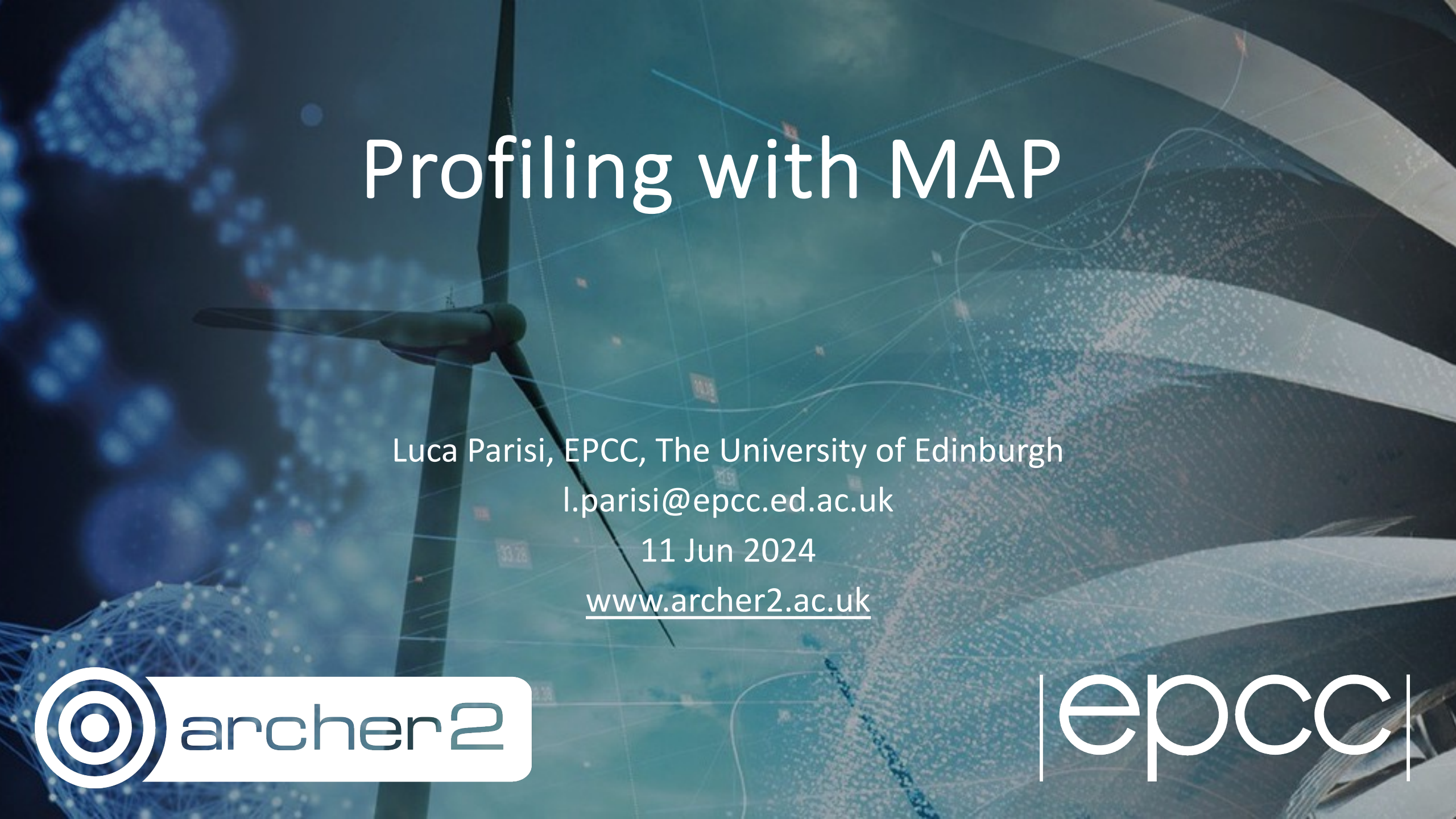


Profiling with MAP

Luca Parisi, EPCC, The University of Edinburgh
l.parisi@epcc.ed.ac.uk

11 Jun 2024

www.archer2.ac.uk

archer2

epcc

Outline

- Introduction to sampling
- Hotspot analysis
- MPI communication
- Memory usage
- Shared memory with OpenMP

Introduction to sampling

Sampling vs tracing

Sampling	Tracing
Interrupts execution at constant frequency	Tracks entering/exiting function calls
Statistical representation of program execution. Might miss some very quick function calls.	All functions are correctly represented
Usually requires relinking. This can sometimes be done automatically	Often requires recompiling

- MAP is a sampling only library
- Tracing is possible using other tools like CRAY-PAT, ITAC , SCOREP …

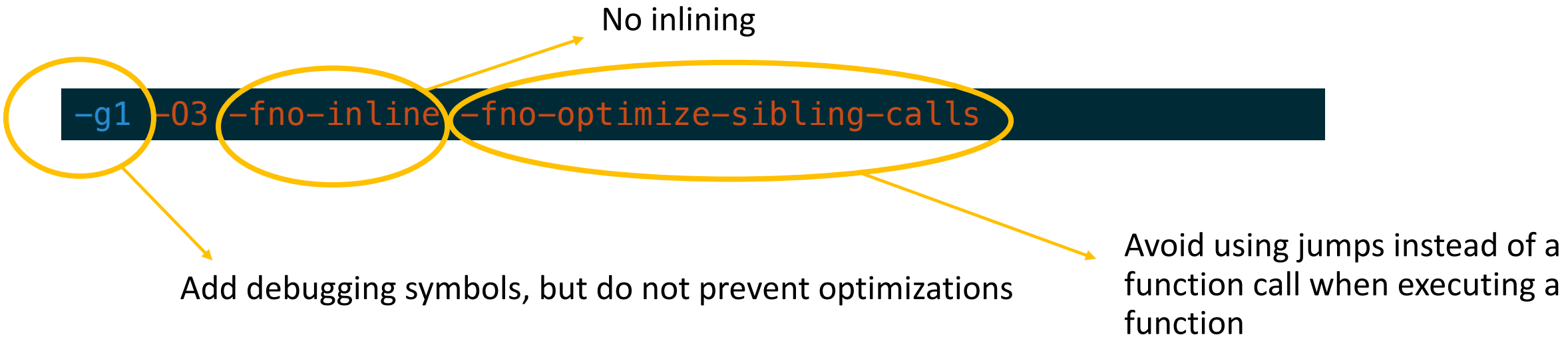
MAP

- ✔ MAP is a sampling library
- ✘ MAP is not a tracing library. Can use other tools like CRAY-PAT, ITAC , SCOREP …
- ✔ MAP is very easy to use and a very simple and intuitive interface
- ✘ MAP can be imprecise
- ✘ Map lacks some more advanced features (PAPI counters, memory allocations)
- ✔ MAP gives a very quick and clear overview of performance bottlenecks

Using map

Compiling

- Usually only needs to compile with debugging symbols and optimization turned on.
- Inlining can confuse the profiler
- Tail recursion optimization can confuse the stack record



Launching map

Launch from a submission script

Launch with SLURM. Alternatively use mpi implementation, such as mpi or mpich

Number of processes

```
map -n 36 --profile --mpi=slurm --mpiargs="--hint=nomultithread --
distribution=block:block" ./test
```

Profile from CLI , without opening the GUI

The .map files

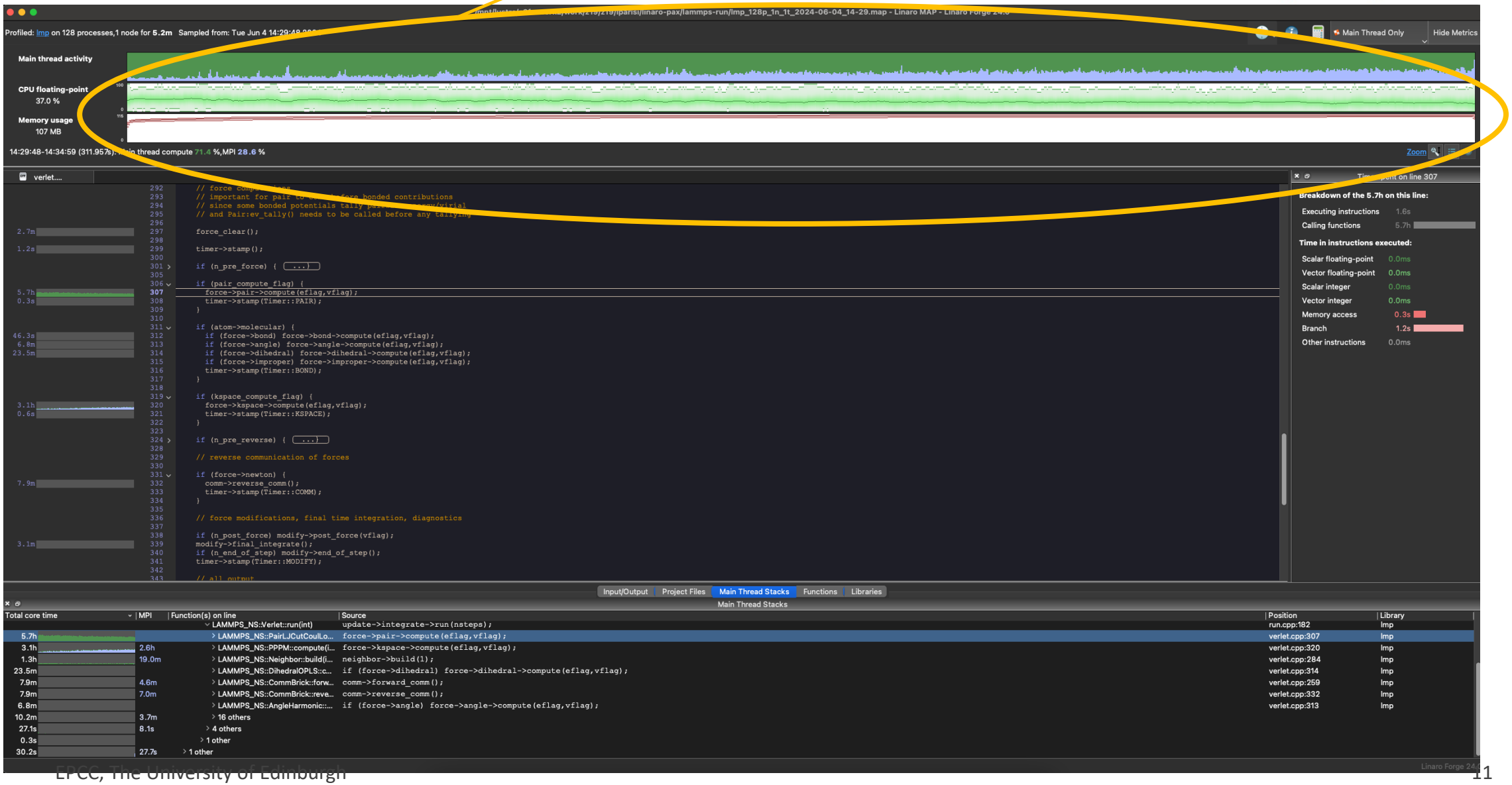
- After execution the profile is saved as a .map file
- .map files contain links to object files and source codes. If you move, or delete, source or object file you might loose information
- You can open the .map file from the forge GUI
- You can export a .map file to a .json file

```
map --export=report.json  resample_2p_2n_1t_2024-06-10_13-39.map
```

Hotspots

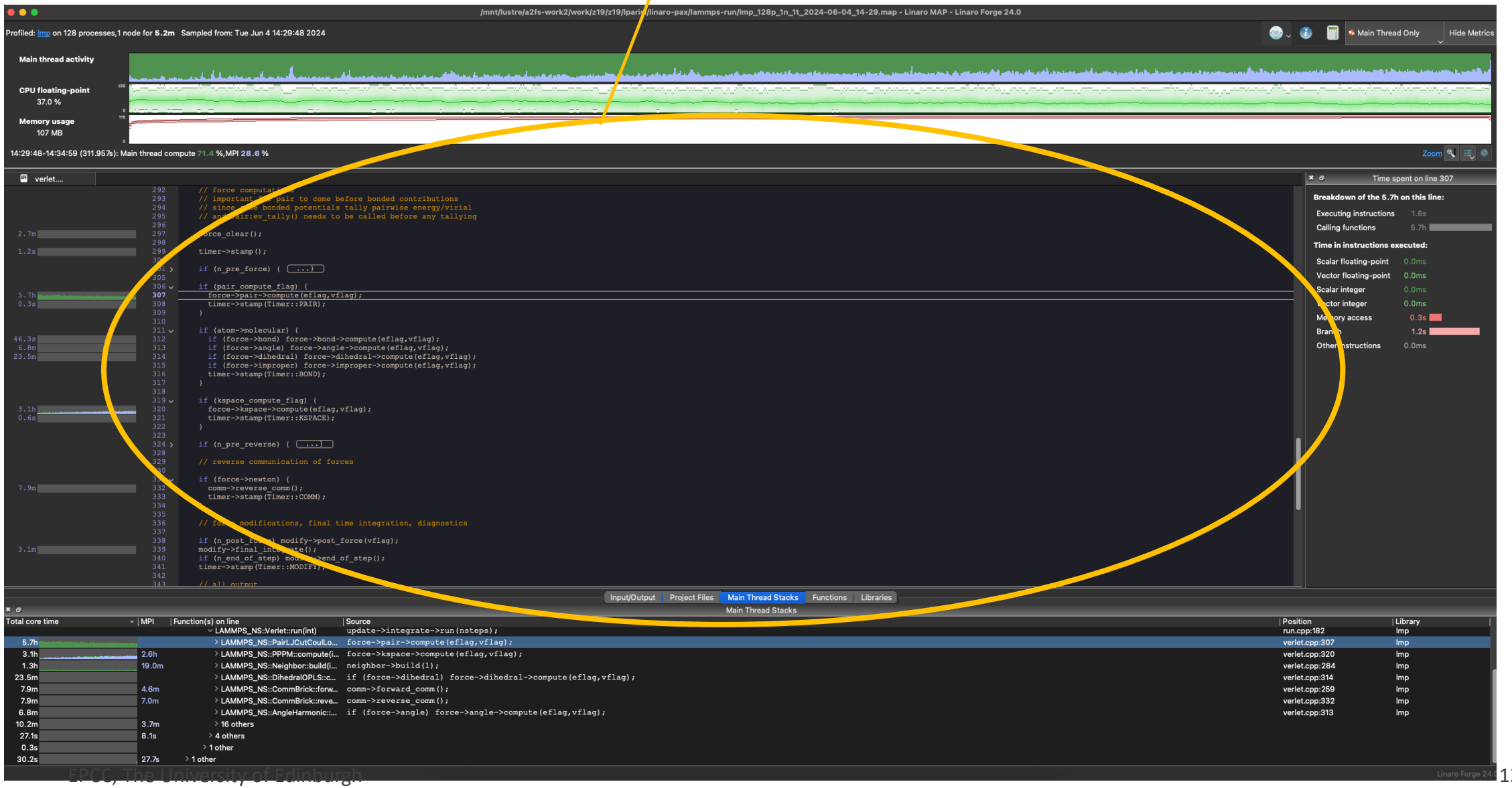
Hotspots

Metrics in function of time



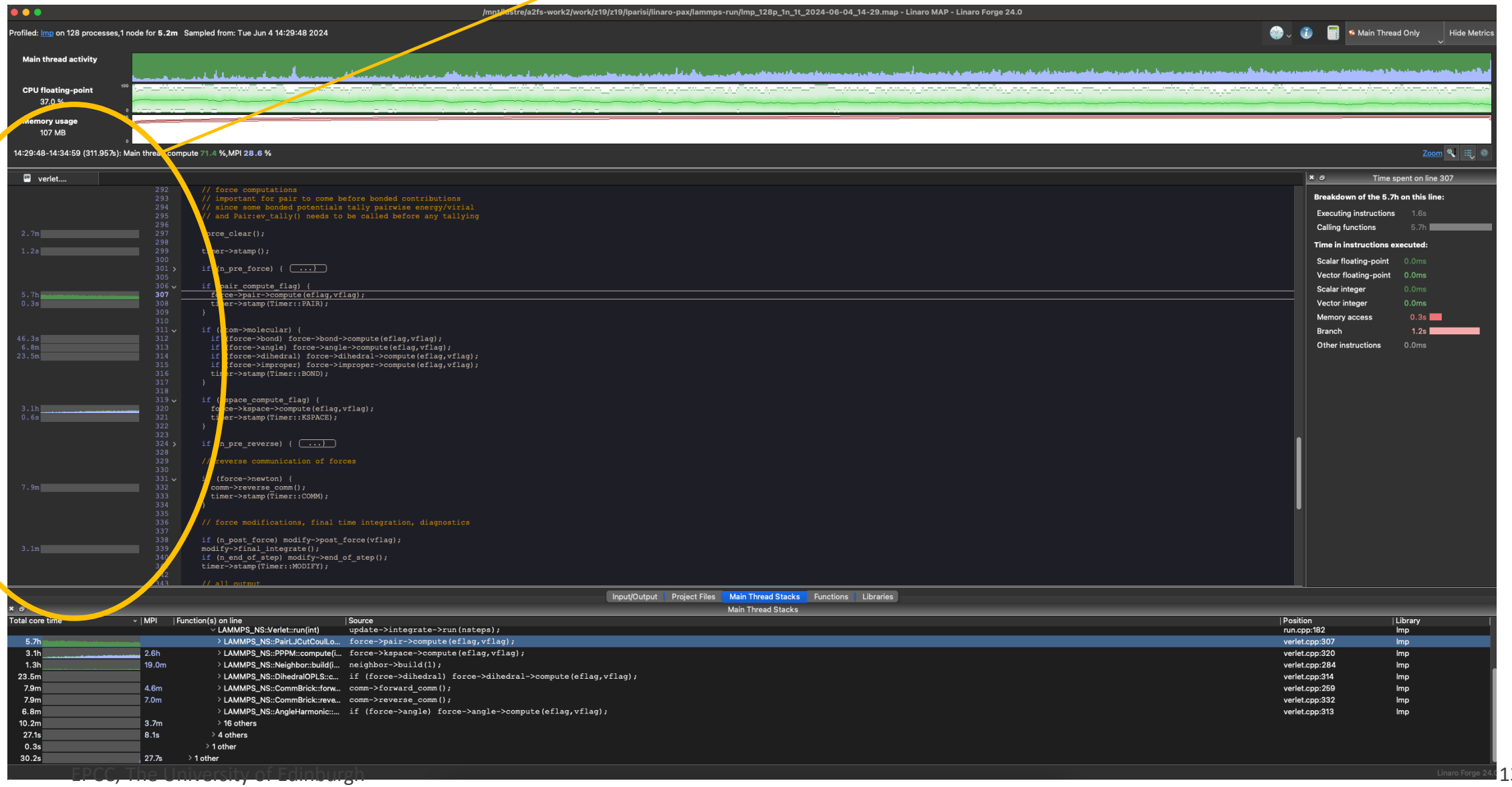
Hotspots

Source code



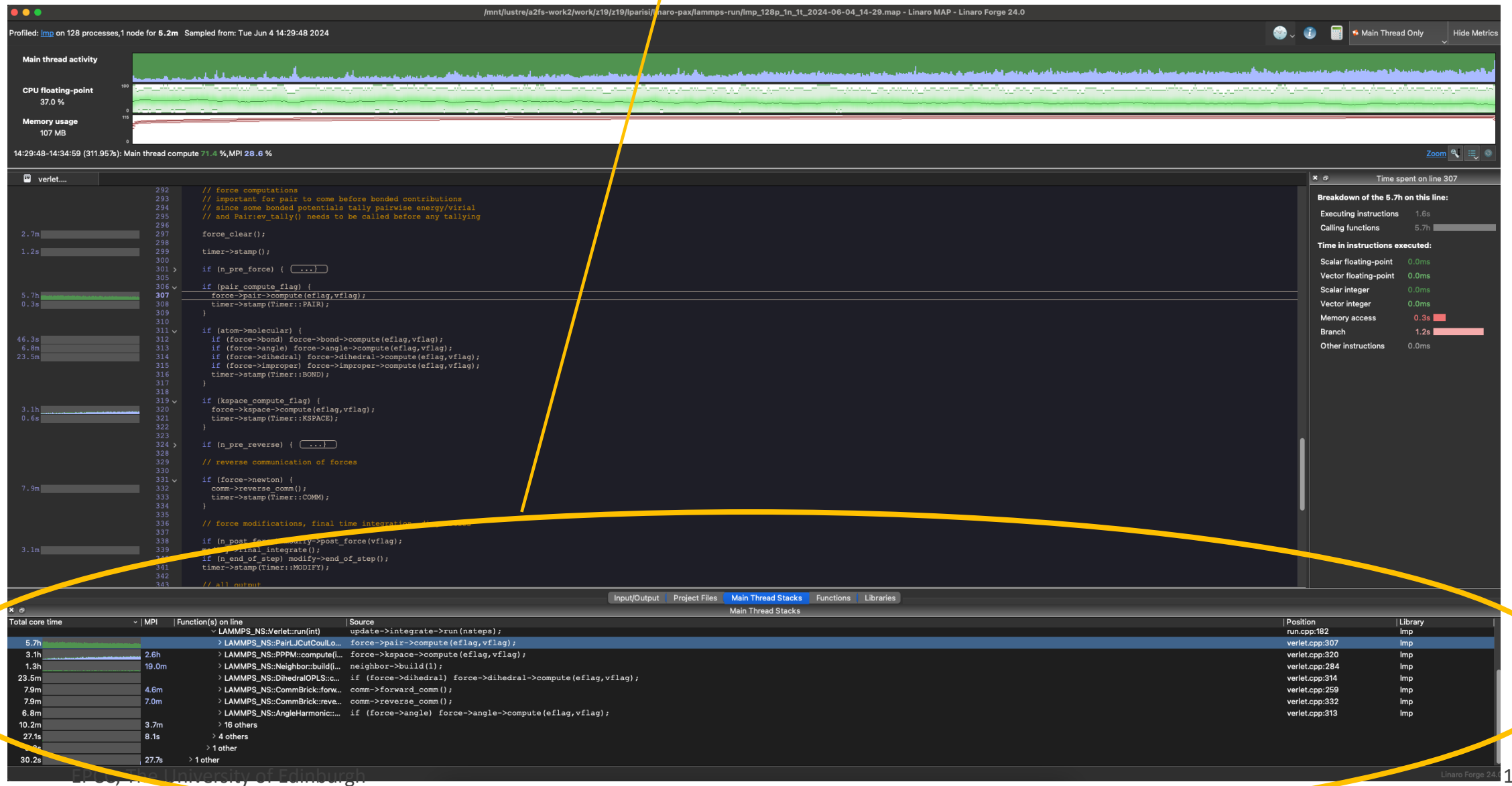
Hotspots

Sparklines next to the top time-consuming lines



Hotspots

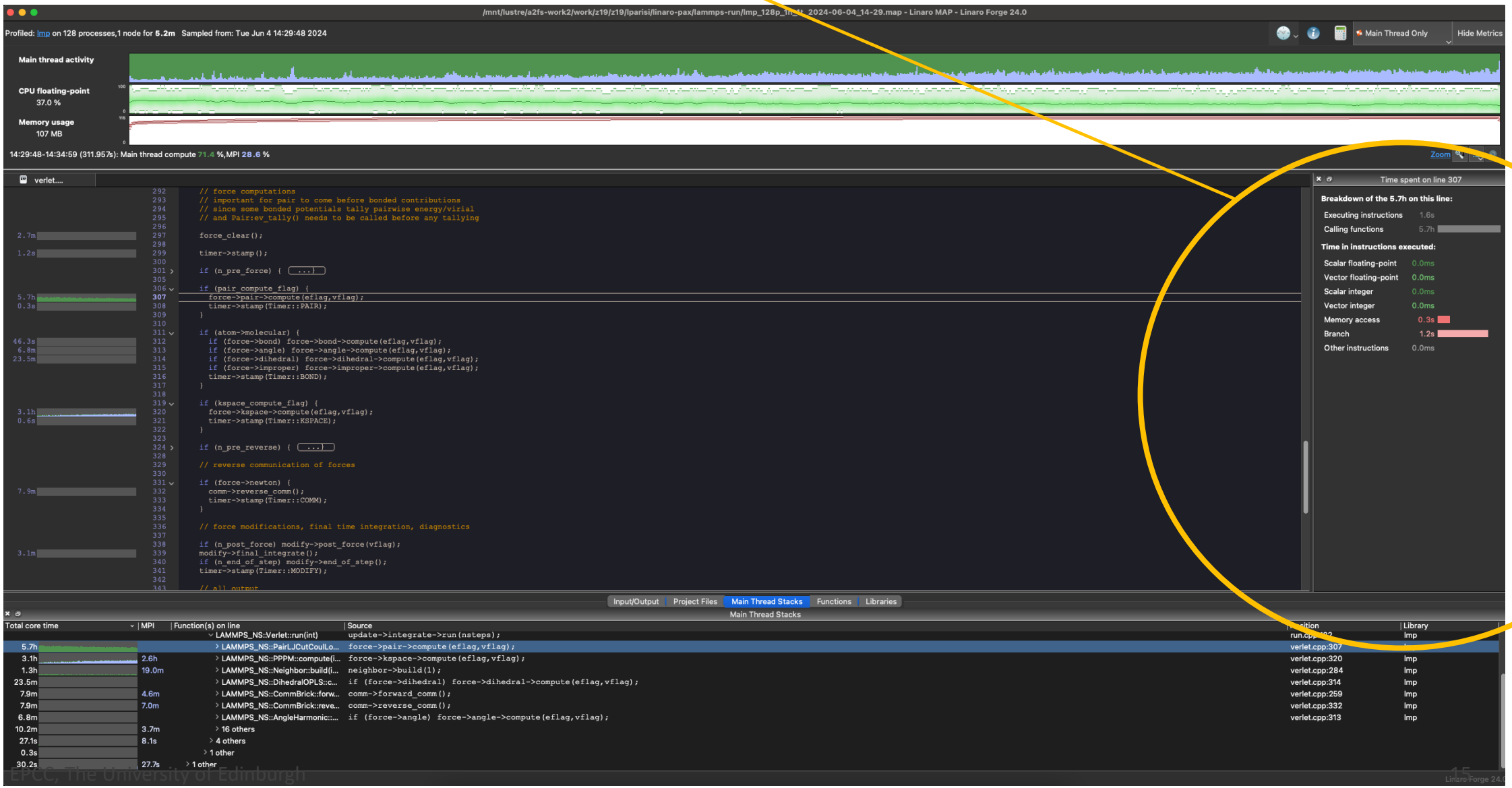
Stack of the main functions



Hotspots

Most consuming instruction types

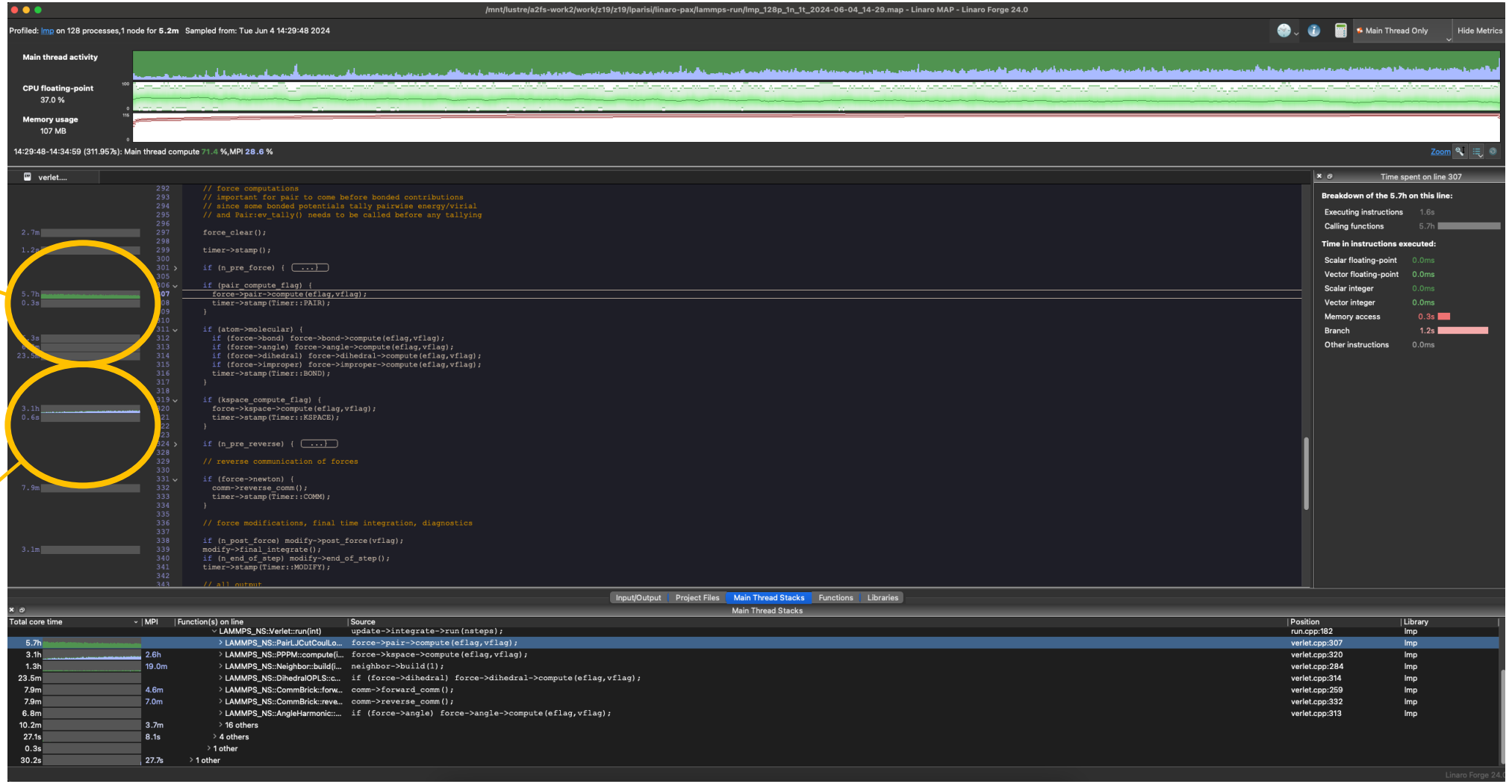
On the selected line



Hotspots

Computation time in green

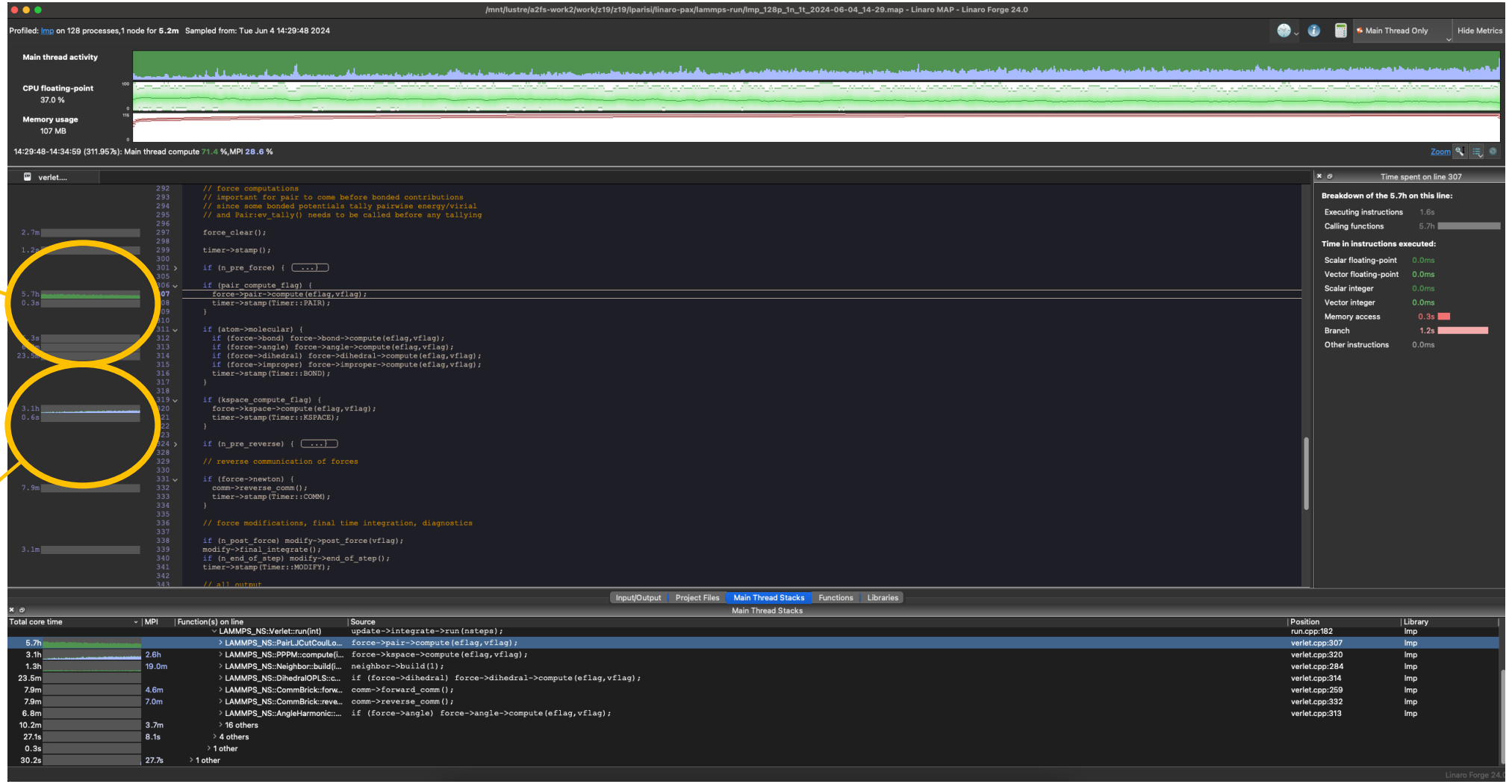
MPI time in blue



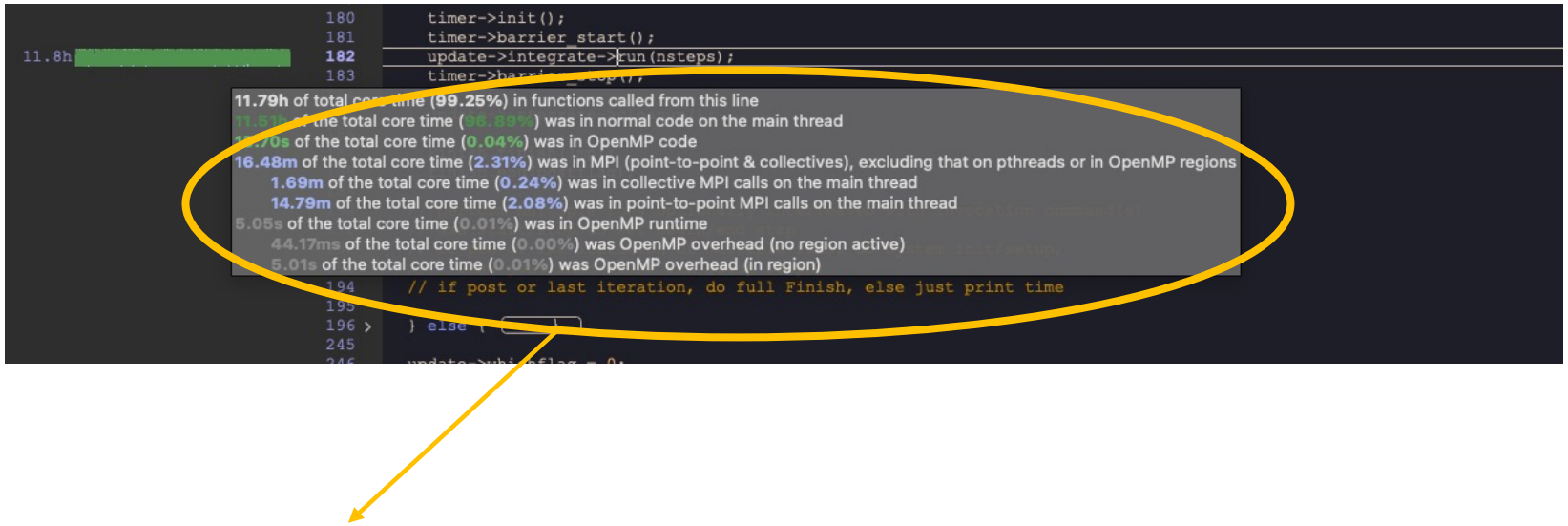
Hotspots

Computation time in green

MPI time in blue

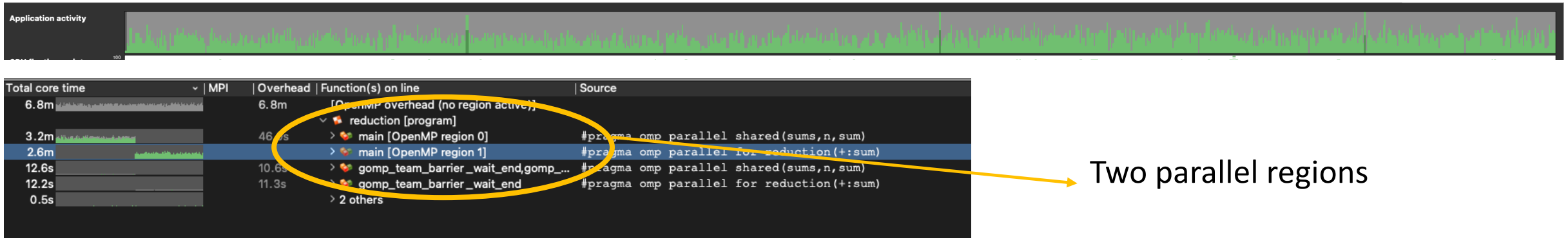


Hotspots

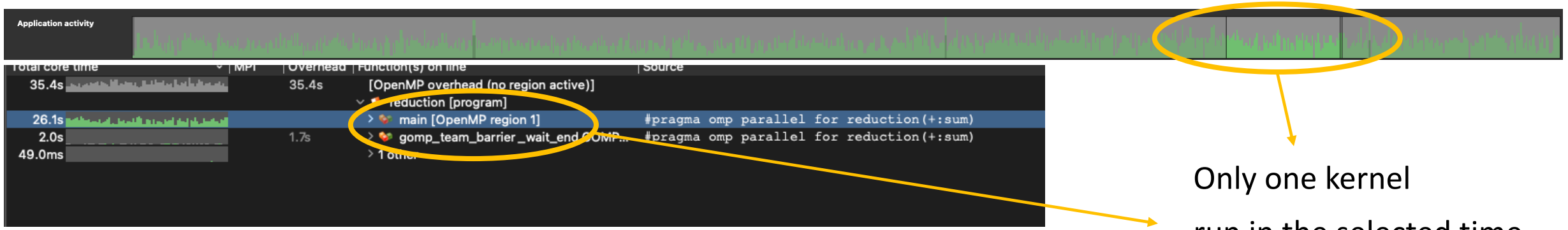


Hovering over the sparkline shows a quick breakdown of time consumption : MPI collectives, point-to-point, OpenMP overehad

Zooming in



Two parallel regions

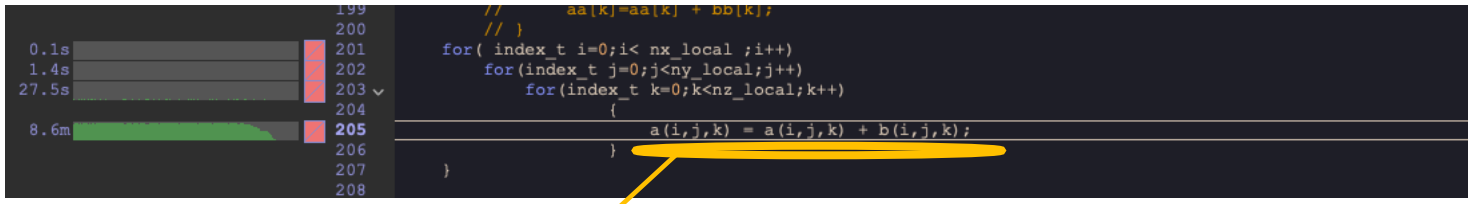


Only one kernel

run in the selected time window

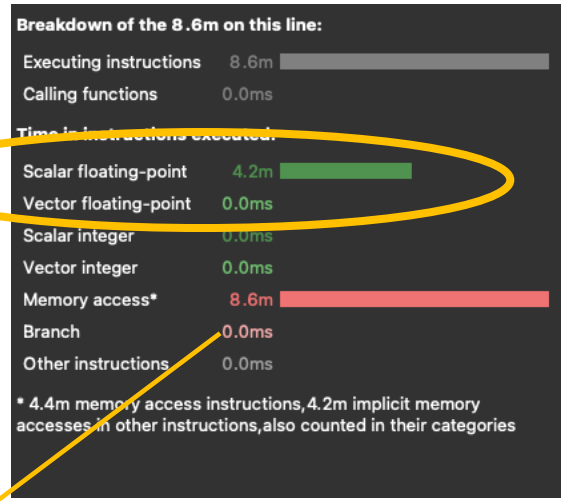
Memory management

Memory access patterns



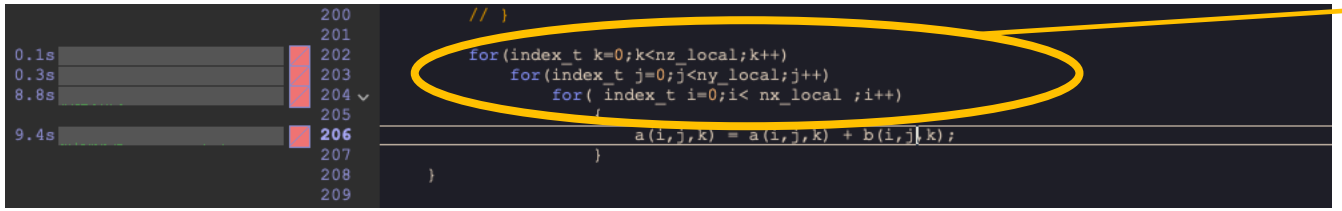
Column storage: a(i,j,k) and a(i+1,j,k) are continous

No vectorization

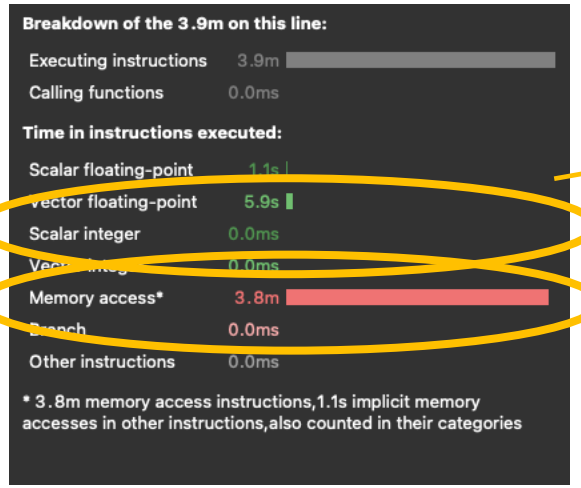


Large amount of time spent in memory accesses

Memory access patterns



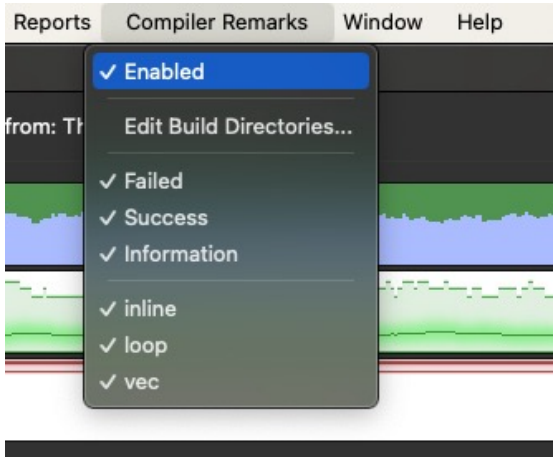
Loop order swapped



Loop gets vectorized

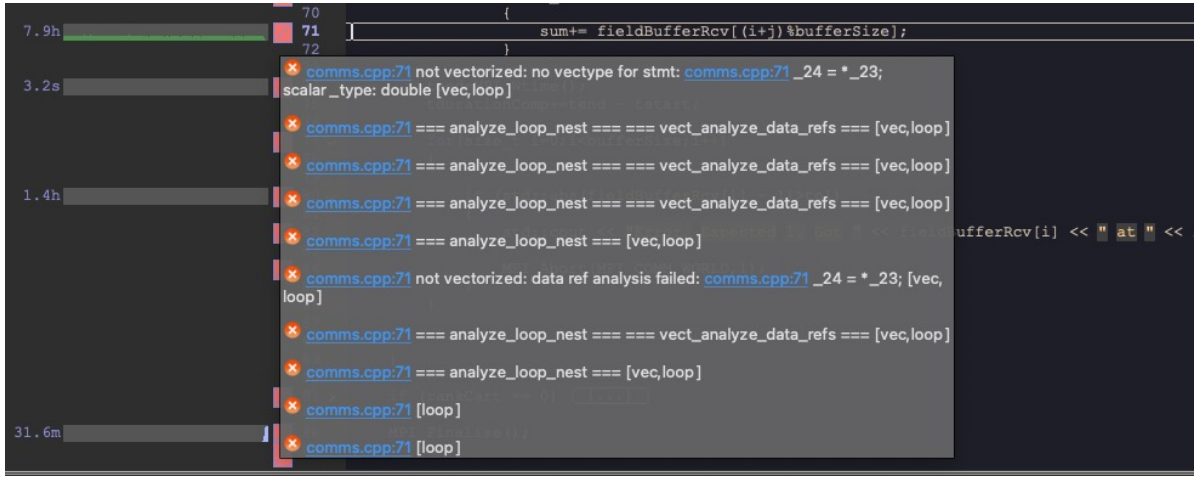
Memory access time reduce by half

Compiler remarks

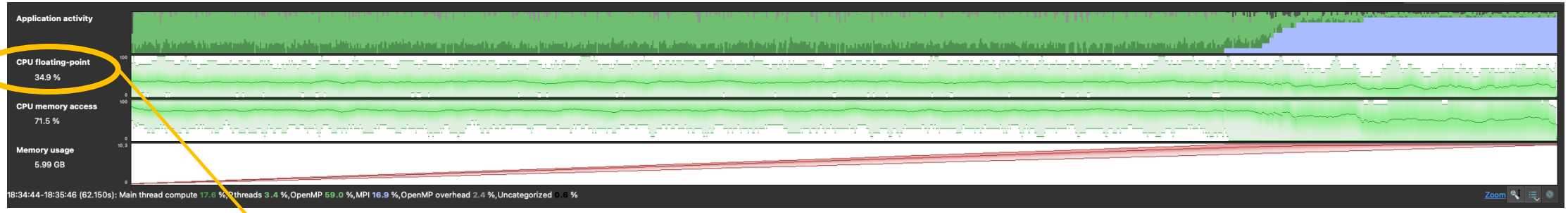


Compile with `-fsave-optimization-record` with GNU compiler

Check the Linaro Forge documentation for other compilers

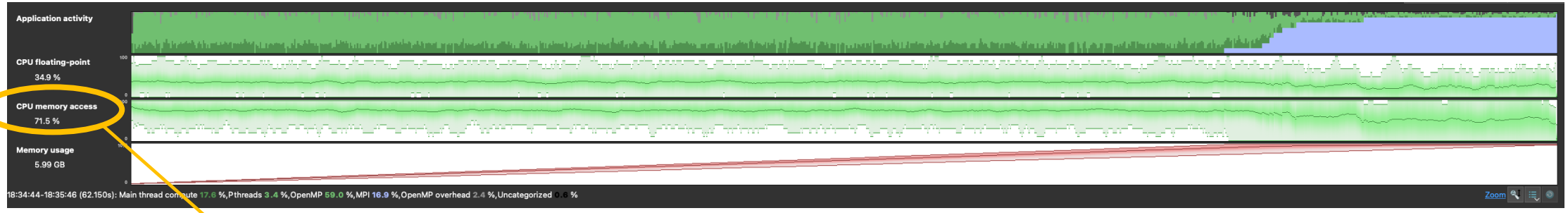


Memory usage



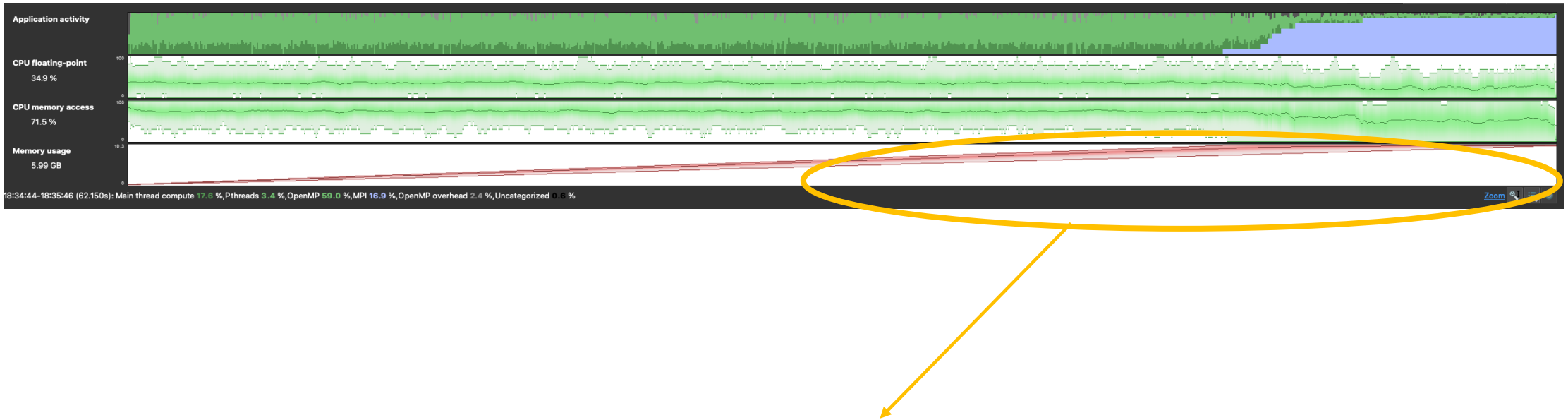
Low usage of CPU ? Bandwidth limited

Memory usage



High usage of bandwidth. Bandwidth limited ? Good access patterns ?

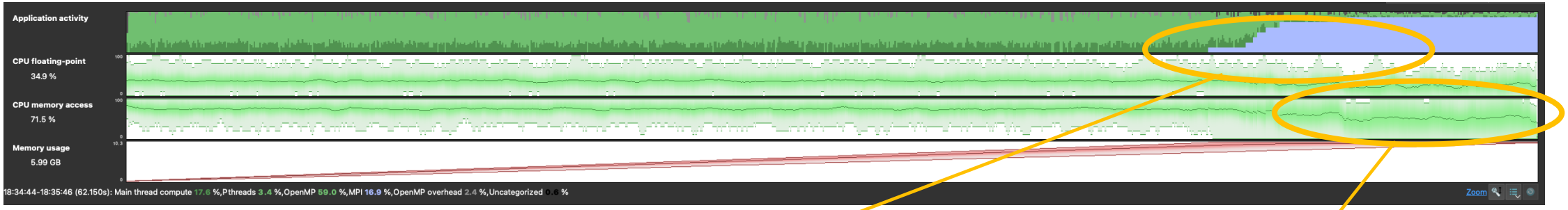
Memory leaks



Memory keeps increasing. Memory leak ?

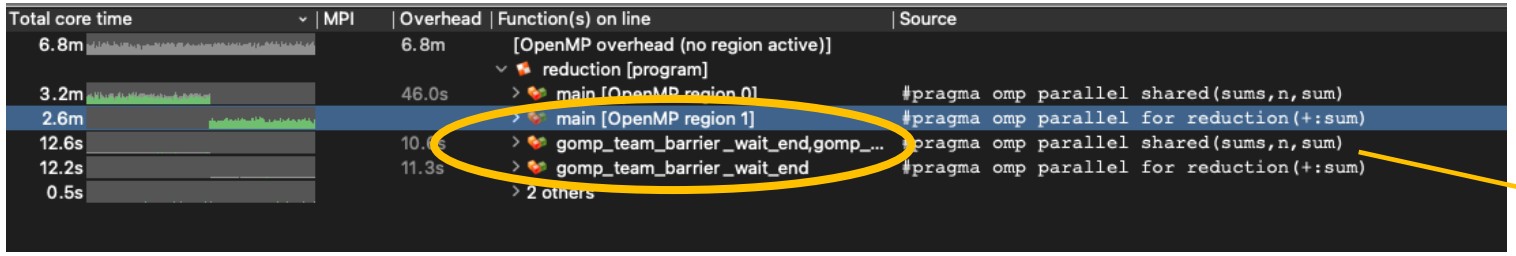
Communication

Synchronization



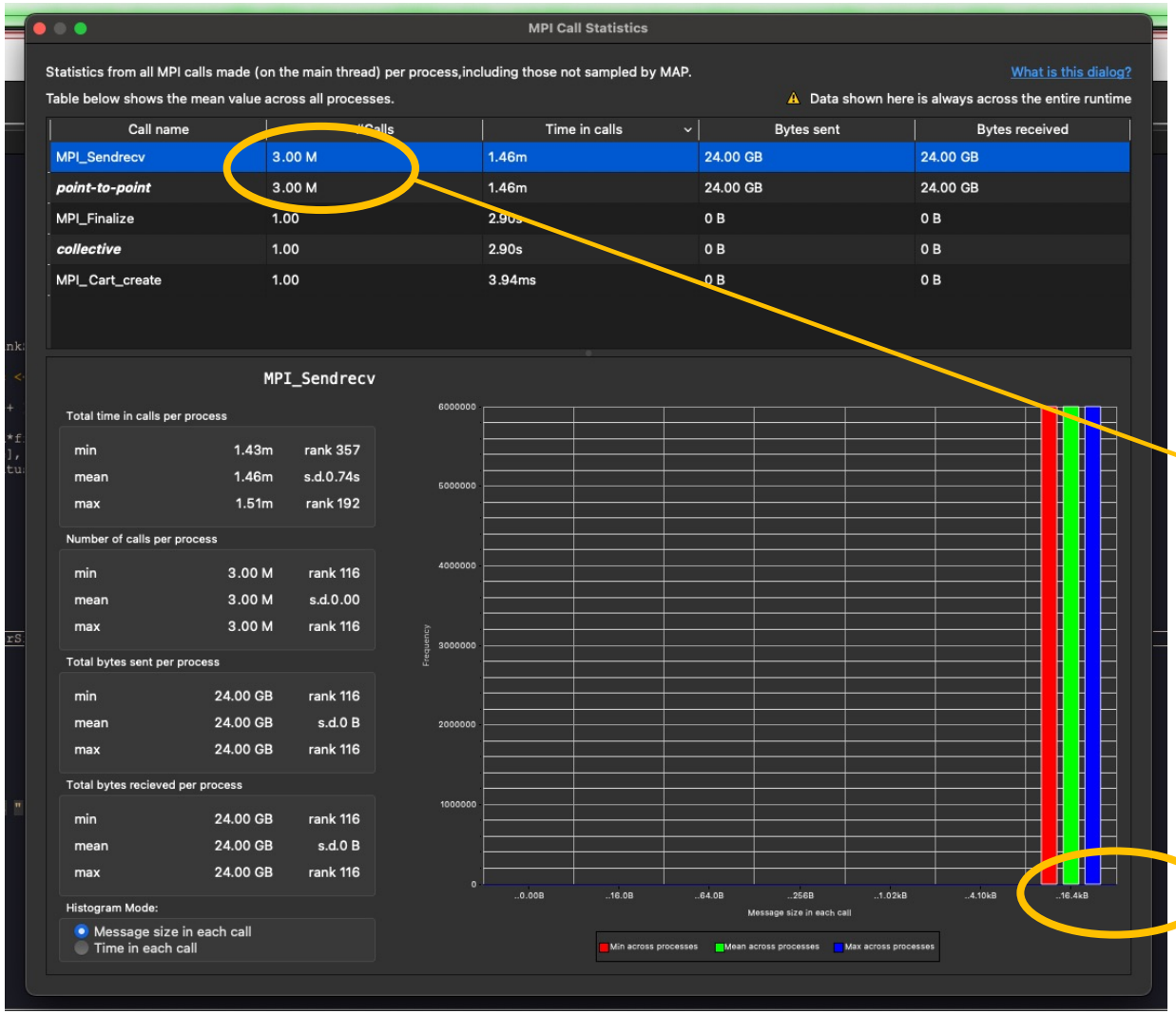
Synchronization issue ?

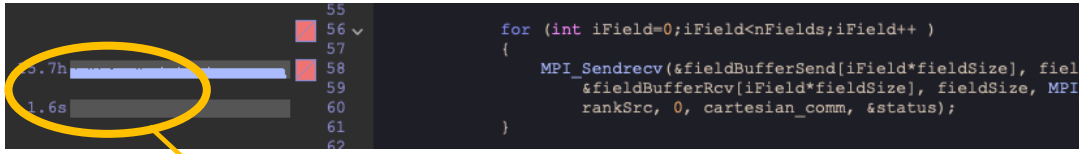
Broad distribution



Time spent in an OpenMP barrier

MPI Communication





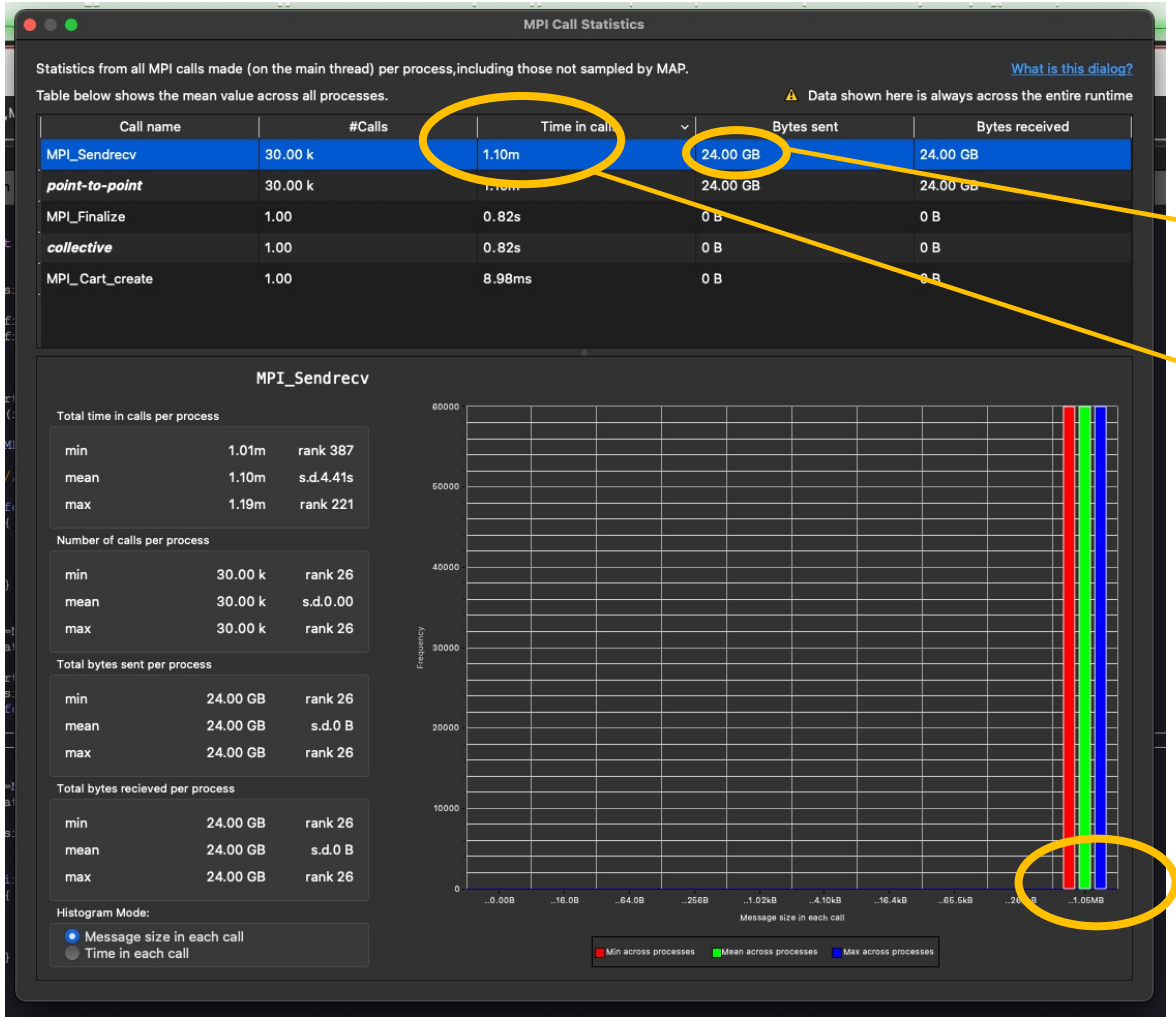
Large amount of CPU time spent in communication

Large number of MPI calls: 3M !!!

Small messages: about 16kb in size

MPI Communication

After grouping together messages…



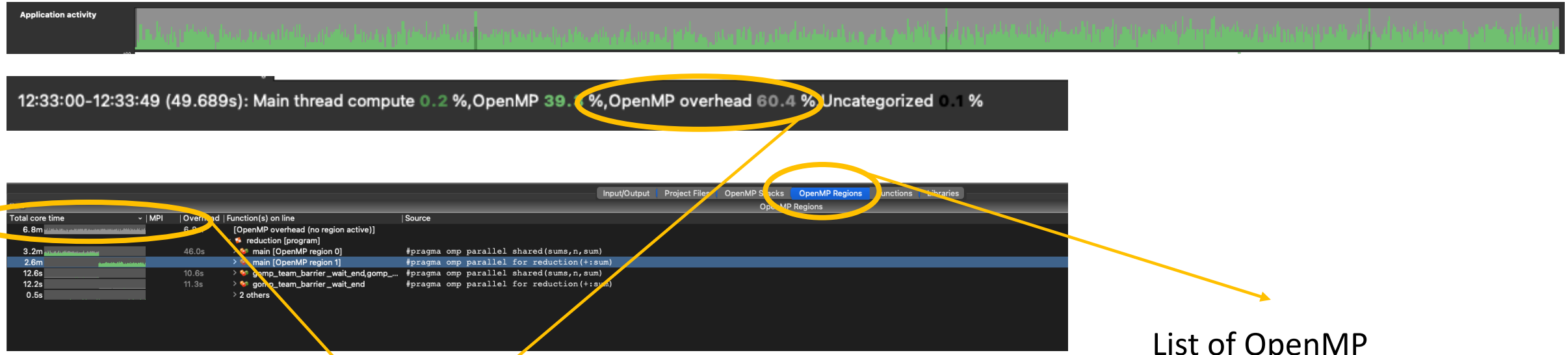
Same total amount of data, but x100 less calls.

25% reduction in MPI wall time

Grouped many small messages in a large message

Multithreading

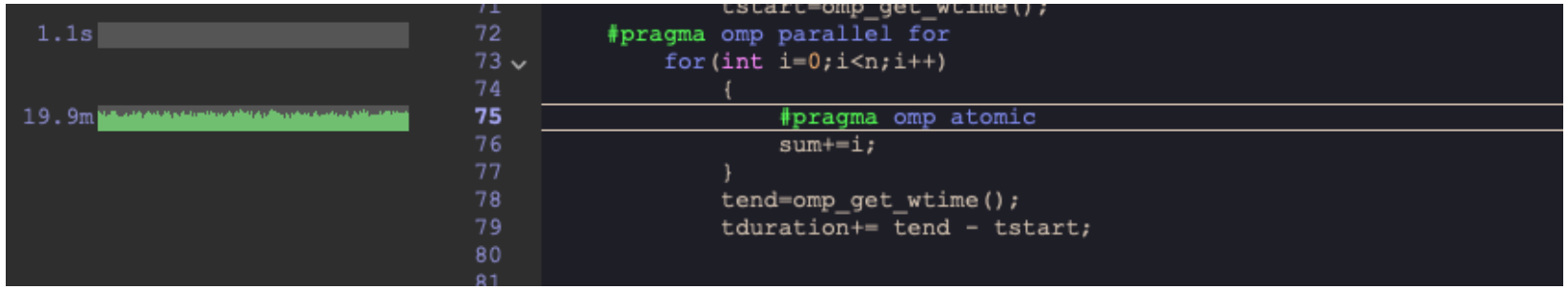
OpenMP

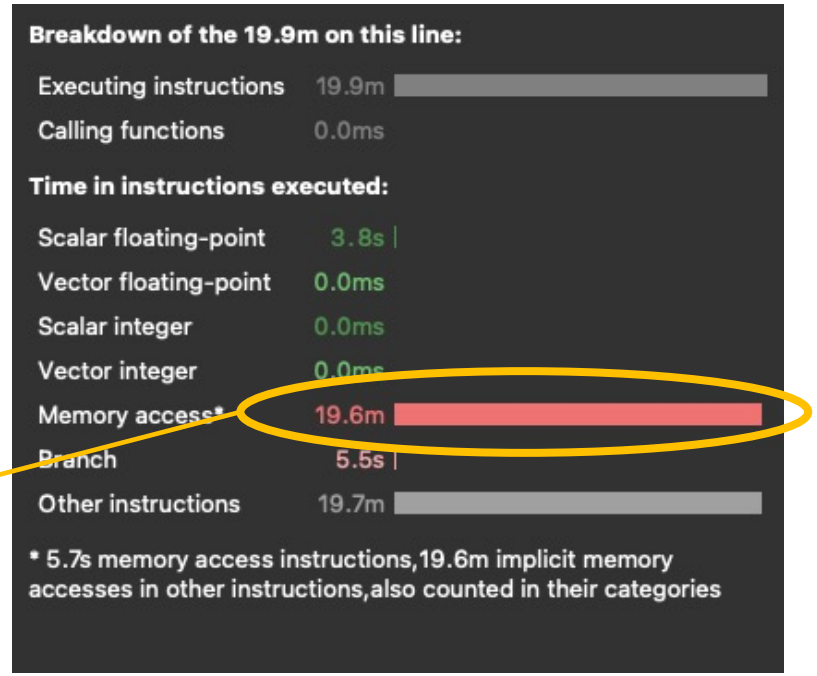


Large fraction of the time spent in initializing parallel regions

List of OpenMP regions

Atomics



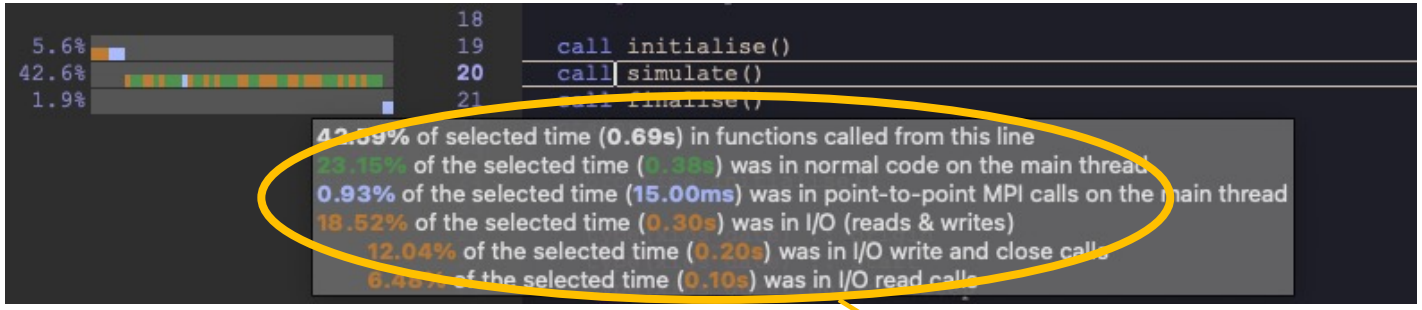


Atomic operations show up as very long memory accesses

I/O

I/O bottlenecks

One rank spends a lot of time in IO



Large fraction of time spent writing to a file

Conclusions

- User friendly interface
- Quickly spot bottlenecks in your code
- Overview of communication patterns
- Overview of memory access and usage