Multigrid solver for elliptic problem on GPU

Using dolfinx (and other things)

Igor, Chris, and Adrian

Cambridge University and Edinburgh University

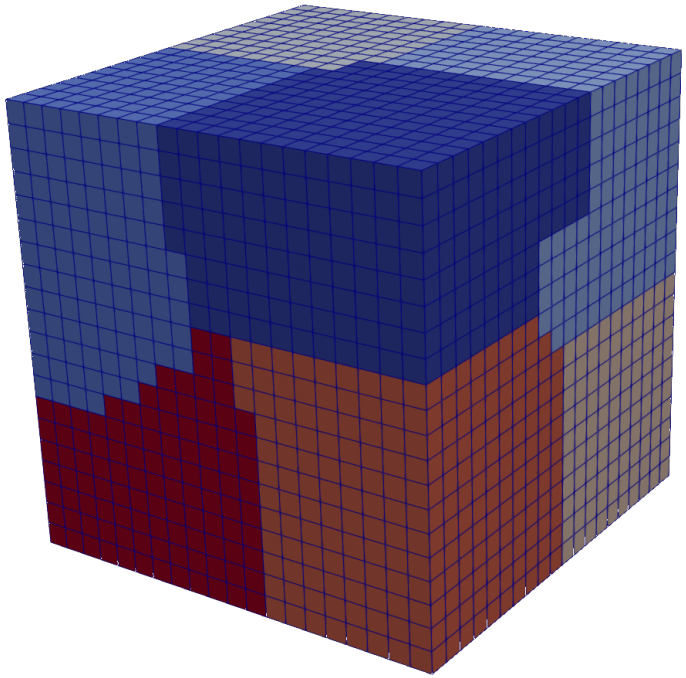
April 21, 2023

epcc

UNIVERSITY OF CAMBRIDGE

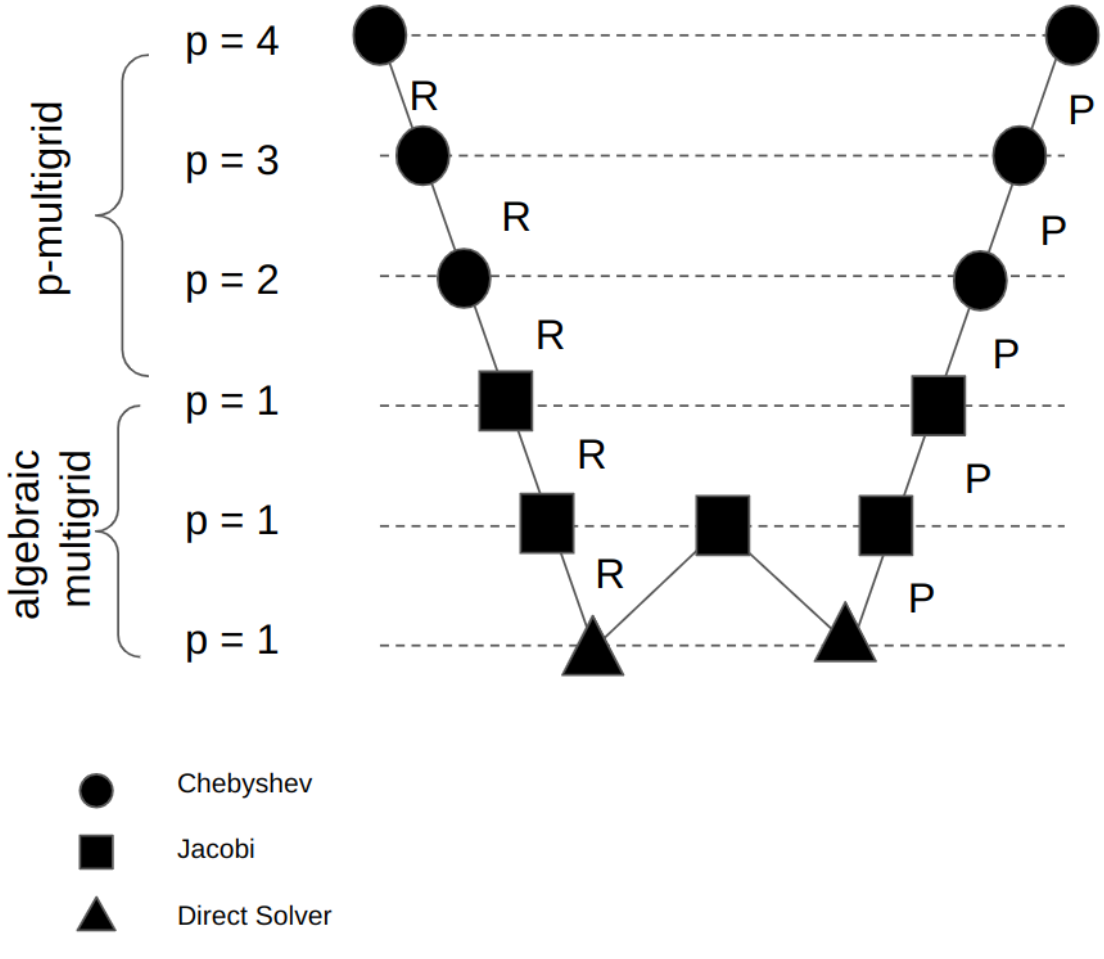
Overview

$$\int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx \tag{1}$$



P-multigrid

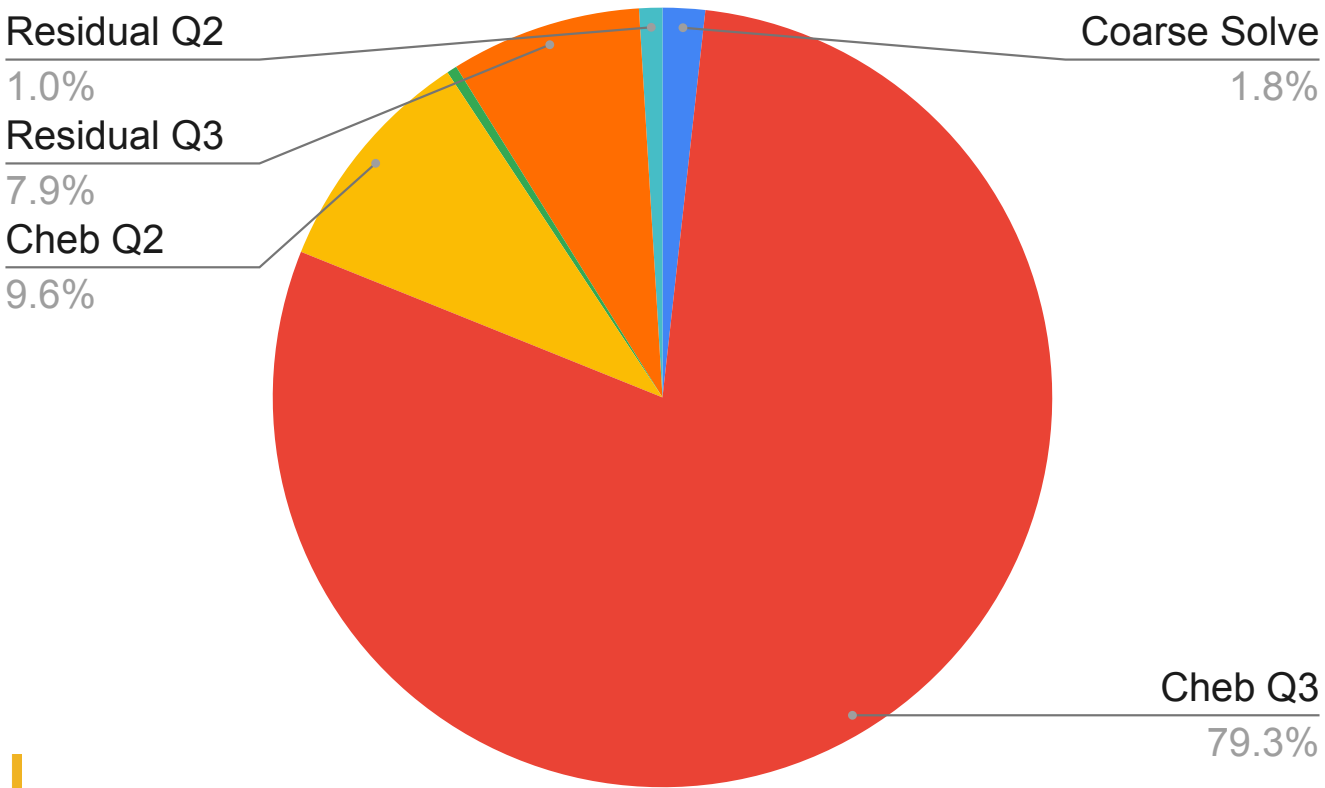
(2)



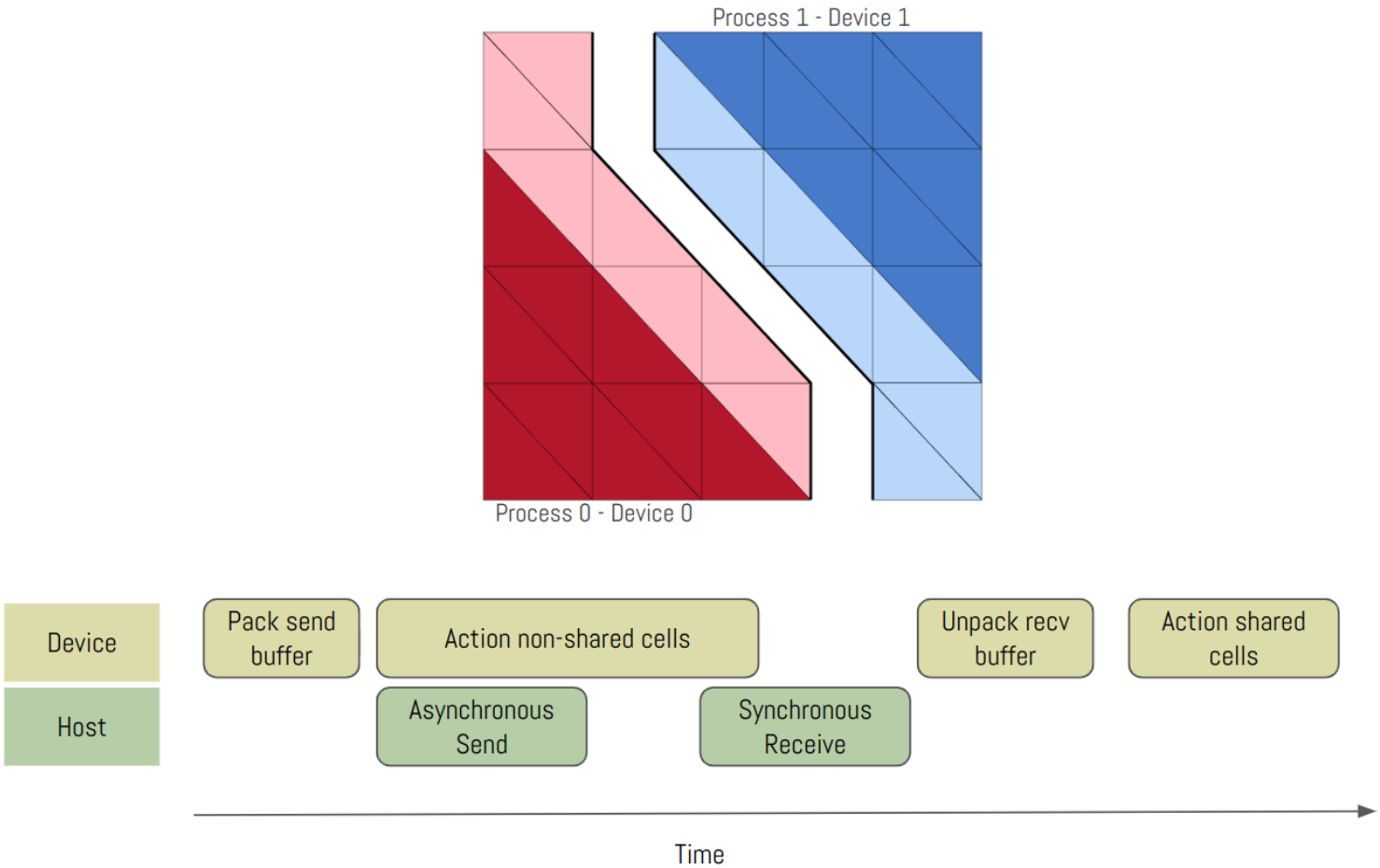
Timing 1 GPU

Hierarchy:

- Level Q1: dofs 191,748 - nnz 4,999,696 – coarse level (AMG?)
- Level Q2: dofs 1,494,425 - nnz 93,773,201
- Level Q3: dofs 4,999,696 - nnz 61,623,0976



Overlapping communication and computation



Distributed MatVec I

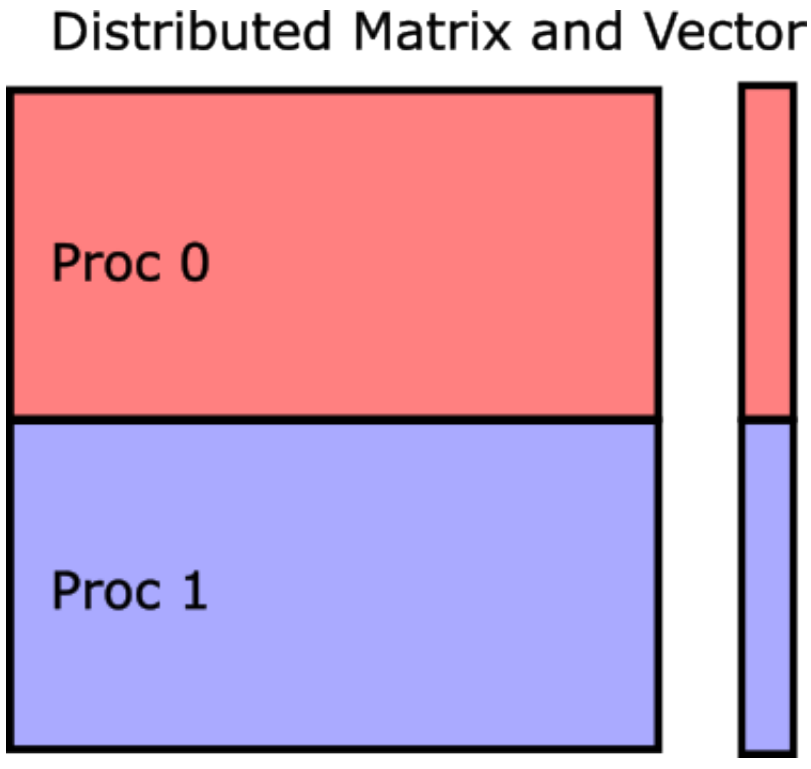
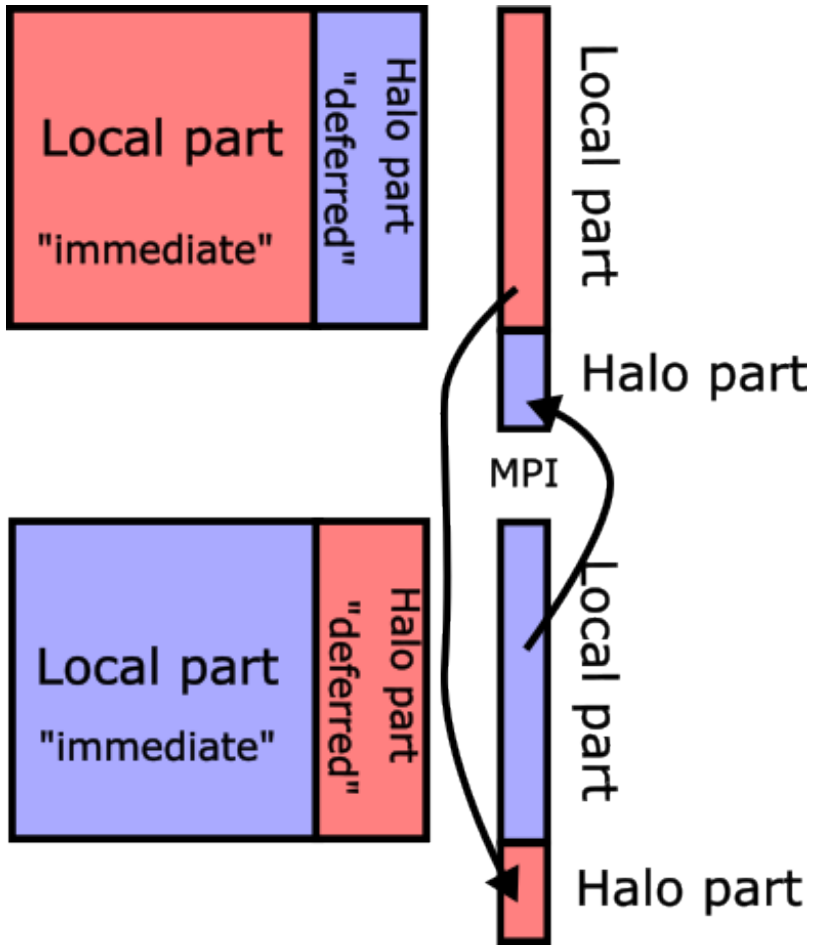
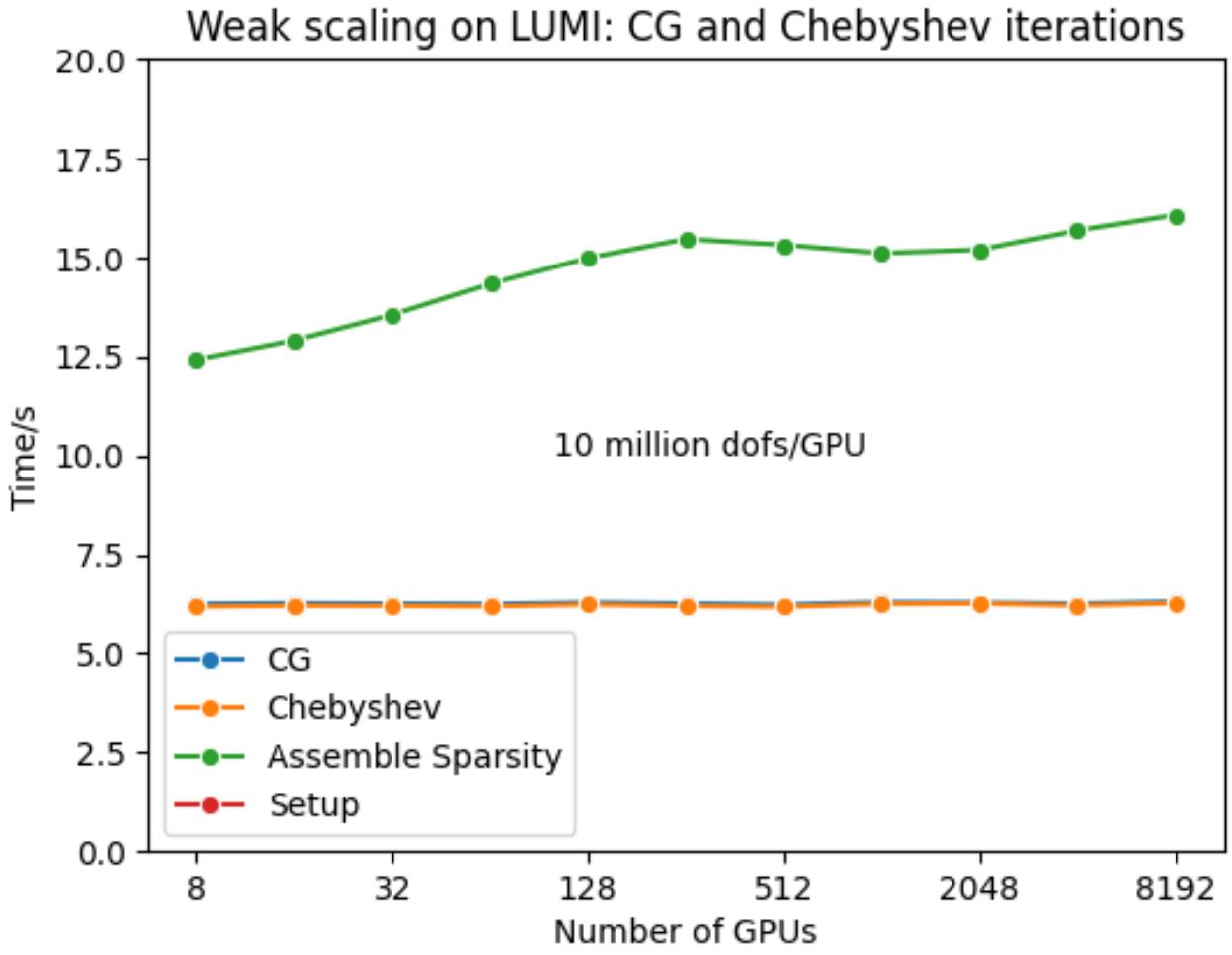


Figure: Matrix-vector Product (two processes)

Distributed MatVec II



Weak scaling



Profiling

- Used both `rocprof` and `omnitrace`
- `rocprof` using for getting started quickly
  - Easy to run on a single GPU
  - Reasonable first results to show
  - Useful programming API to allow marking up sections of code
  - Wrapper scripted needed for more than one GPU
- `rocprof` API allowed for adding mark up to the profiles

```
roctxRangePush{("This section");
roctxRangePop();
ret = rsmi_init(0);
ret = rsmi_shut_down();
```

rocprof wrapper script

```
#!/bin/bash
set -euo pipefail
name="$1"
if [[ -n ${OMPI_COMM_WORLD_RANK+z} ]]; then
    export MPI_RANK=${OMPI_COMM_WORLD_RANK}
elif [[ -n ${MV2_COMM_WORLD_RANK+z} ]]; then
  export MPI_RANK=\${MV2_COMM_WORLD_RANK}
elif [[ -n ${SLURM_PROCID+z} ]]; then
  export MPI_RANK=${SLURM_PROCID}
else
  echo "Unknown MPI layer detected! Must use OpenMPI, MVAPICH, or Slurm"
  exit 1
fi
export ROCR_VISIBLE_DEVICES=$SLURM_LOCALID
rocprof="/opt/rocm/bin/rocprof"
pid="$$"
outdir="rank_${pid}_${MPI_RANK}"
outfile="${name}_${pid}_${MPI_RANK}.csv"
${rocprof} -d ${outdir} -o ${outdir}/${outfile} --hsa-trace
 --hip-trace --roctx-trace "${@:2}"
```

Wrapper scripted called like this:

```
srun -N ${SLURM_NNODES} -n ${SLURM_NTASKS} ${cpu_bind} ./rocprof_wrapper.sh app_name app
```

Note the moving of the `ROCM_VISIBLE_DEVICES` into the wrapper script.

Getting GPU usage information from the application

- Wanted to be able to get memory usage (and other things) inside the application
- Enabled tracking where memory allocated, not just how much
- `rocm-smi` can do this from the command line
- Also has an API

rocm-smi API calls

```
rsmi_status_t err;
uint64_t total, usage;
float return_value = 0.0;
uint32_t num_devices;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
num_devices = num_monitored_devices();
if(num_devices == 1 || rank == 0){
  for(uint32_t i = 0; i < num_devices; ++i) {
    std::cout << text << " MPI Rank " << rank << " GPU " << i << "  % memory used:";
    for (uint32_t mem_type = RSMI_MEM_TYPE_FIRST; mem_type <= RSMI_MEM_TYPE_LAST; ++mem_type) {
      err = rsmi_dev_memory_total_get(i, static_cast<rsmi_memory_type_t>(mem_type), &total);
      if (err != RSMI_STATUS_SUCCESS) {
        return return_value;
      }
      err = rsmi_dev_memory_usage_get(i, static_cast<rsmi_memory_type_t>(mem_type), &usage);
      if (err != RSMI_STATUS_SUCCESS) {
        return return_value;
      }
      if(mem_type == RSMI_MEM_TYPE_VRAM){
        return_value = (static_cast<float>(usage)*100)/total;
      }
      std::cout << " " << kDevMemoryTypeNameMap.at(static_cast<rsmi_memory_type_t>(mem_type));
      std::cout << " " << std::setprecision(2) << (static_cast<float>(usage)*100)/total;
    }
    std::cout << "\n";
  }
}
```

Whole application profile

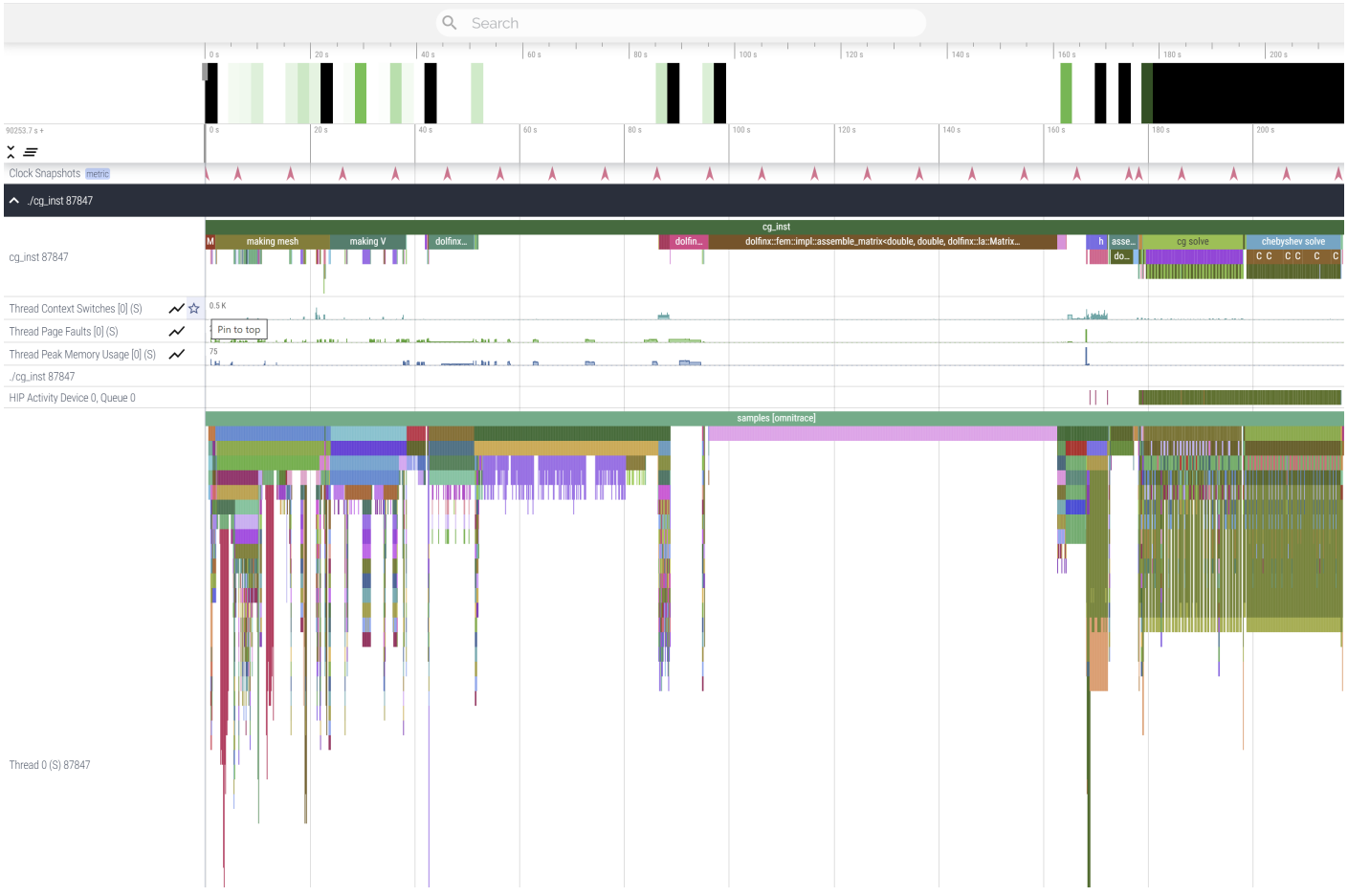


Figure: `omnitrace` profile of the whole application, 2 nodes, 8 GPU per node, 30m DoF per GPU. Profile from MPI rank 0

Main solving phase

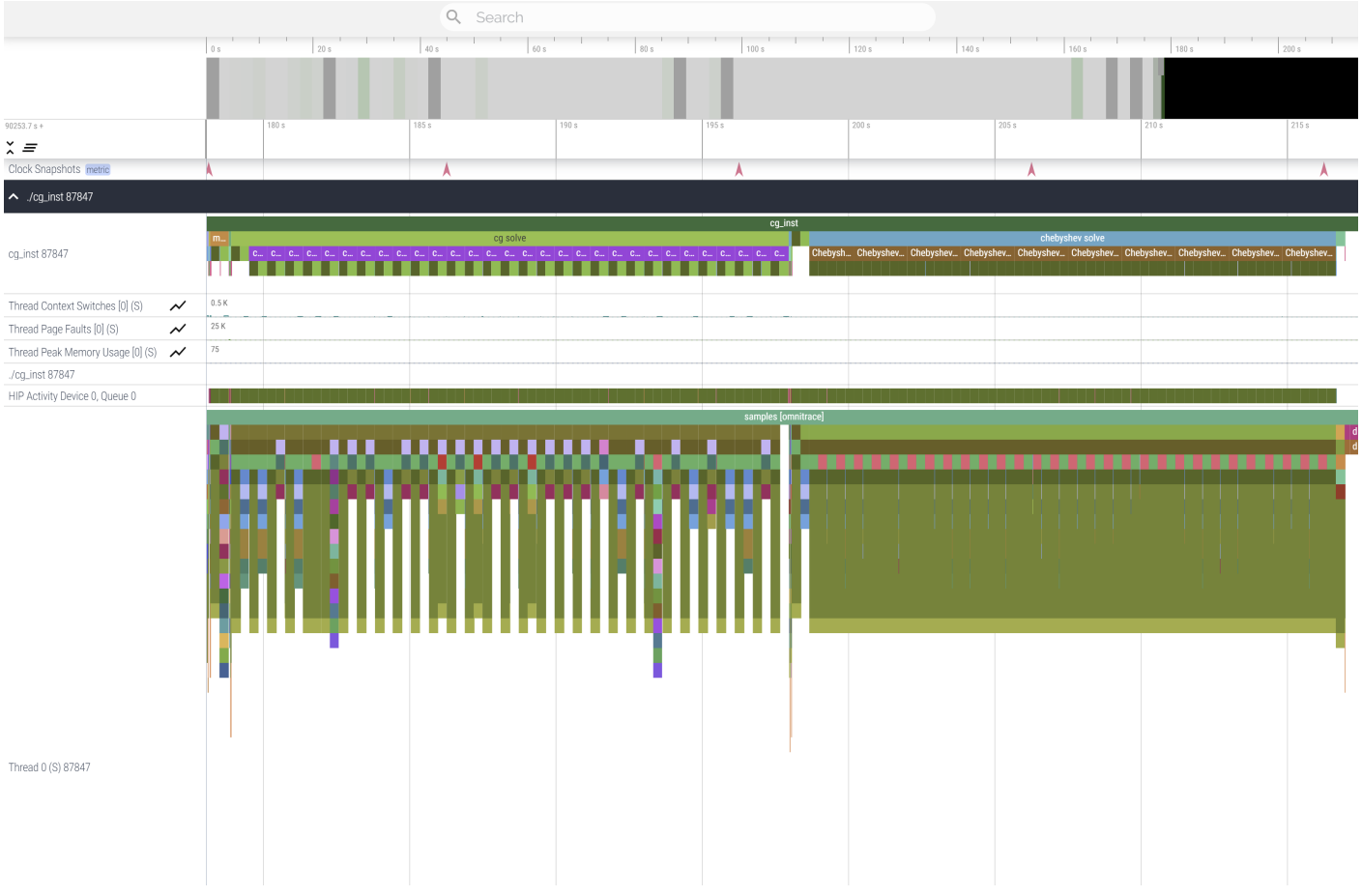


Figure: `omnitrace` profile of the main solving phase of the application, 2 nodes, 8 GPU per node, 30m DoF per GPU. Profile from MPI rank 0

CG phase

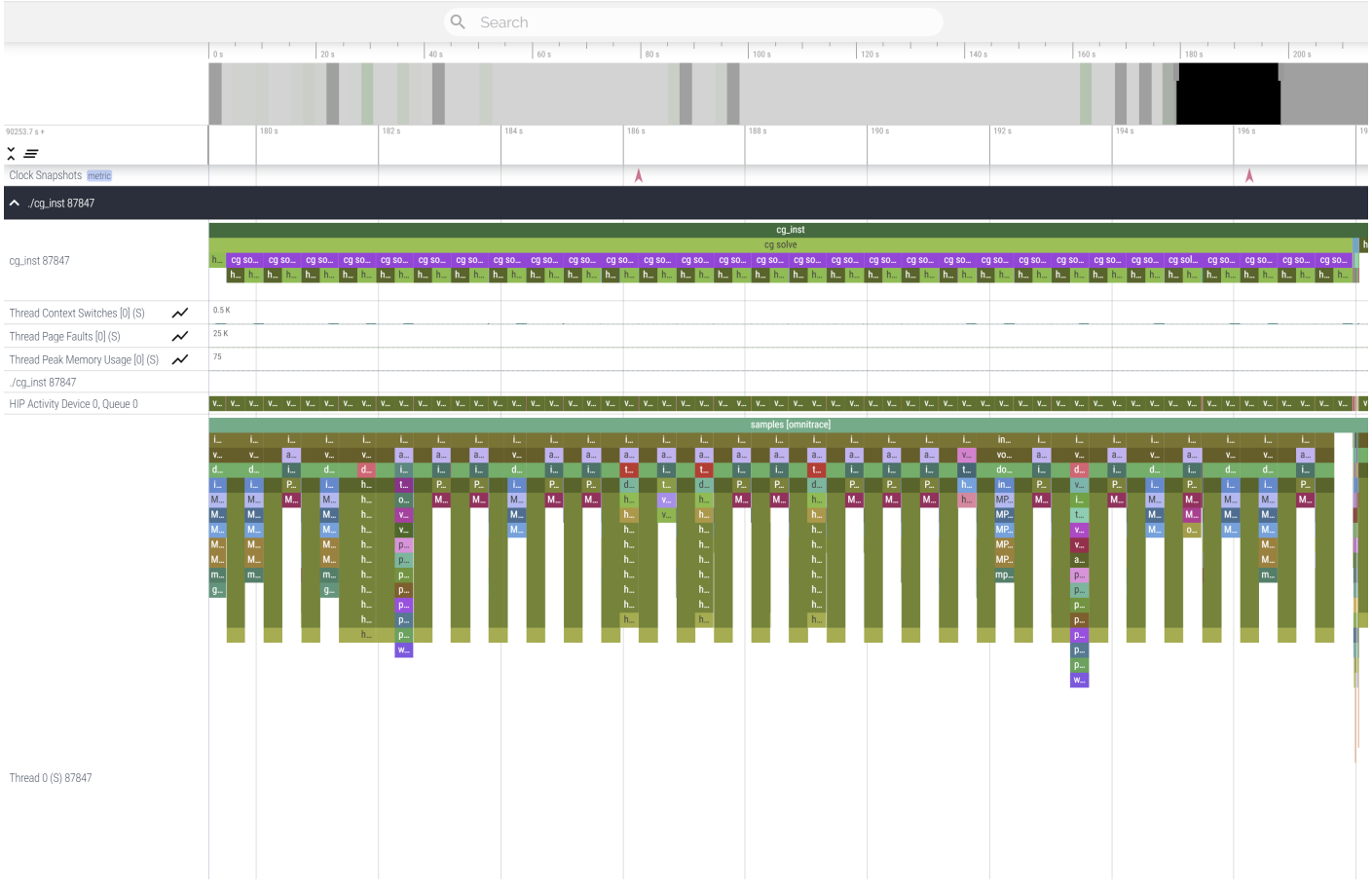


Figure: `omnitrace` profile of the CG solving phase

Chebyshev phase

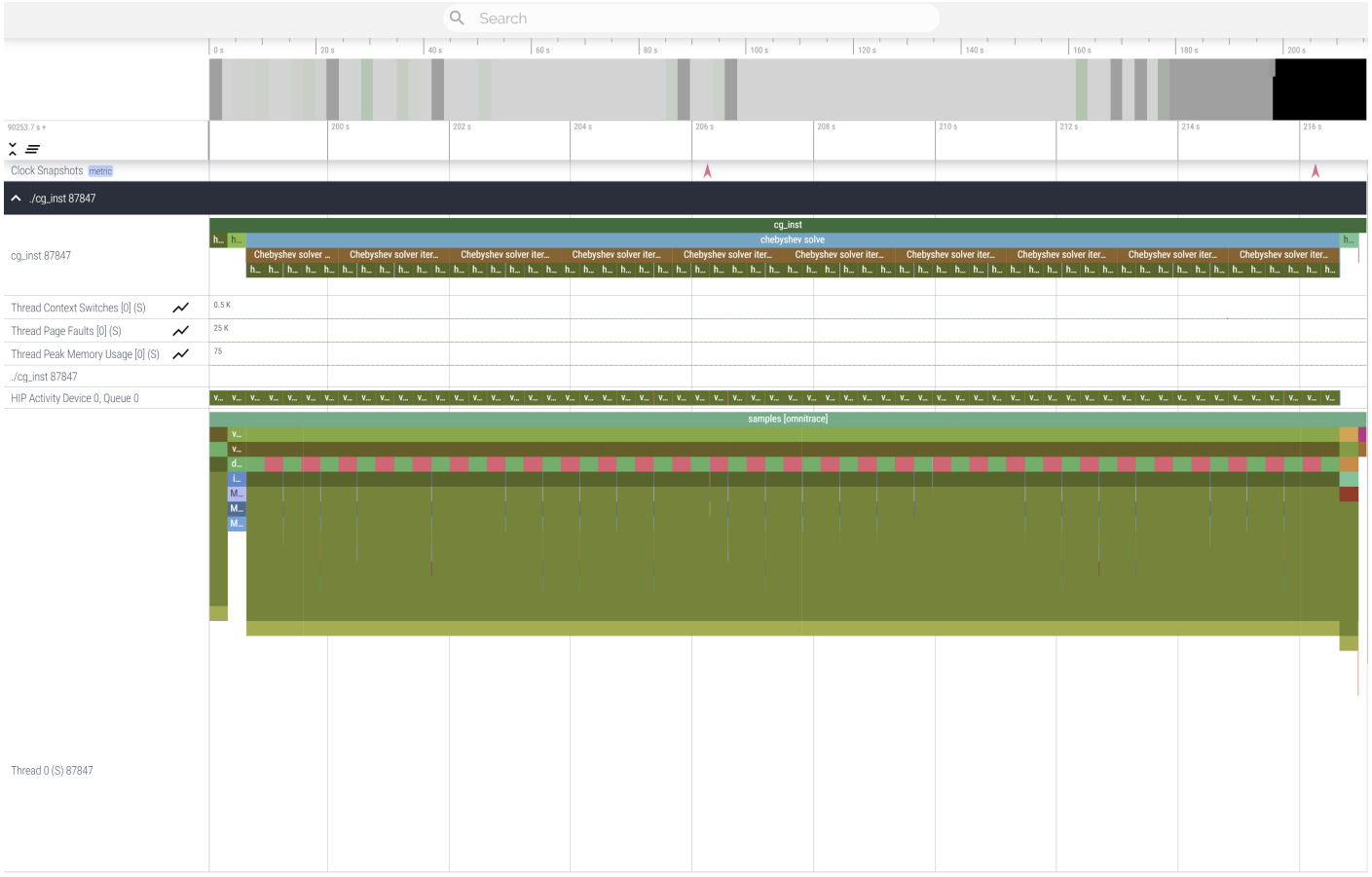


Figure: `omnitrace` profile of the Chebyshev solving phase

CG communications

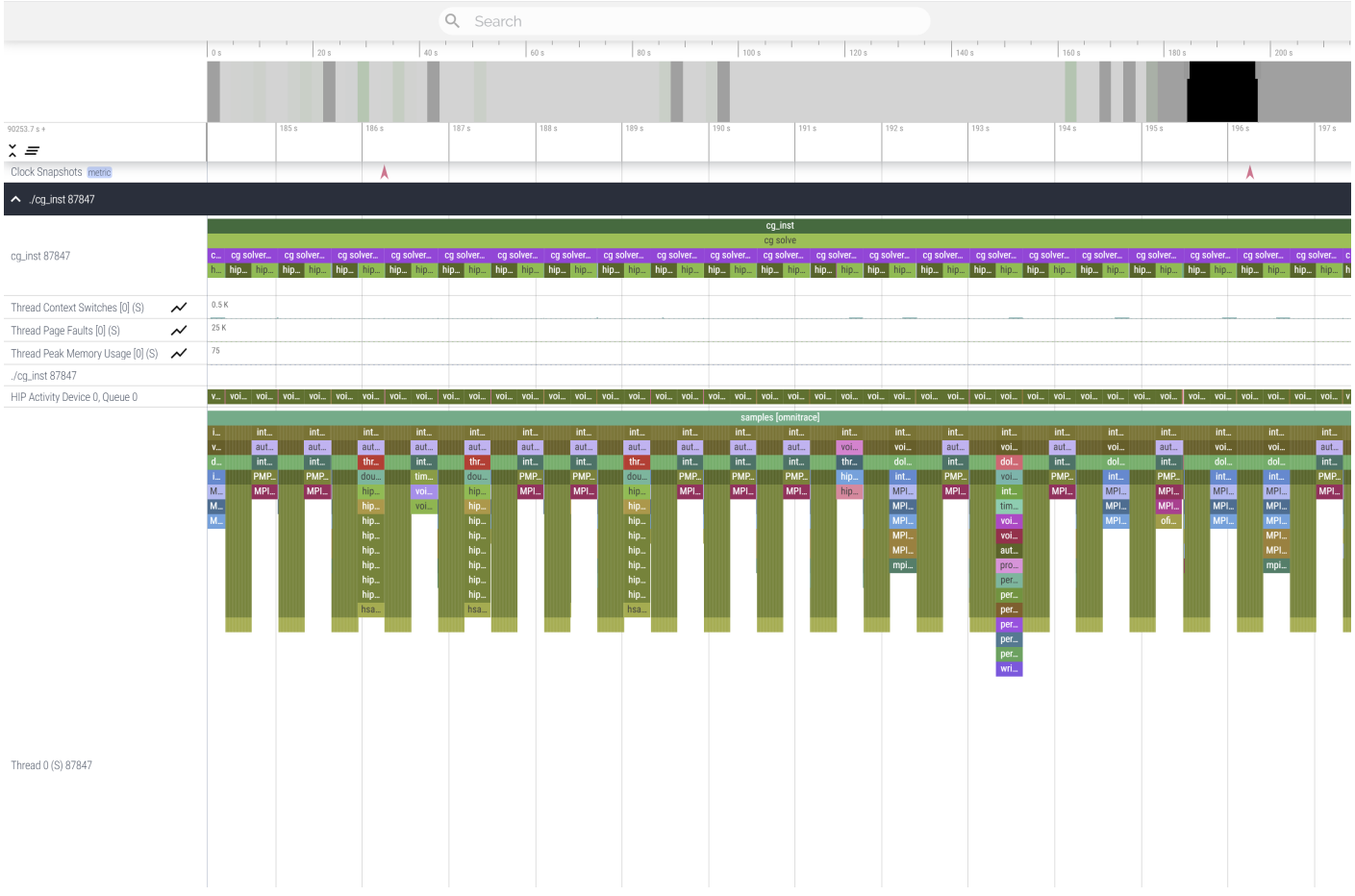


Figure: `omnitrace` profile of a CG communications

CG communications

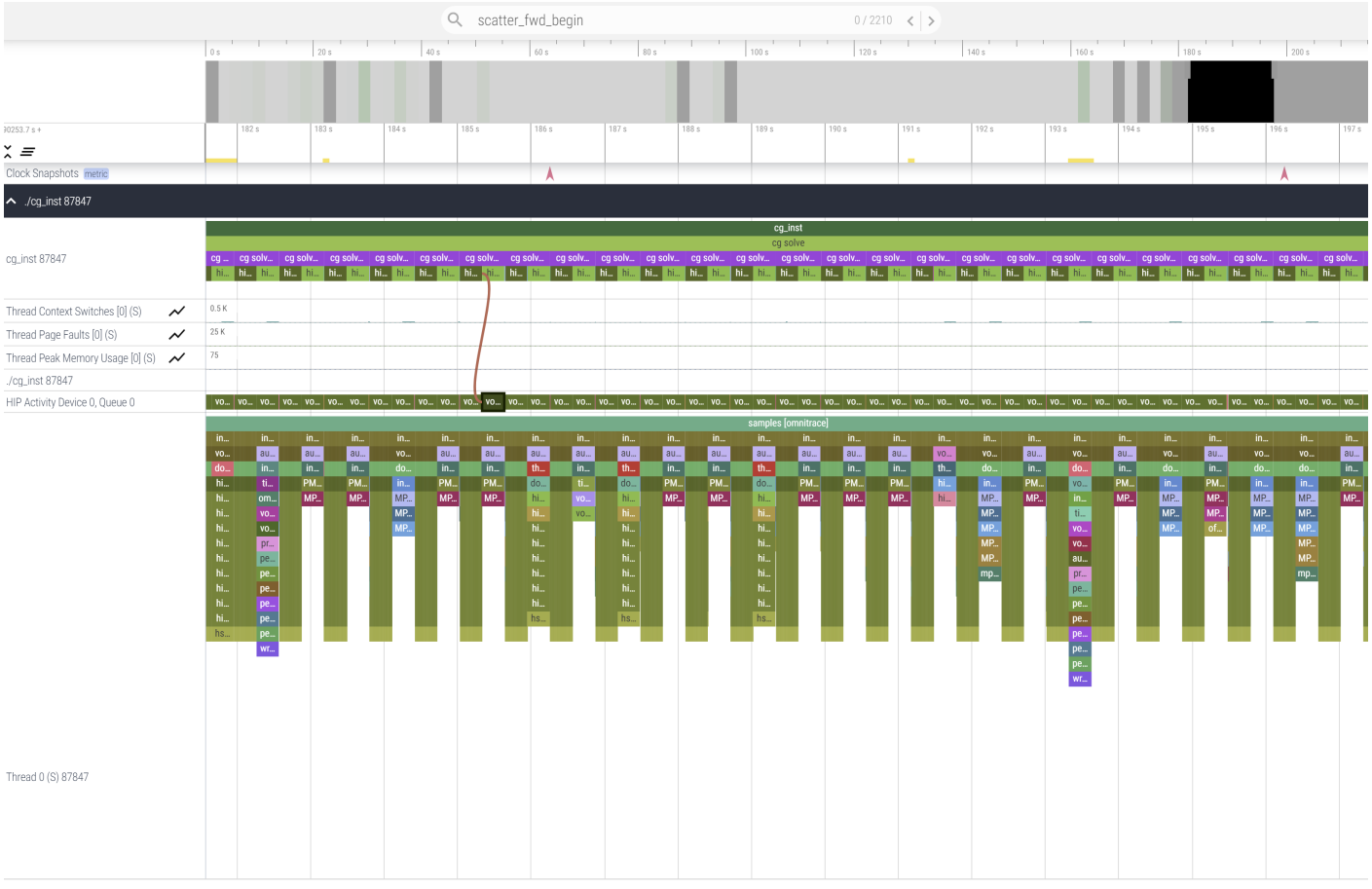


Figure: `omnitrace` profile of a CG communications in more detail

CG scatter

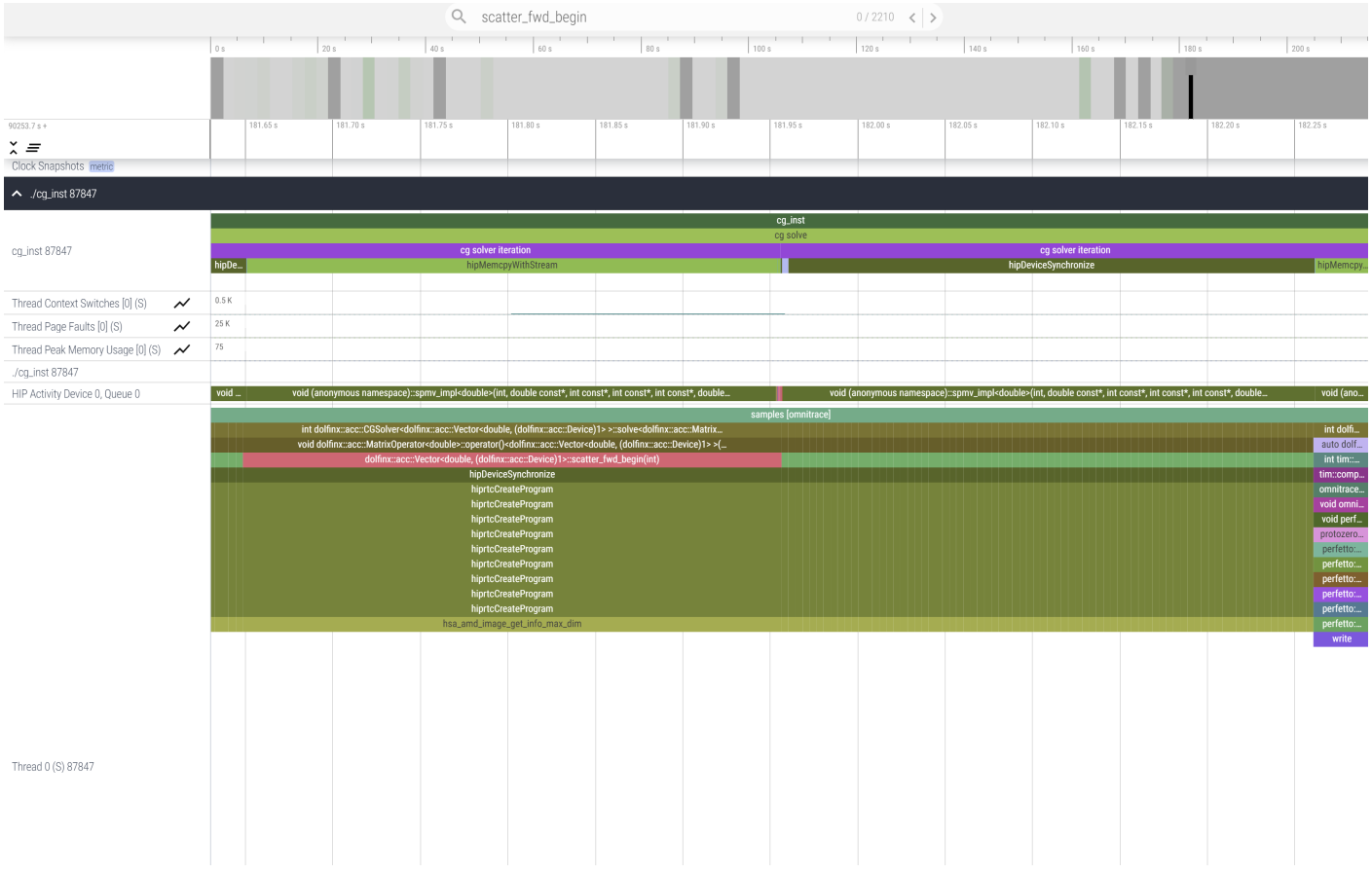


Figure: `omnitrace` profile of a CG iteration scatter communication call

CG iteration

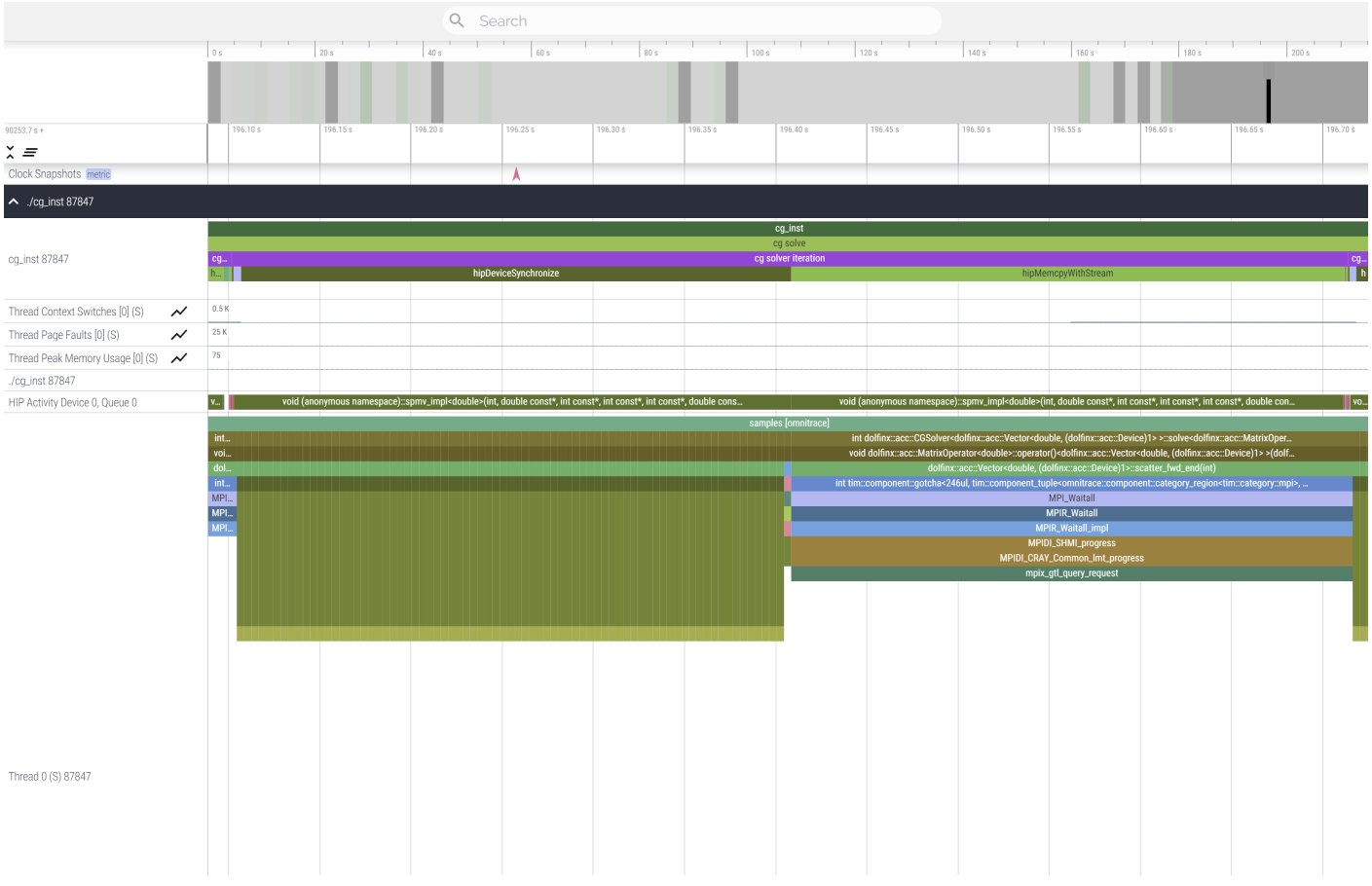


Figure: `omnitrace` profile of a CG iteration

Chebyshev communications

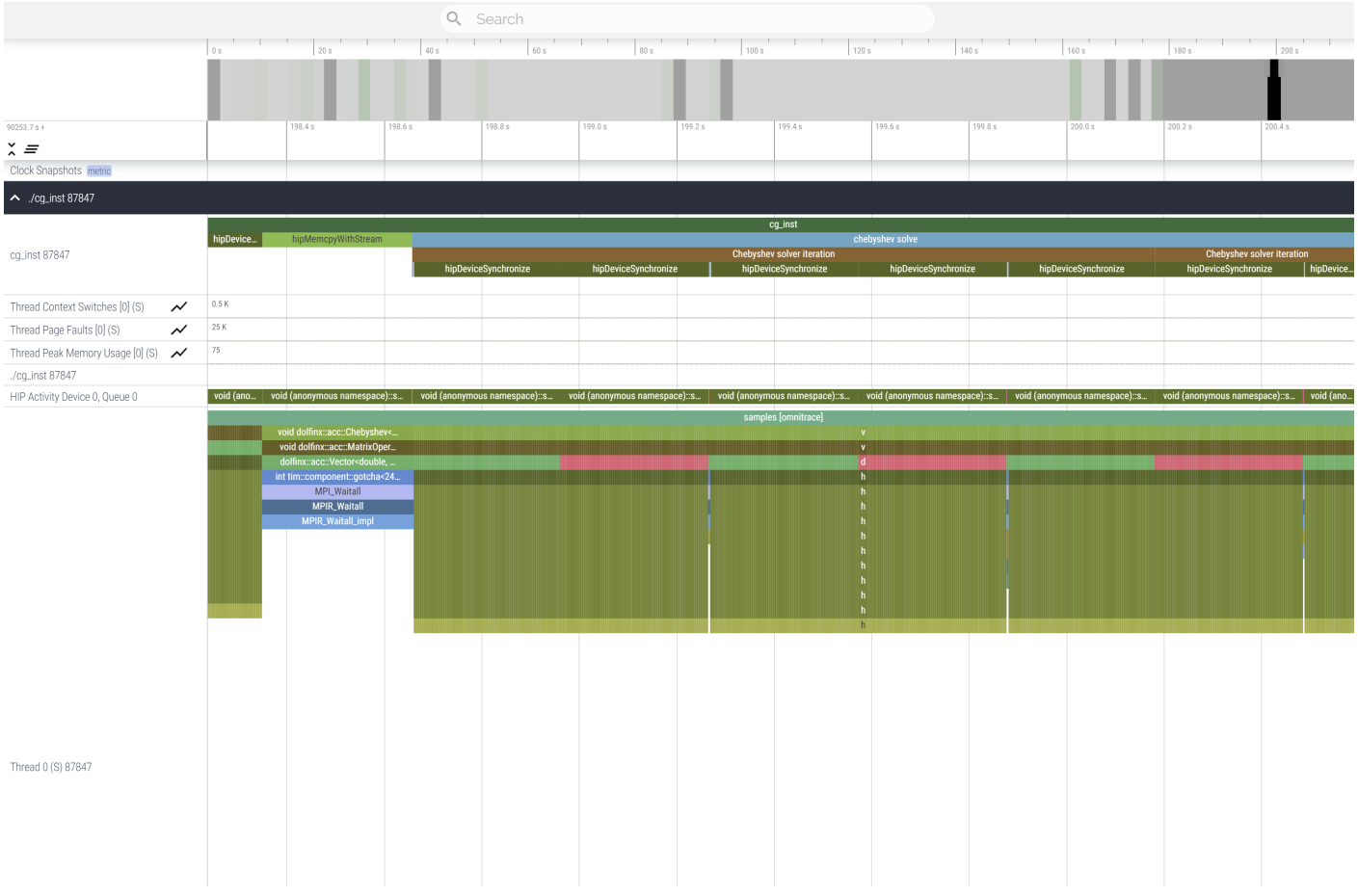
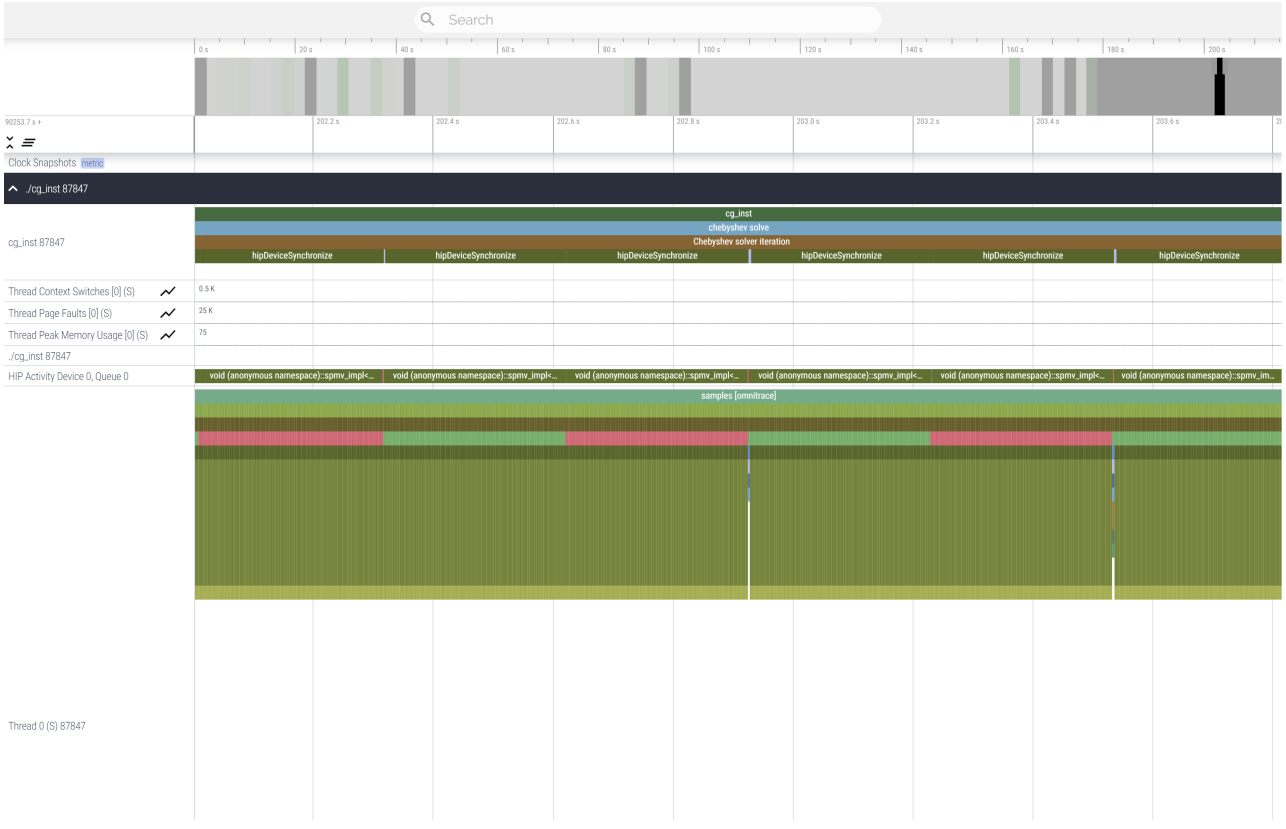


Figure: `omnitrace` profile of the Chebyshev communication phases

Chebyshev iteration



Chebyshev iteration

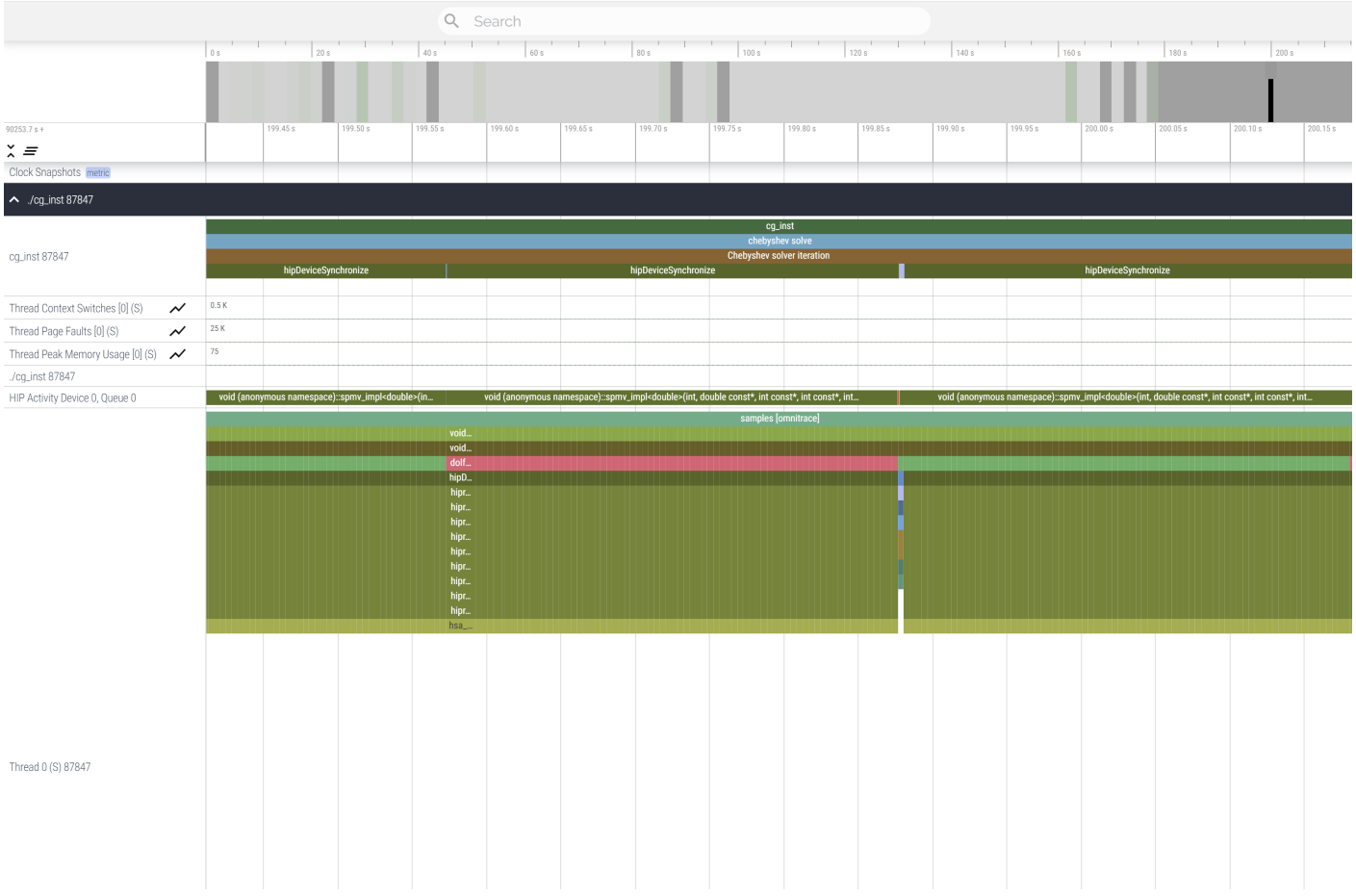


Figure: `omnitrace` profile of the Chebyshev iteration in more detail

Chebyshev scatter

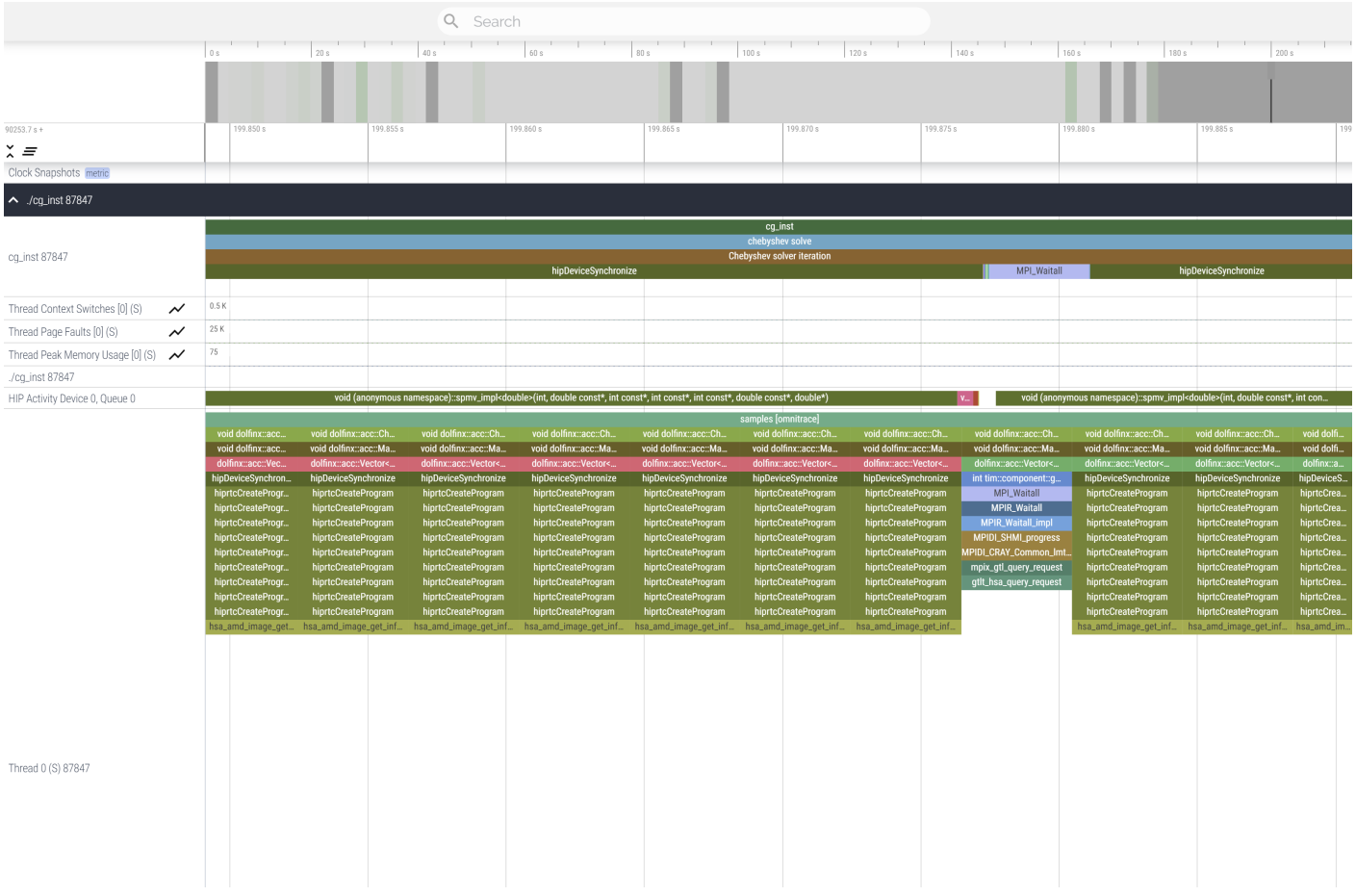


Figure: `omnitrace` profile of the Chebyshev scatter

Lessons

- GPU visibility needs to be remembered
- Failure messages not always obvious
  - Crashes sometimes appeared out of memory
  - HIP kernel errors not obvious
  - Adding error handling/checking functions can help
- Hard to know without profiling if GPU is being used
- Hard to know if libraries properly built for GPU functionality
- Hard to tell if GPU aware MPI is being used
- PETSc not currently giving us good performance

Next Steps

- Write-up report
- Integrate the preconditioner with CG solver (wip)
- Experiment with coarse grid solvers (Rocalution, Boomerang)
- Add a matrix-free operator for Q3 level
- Compare the results with A100 (CSD3 local cluster in Cambridge)
- Run full solver on CPU (Archer 2)
- Fix index overflows
  - Current implementation runs up to ~30m DoFs per GPU
  - ~40% of GPU memory