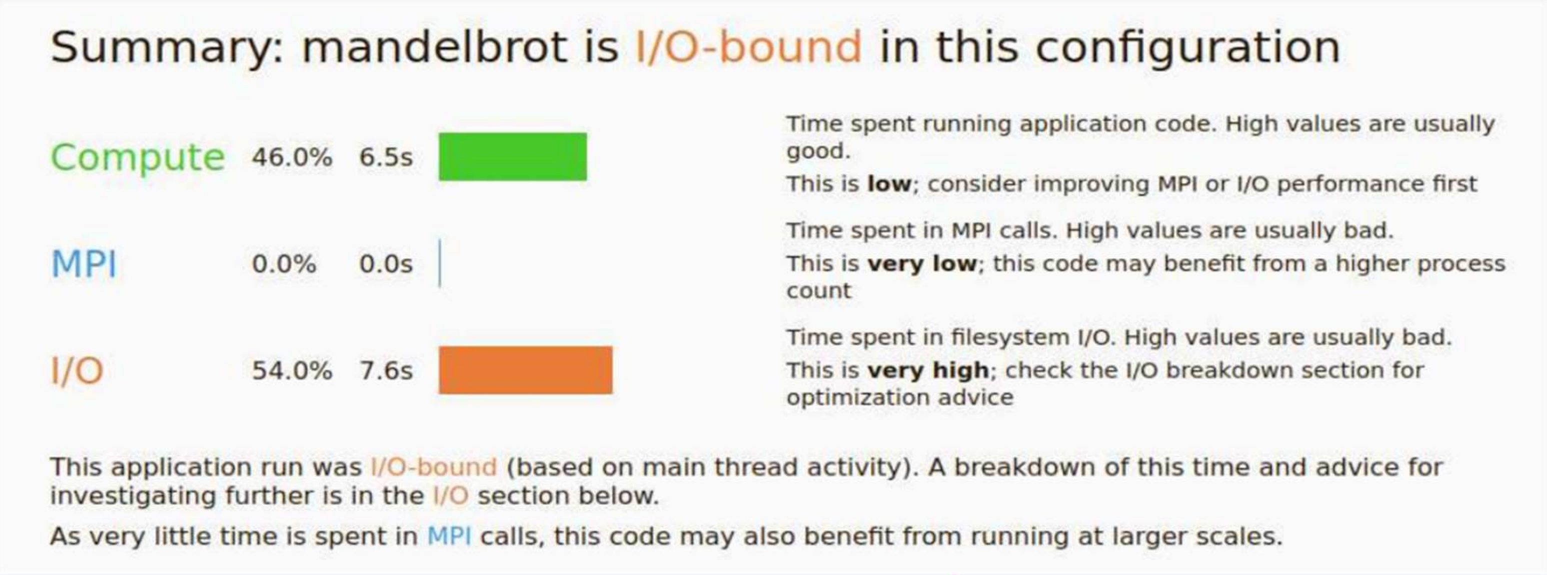


```
// Initialize result array
int *arr_dev = NULL;
int err = cudaMalloc(&arr_dev, sizeof(int) * myNx * myNy);
if (err != 0) {
    printf("cudaMalloc failed!\n");
    return err;
}

// Do the actual work
dim3 dimBlock(blocksize, blocksize);
dim3 dimGrid(myNx/blocksize, myNy/blocksize);
mandelbrot_kernel<<<dimGrid, dimBlock>>>( N, myNx, myNy, arr_dev);

// Copy data from device to host for postprocessing
int *arr = (int *)malloc(sizeof(int) * myNx * myNy);
cudaMemcpy(arr, arr_dev, sizeof(int) * myNx * myNy, cudaMemcpyDeviceToHost);
```



- Allocate result array on GPU
- Launch kernel
- Copy back results before writing file

CUDA performance report

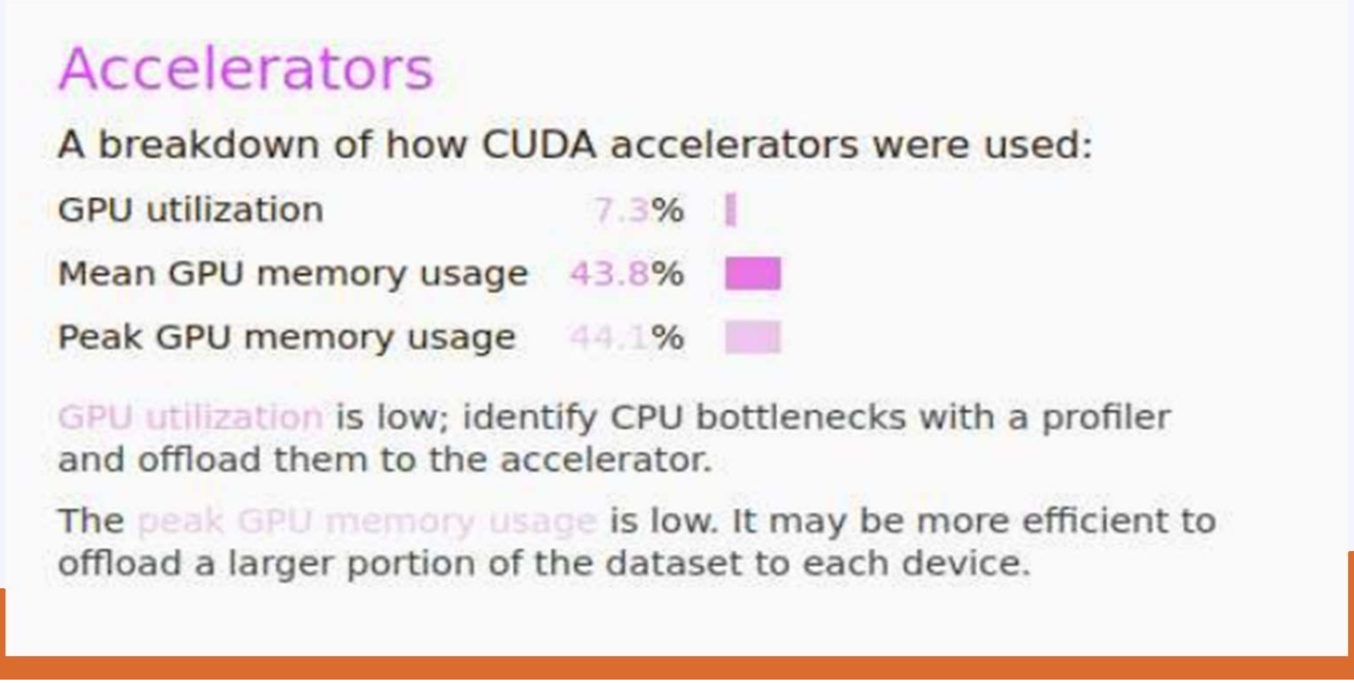


Summary: mandelbrot is I/O-bound in this configuration

Compute	46.0%	6.5s	🟩	Time spent running application code. High values are usually good. This is **low**; consider improving MPI or I/O performance first
MPI	0.0%	0.0s		Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from a higher process count
I/O	54.0%	7.6s	🟧	Time spent in filesystem I/O. High values are usually bad. This is **very high**; check the I/O breakdown section for optimization advice

This application run was I/O-bound (based on main thread activity). A breakdown of this time and advice for investigating further is in the I/O section below.

As very little time is spent in MPI calls, this code may also benefit from running at larger scales.



Accelerators

A breakdown of how CUDA accelerators were used:

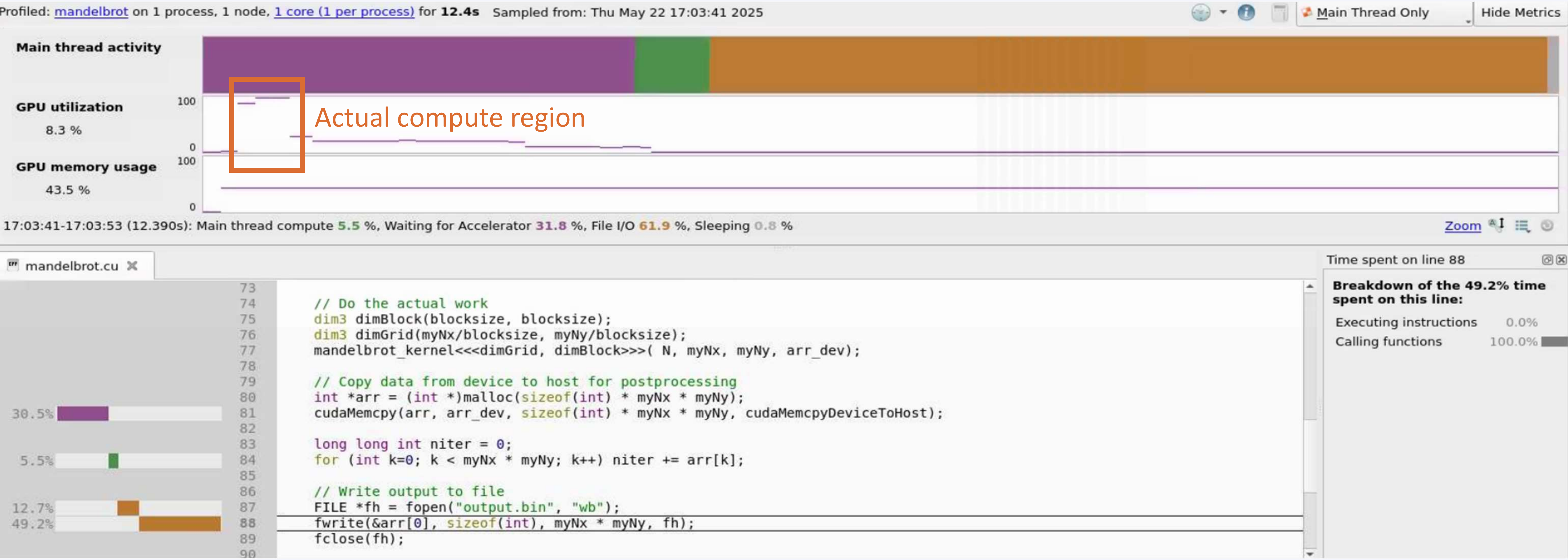
GPU utilization	7.3%	▮
Mean GPU memory usage	43.8%	▬
Peak GPU memory usage	44.1%	▬

GPU utilization is low; identify CPU bottlenecks with a profiler and offload them to the accelerator.

The peak GPU memory usage is low. It may be more efficient to offload a larger portion of the dataset to each device.

- Code becomes I/O bound again
- GPU is so fast at "compute" portion, the GPU utilization is low
- This particular code is probably not very well suited for GPU…

CUDA timeline view



Profiled: mandelbrot on 1 process, 1 node, 1 core (1 per process) for **12.4s** Sampled from: Thu May 22 17:03:41 2025

Actual compute region

17:03:41-17:03:53 (12.390s): Main thread compute 5.5 %, Waiting for Accelerator 31.8 %, File I/O 61.9 %, Sleeping 0.8 %

```
        73
        74     // Do the actual work
        75     dim3 dimBlock(blocksize, blocksize);
        76     dim3 dimGrid(myNx/blocksize, myNy/blocksize);
        77     mandelbrot_kernel<<<dimGrid, dimBlock>>>( N, myNx, myNy, arr_dev);
        78
        79     // Copy data from device to host for postprocessing
        80     int *arr = (int *)malloc(sizeof(int) * myNx * myNy);
30.5%   81     cudaMemcpy(arr, arr_dev, sizeof(int) * myNx * myNy, cudaMemcpyDeviceToHost);
        82
        83     long long int niter = 0;
5.5%    84     for (int k=0; k < myNx * myNy; k++) niter += arr[k];
        85
        86     // Write output to file
12.7%   87     FILE *fh = fopen("output.bin", "wb");
49.2%   88     fwrite(&arr[0], sizeof(int), myNx * myNy, fh);
        89     fclose(fh);
        90
```

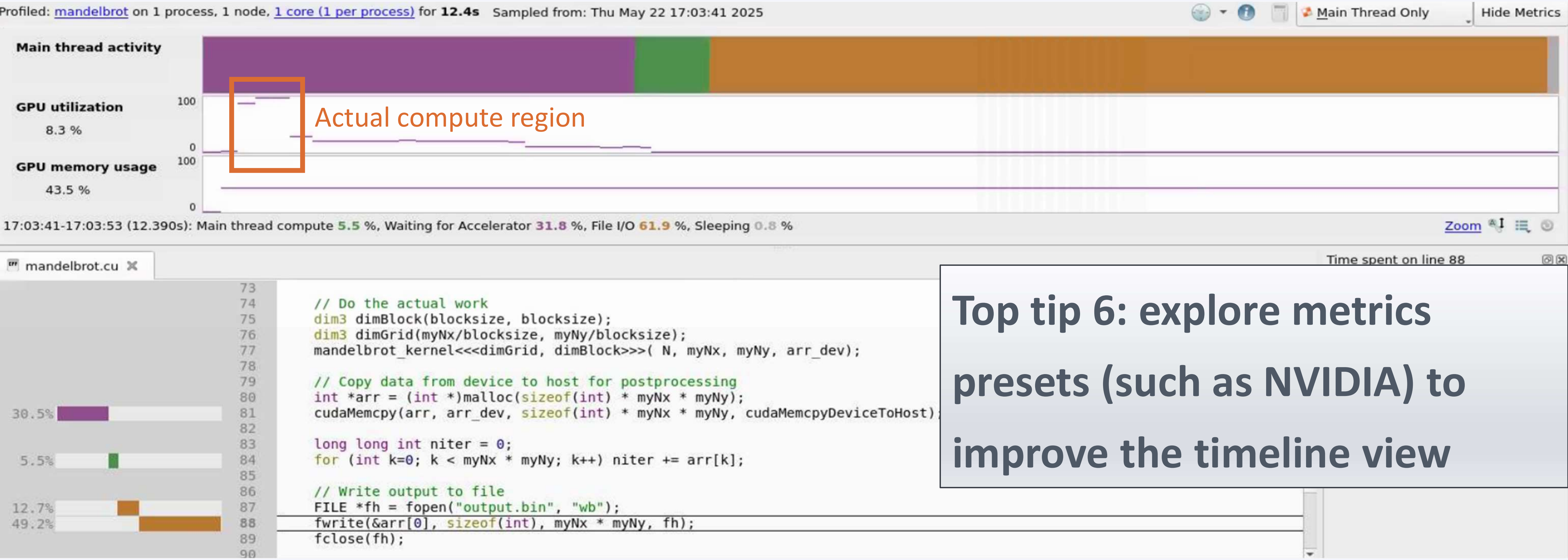
Time spent on line 88

Breakdown of the 49.2% time spent on this line:

Executing instructions	0.0%
Calling functions	100.0%

- Kernel is launched asynchronously => does not show up in code view
- Most time is spent transferring data from GPU to CPU and from CPU to file

CUDA timeline view



Profiled: mandelbrot on 1 process, 1 node, 1 core (1 per process) for 12.4s Sampled from: Thu May 22 17:03:41 2025

Actual compute region

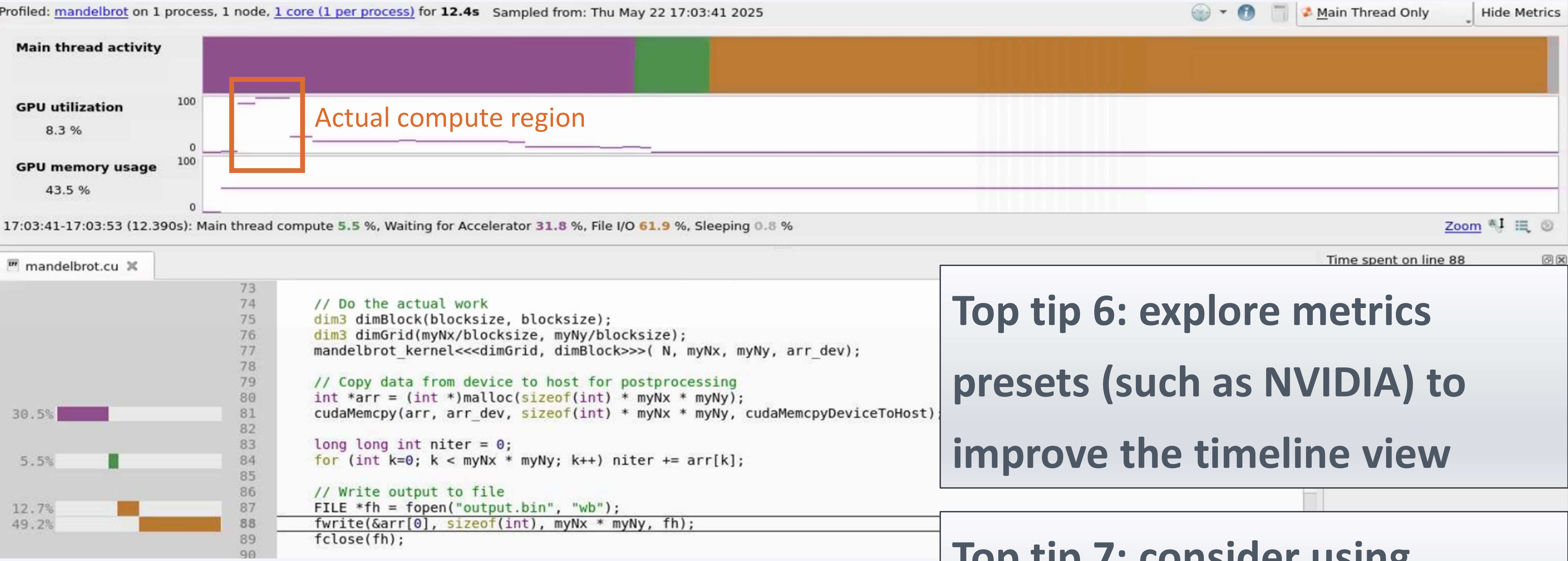
17:03:41-17:03:53 (12.390s): Main thread compute 5.5 %, Waiting for Accelerator 31.8 %, File I/O 61.9 %, Sleeping 0.8 %

```
       73
       74      // Do the actual work
       75      dim3 dimBlock(blocksize, blocksize);
       76      dim3 dimGrid(myNx/blocksize, myNy/blocksize);
       77      mandelbrot_kernel<<<dimGrid, dimBlock>>>( N, myNx, myNy, arr_dev);
       78
       79      // Copy data from device to host for postprocessing
       80      int *arr = (int *)malloc(sizeof(int) * myNx * myNy);
30.5%  81      cudaMemcpy(arr, arr_dev, sizeof(int) * myNx * myNy, cudaMemcpyDeviceToHost);
       82
       83      long long int niter = 0;
 5.5%  84      for (int k=0; k < myNx * myNy; k++) niter += arr[k];
       85
       86      // Write output to file
12.7%  87      FILE *fh = fopen("output.bin", "wb");
49.2%  88      fwrite(&arr[0], sizeof(int), myNx * myNy, fh);
       89      fclose(fh);
       90
```

Top tip 6: explore metrics presets (such as NVIDIA) to improve the timeline view

- Kernel is launched asynchronously => does not show up in code view
- Most time is spent transferring data from GPU to CPU and from CPU to file

CUDA timeline view



Profiled: mandelbrot on 1 process, 1 node, 1 core (1 per process) for **12.4s** Sampled from: Thu May 22 17:03:41 2025

17:03:41-17:03:53 (12.390s): Main thread compute 5.5 %, Waiting for Accelerator 31.8 %, File I/O 61.9 %, Sleeping 0.8 %

```
                  73
                  74      // Do the actual work
                  75      dim3 dimBlock(blocksize, blocksize);
                  76      dim3 dimGrid(myNx/blocksize, myNy/blocksize);
                  77      mandelbrot_kernel<<<dimGrid, dimBlock>>>( N, myNx, myNy, arr_dev);
                  78
                  79      // Copy data from device to host for postprocessing
                  80      int *arr = (int *)malloc(sizeof(int) * myNx * myNy);
30.5%             81      cudaMemcpy(arr, arr_dev, sizeof(int) * myNx * myNy, cudaMemcpyDeviceToHost)
                  82
                  83      long long int niter = 0;
 5.5%             84      for (int k=0; k < myNx * myNy; k++) niter += arr[k];
                  85
                  86      // Write output to file
12.7%             87      FILE *fh = fopen("output.bin", "wb");
49.2%             88      fwrite(&arr[0], sizeof(int), myNx * myNy, fh);
                  89      fclose(fh);
                  90
```

Top tip 6: explore metrics presets (such as NVIDIA) to improve the timeline view

Top tip 7: consider using specialized profiler (NVIDIA Nsight) for NVIDIA GPU runs

- Kernel is launched asynchronously => does not show u
- Most time is spent transferring data from GPU to CPU a

Conclusions

- Using a profiler gives insight into performance of your code

Conclusions

- Using a profiler gives insight into performance of your code
- Many profilers suited for HPC to choose from, combining tools can be a good idea
- LinaroForge MAP is a good starting point, main limitation is the number of license tokens

Conclusions

- Using a profiler gives insight into performance of your code
- Many profilers suited for HPC to choose from, combining tools can be a good idea
- LinaroForge MAP is a good starting point, main limitation is the number of license tokens
- Code examples are available at https://github.com/hpcleuven/code-profiling-workshop
- This presentation was inspired by https://gjbex.github.io/Code-optimization/
  => recommended to learn about code optimization (next run in 2026)
  => keep an eye on https://www.vscentrum.be/vsctraining