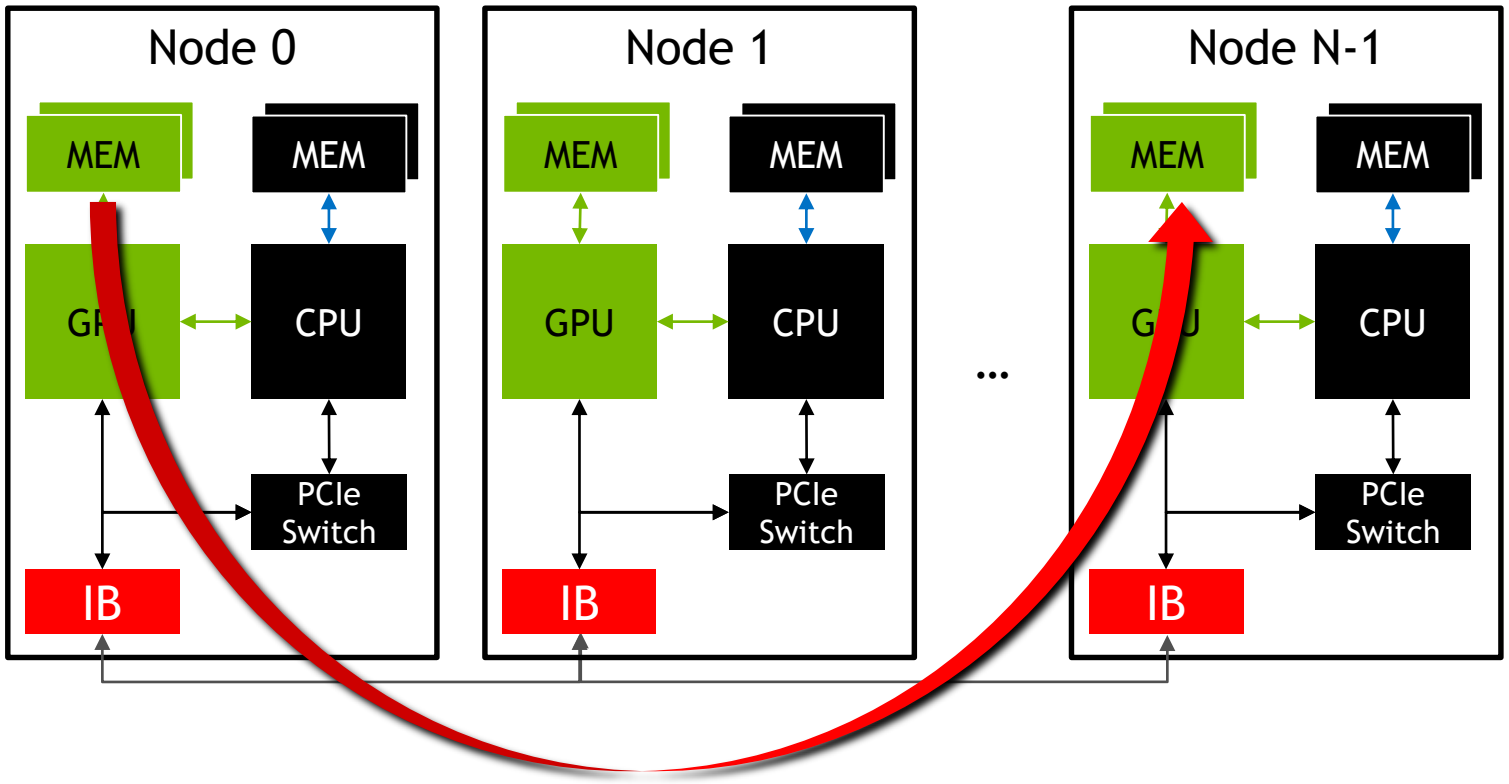
JÜLICH
FORSCHUNGSZENTRUM

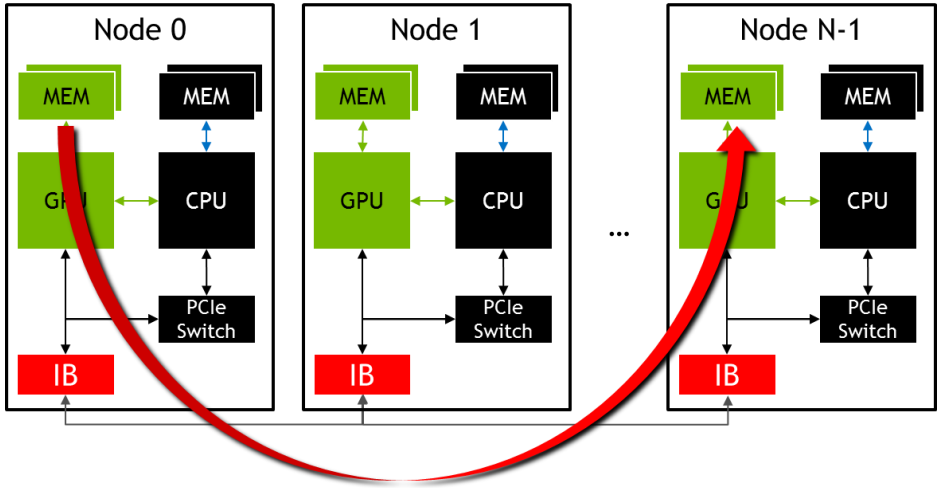
Multi GPU Programming with MPI and OpenACC

25.10.2018| J. Kraus (NVIDIA)

MPI+OpenACC



MPI+OpenACC




```
//MPI rank 0
#pragma acc host_data use_device( sbuf )
MPI_Send(sbuf, size, MPI_DOUBLE, n-1, tag, MPI_COMM_WORLD);

//MPI rank n-1
#pragma acc host_data use_device( rbuf )
MPI_Recv(rbuf, size, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

What you will learn

- What MPI is
- How to use MPI for inter GPU communication with OpenACC
- How to use pgprof for MPI+OpenACC applications
- How to hide MPI communication times

Message Passing Interface - MPI

- Standard to exchange data between processes via messages
  - Defines API to exchanges messages
    - Pt. 2 Pt.: e.g. MPI_Send, MPI_Recv
    - Collectives, e.g. MPI_Allreduce
- Multiple implementations (open source and commercial)
  - Binding for C/C++, Fortran, Python, …
  - E.g. MPICH, OpenMPI, MVAPICH, IBM Platform MPI, Cray MPT, …

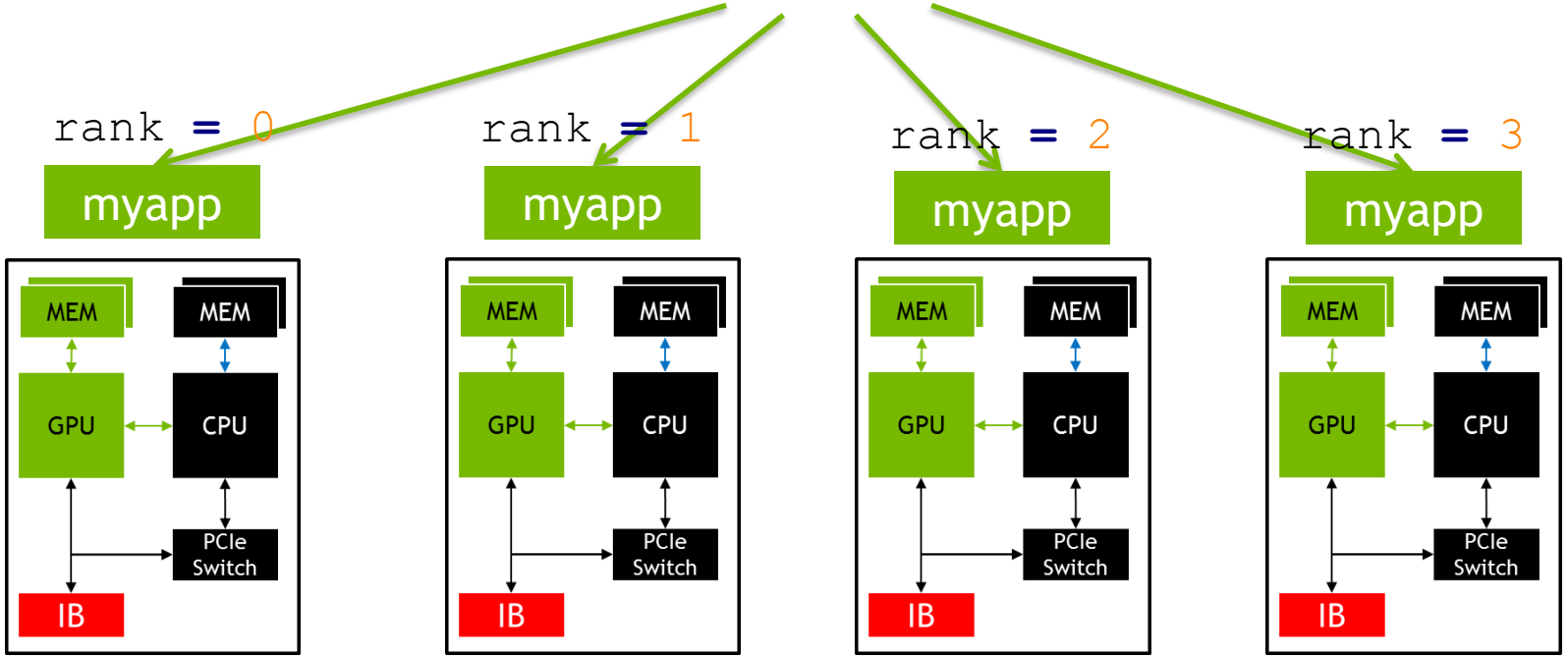
MPI – A minimal program

```c
#include <mpi.h>
int main(int argc, char *argv[]) {
    int rank,size;
    /* Initialize the MPI library */
    MPI_Init(&argc,&argv);
    /* Determine the calling rank and total number of ranks */
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    /* Call MPI routines like MPI_Send, MPI_Recv, ... */
    ...
    /* Shutdown MPI library */
    MPI_Finalize();
    return 0;
}
```

Remark: Almost all MPI routines return an error value which should be checked. The examples and tasks leave that out for brevity.

MPI – Compiling and Launching

```
$ mpicc -o myapp myapp.c
$ srun -n 4 ./myapp <args>
```



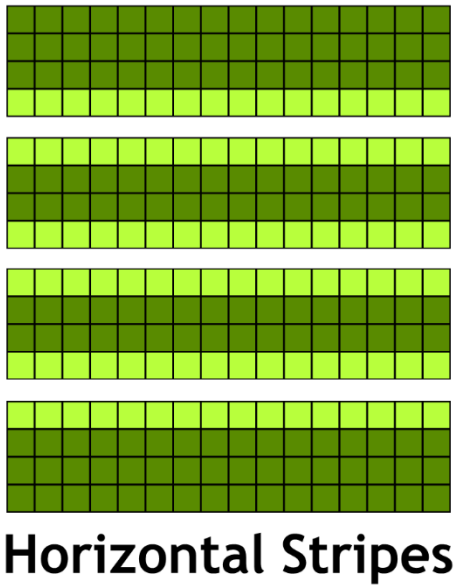
Example: Jacobi Solver

Solves the 2D-Poission Equation on a rectangle

$$\Delta u(x, y) = e^{-10*(x^2+y^2)} \; \forall \; (x, y) \in \Omega \backslash \delta\Omega$$

Periodic boundary conditions

Domain decomposition with stripes



Example: Jacobi Solver – Single GPU

While not converged

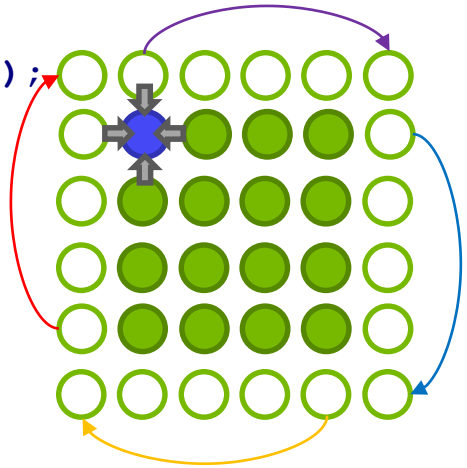
Do Jacobi step:

```
for (int iy = 1; iy < ny-1; ++iy)
for (int ix = 1; ix < nx-1; ++ix)
  Anew[iy*nx+ix]=-0.25f*(rhs[iy*nx+ix]
                  -(A[iy*nx+(ix-1)]+A[iy*nx+(ix+1)]
                   +A[(iy-1)*nx+ix]+A[(iy+1)*nx+ix]));
```

Copy `Anew` to `A`

Apply periodic boundary conditions

Next iteration



Domain Decomposition

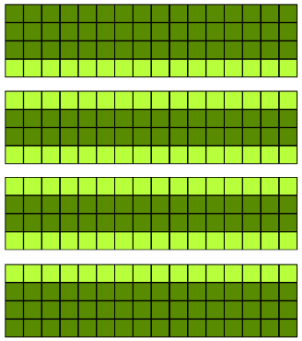
Different ways to split the work between processes:

Minimizes number of neighbors:

- Communicate to less neighbors
- Optimal for latency bound communication

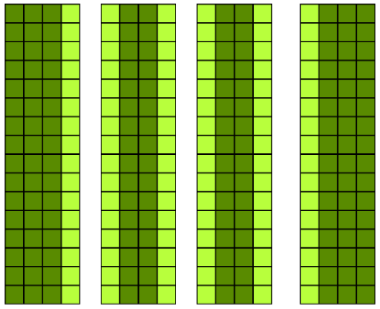
Minimizes surface area/volume ratio:

- Communicate less data
- Optimal for bandwidth bound communication



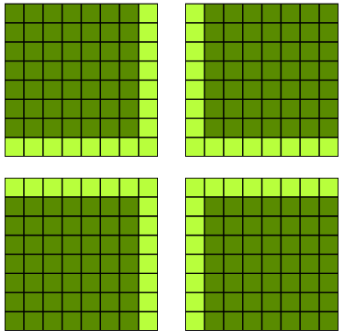
Horizontal Stripes

Contiguous if data is row-major



Vertical Stripes

Contiguous if data is column-major



Tiles

Example: Jacobi Solver – Multi GPU

While not converged

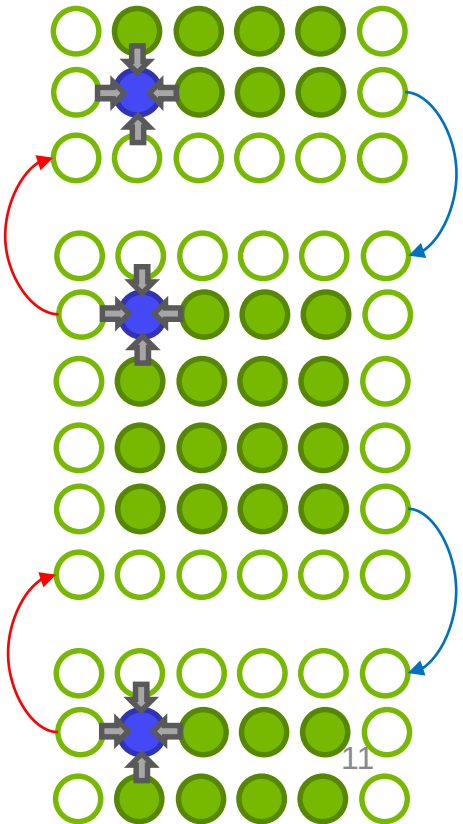
Do Jacobi step:

```
for (int iy = iy_start; iy < iy_end; ++iy)
for (int ix = 1; ix < NX-1; ++ix)
  Anew[iy*nx+ix]=-0.25f*(rhs[iy*nx+ix]
                  -(A[iy*nx+(ix-1)]+A[iy*nx+(ix+1)]
                   +A[(iy-1)*nx+ix]+A[(iy+1)*nx+ix]));
```

Copy `Anew` to `A`

Apply periodic boundary conditions and exchange halo with 2 neighbors

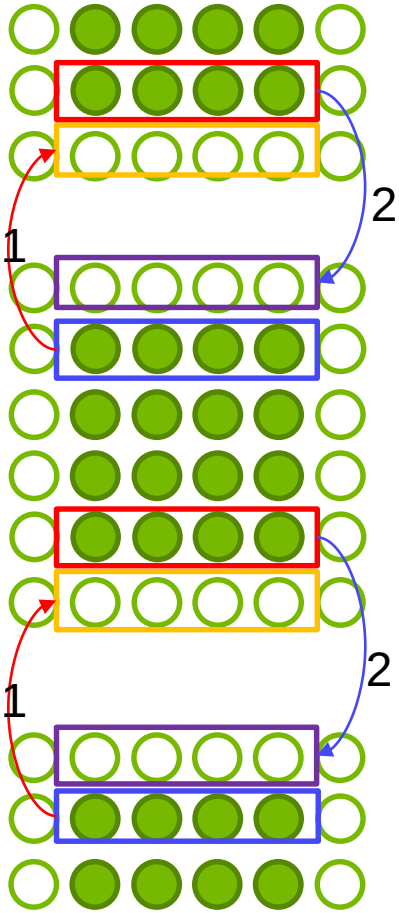
Next iteration



Example: Jacobi – Top/Bottom Halo

```
#pragma acc host_data use_device ( A ) {
MPI_Sendrecv(A+iy_start*nx+1, nx-2, MPI_DOUBLE, top, 0,
             A+iy_end*nx+1, nx-2, MPI_DOUBLE, bottom, 0,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);

MPI_Sendrecv(A+(iy_end-1)*nx+1, nx-2, MPI_DOUBLE, bottom, 1,
             A+(iy_start-1)*nx+1, nx-2, MPI_DOUBLE, top, 1,
             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```



Handling GPU Affinity

- Rely on process placement (with on rank per GPU)*

```
#if _OPENACC

acc_device_t device_type = acc_get_device_type();

int ngpus=acc_get_num_devices(device_type);

int devicenum=rank%ngpus;

#endif /*_OPENACC*/

#pragma acc set device_num( devicenum )

#if _OPENACC // Or using the API:

acc_set_device_num(devicenum,device_type);

#endif /*_OPENACC*/
```

Alternative (JURECA):

```
int devicenum = atoi(getenv("MPI_LOCALRANKID"));
```

* This assumes the node is homogeneous, i.e. that all the GPUs are the same. If you have different GPUs in the same node then you may need some more complex GPU selection

Profiling MPI+OPENACC applications

Embed MPI rank in output filename, process name, and context name

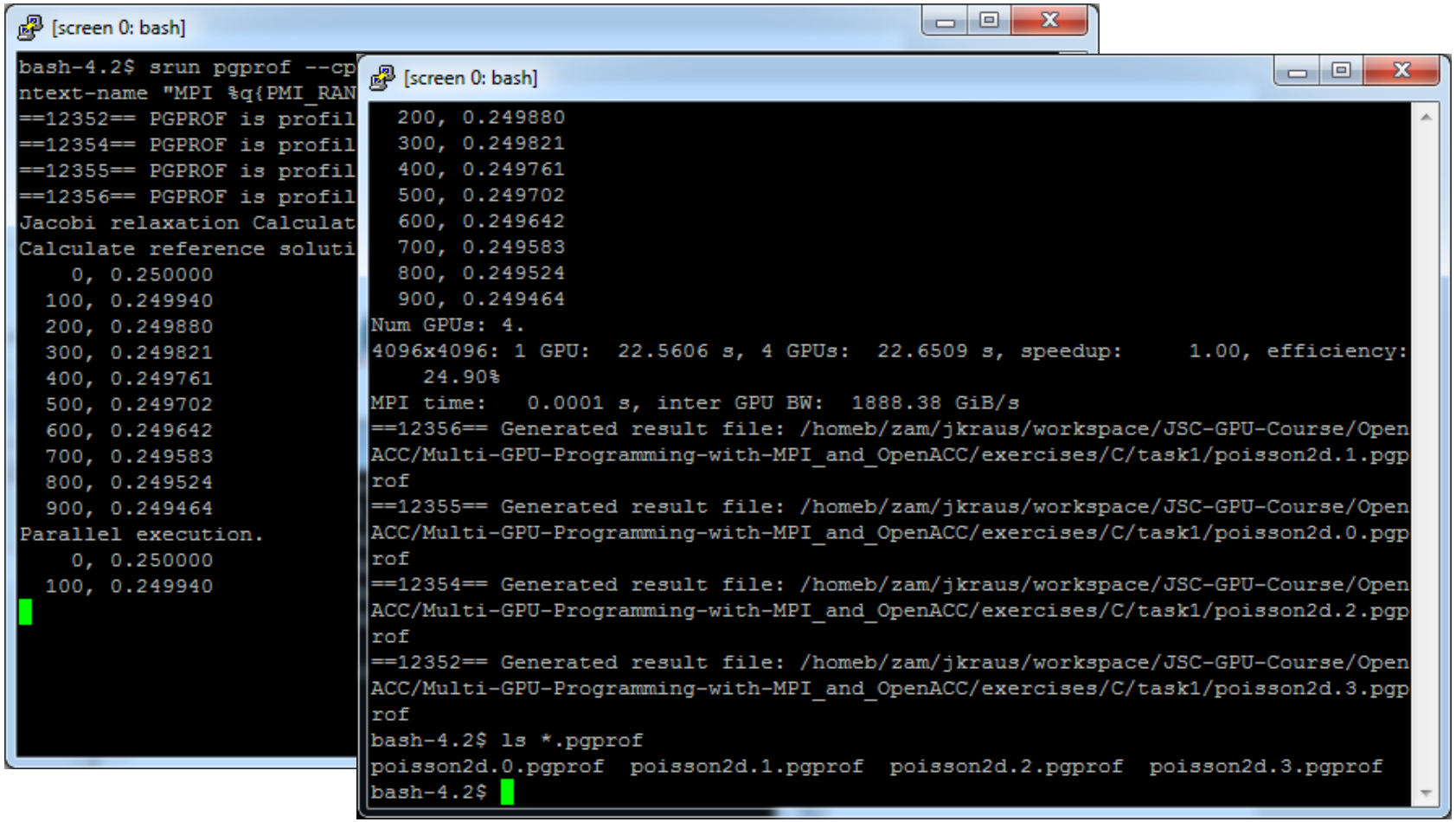
```
srun pgprof --cpu-profiling off
        --output-profile profile.%q{PMI_RANK}          \
         --process-name "rank %q{PMI_RANK}"            \
         --context-name "rank %q{PMI_RANK}"
```

JURECA: `PMI_RANK`

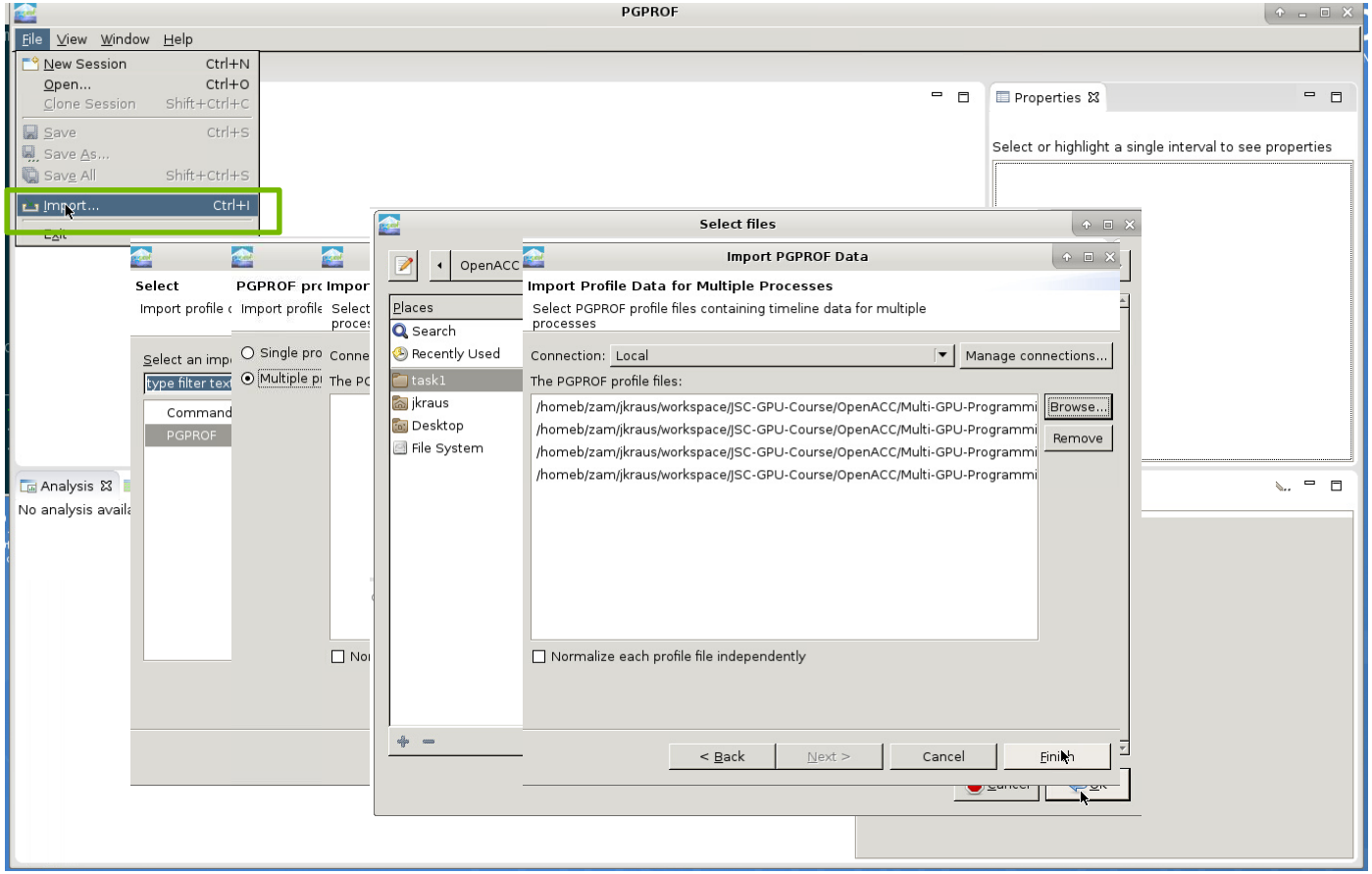
OpenMPI: `OMPI_COMM_WORLD_RANK`

MVAPICH2: `MV2_COMM_WORLD_RANK`

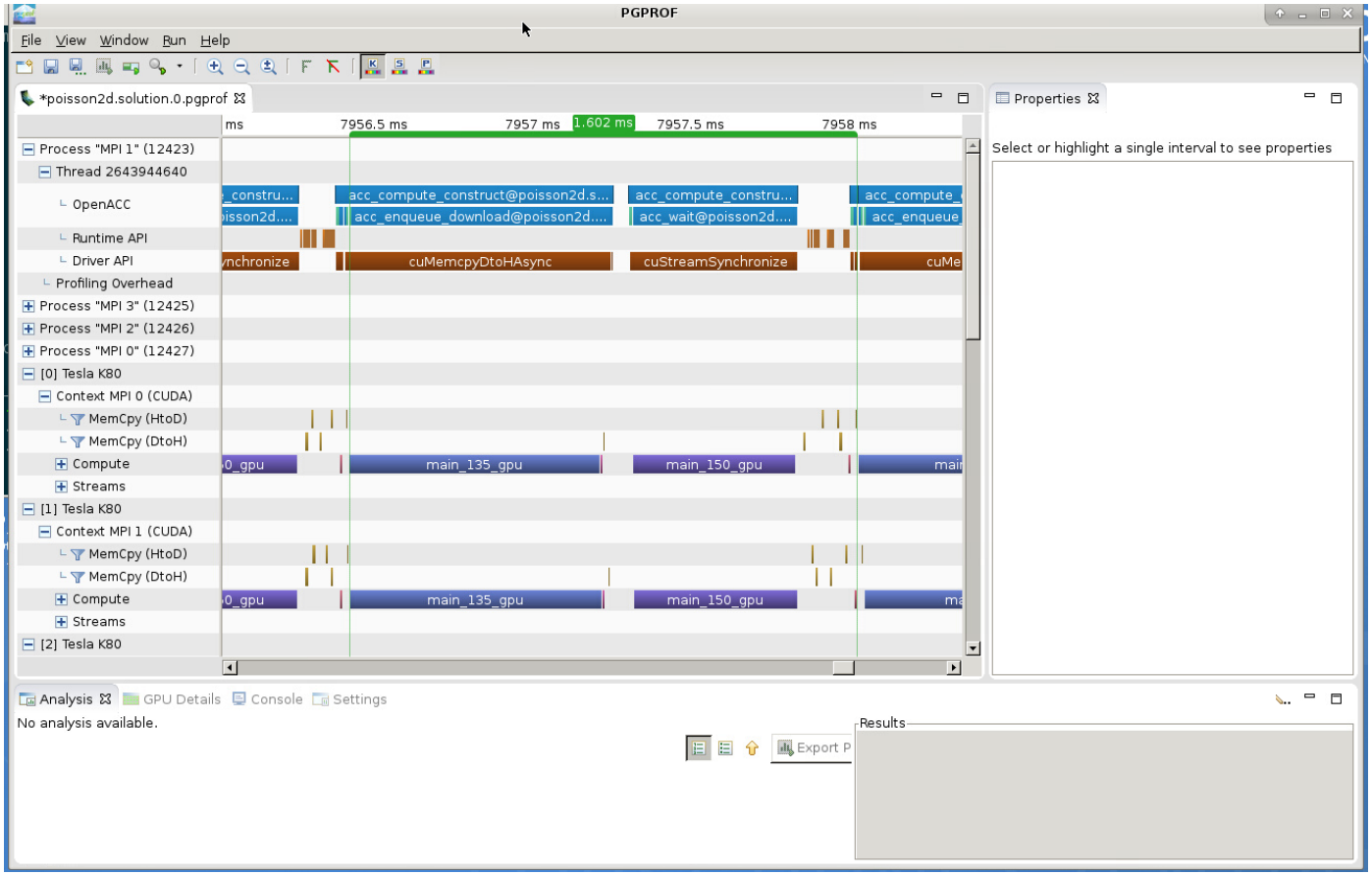
Profiling MPI+OpenACC applications



Profiling MPI+OpenACC applications



Profiling MPI+OpenACC applications

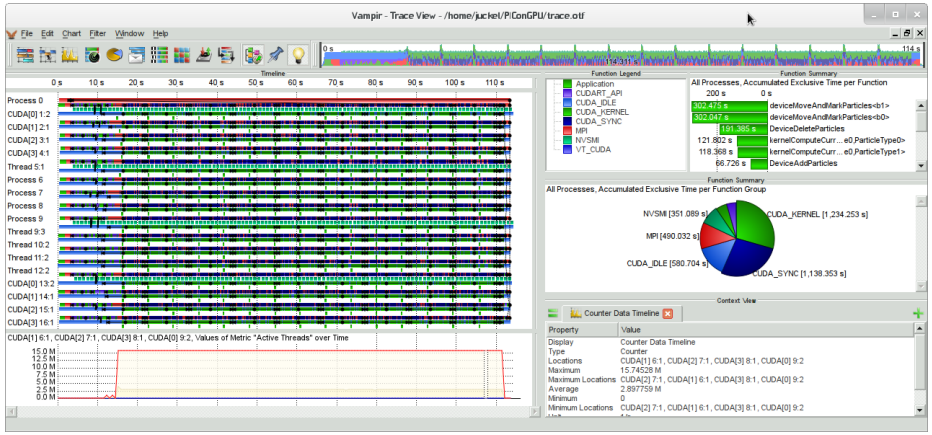


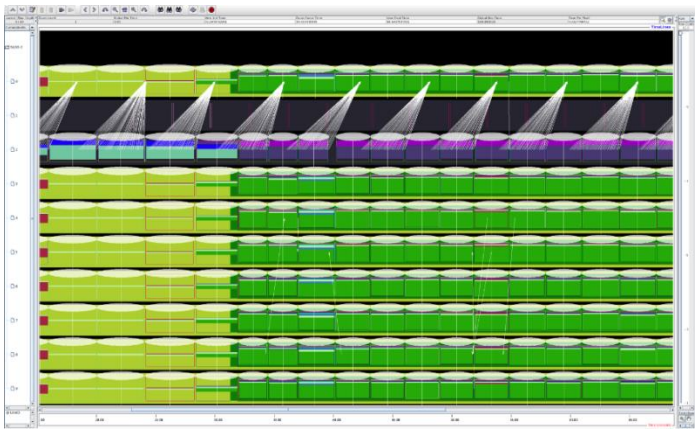
Profiling MPI+OpenACC applications

Multiple parallel profiling tools are CUDA-aware

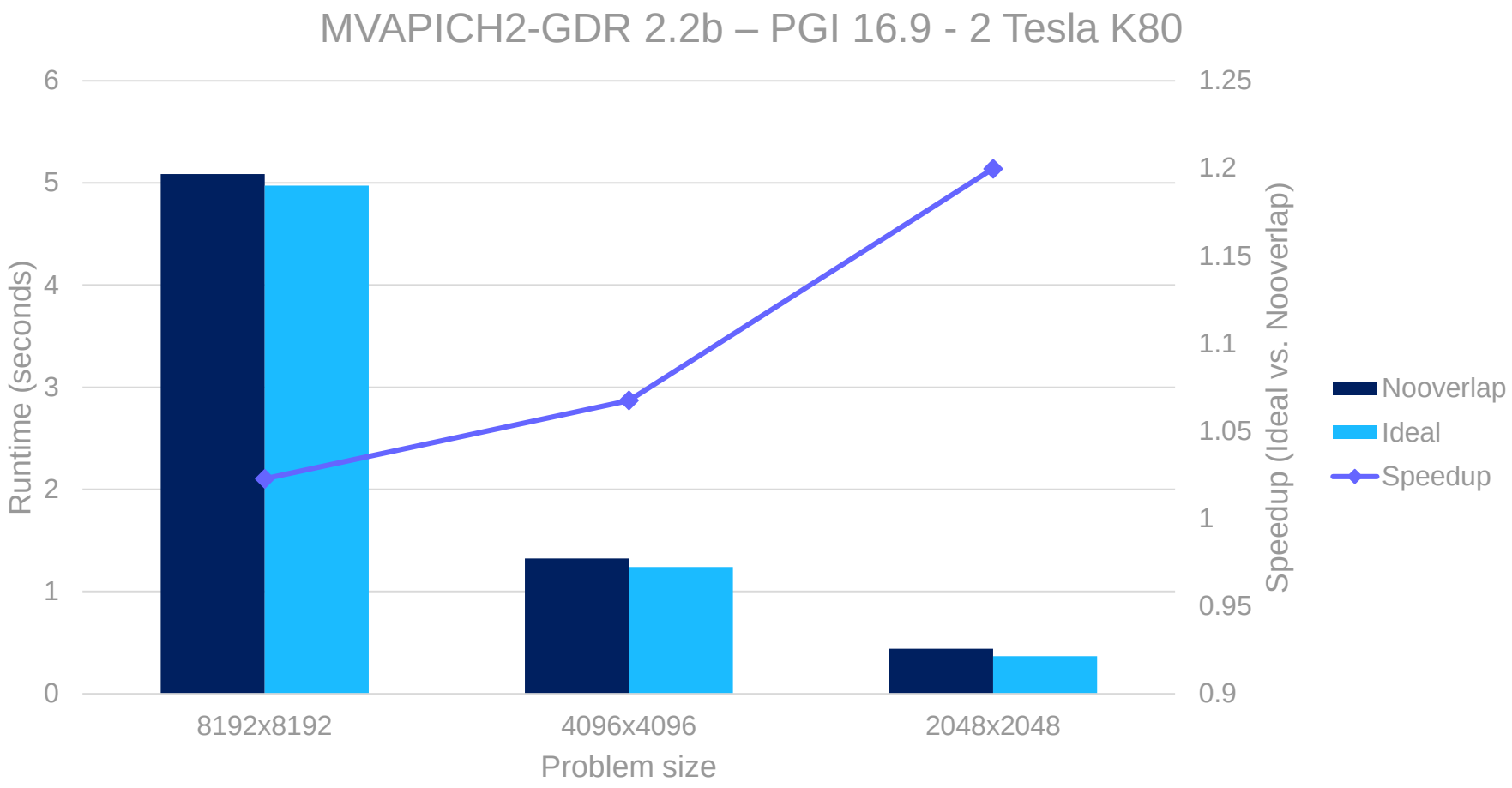
- Score-P
- Vampir
- Tau

These tools are good for discovering MPI issues as well as basic CUDA performance inhibitors.

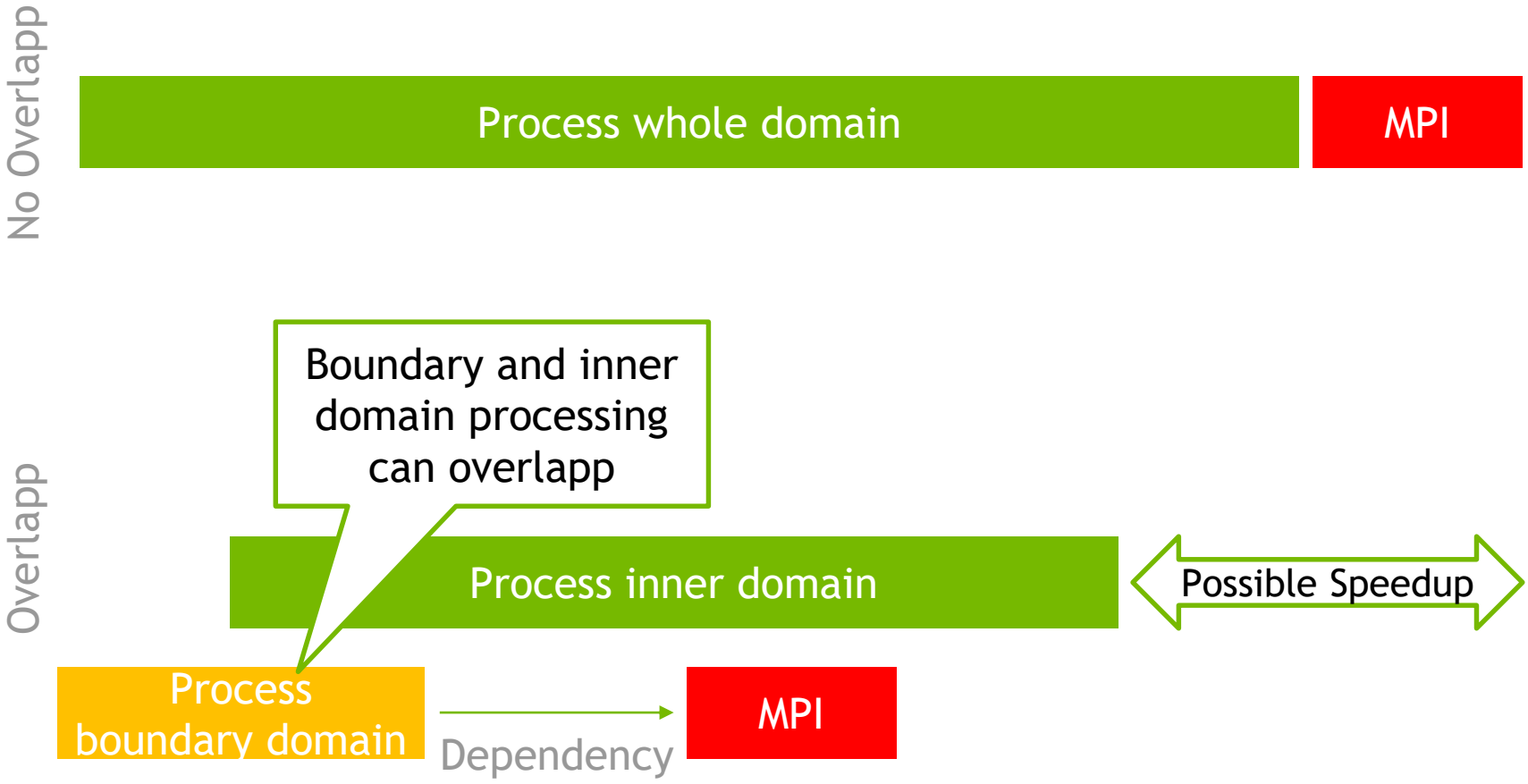




Communication + Computation Overlap



Communication + Computation Overlap



Communication + Computation Overlap

```
#pragma acc parallel loop
for ( ... )
          //Process boundary
#pragma acc parallel loop async
for ( ... )
          //Process inner domain

#pragma acc host_data use_device ( A )
{
  //Exchange halo with top and bottom neighbor
  MPI_Sendrecv( A…);
  //…
}
//wait for iteration to finish
#pragma acc wait
```

Scalability Metrics For Success

- Serial Time: Ts:
  - How long it takes to run the problem with a single process
- Parallel Time: Tp
  - How long it takes to run the problem with multiple processes
- Number of Processes: P
  - The number of Processes operating on the task at hand
- Speedup: $S=Ts/Tp$
  - How much faster is the parallel version vs. serial. (optimal is P)
- Efficiency: $E= S/P$
  - How efficient are the processors used (optimal is 1)

Task 1: Apply domain decomposition

Look for TODOs

Handle GPU affinity

Halo Exchange

```
$ make
mpicc -c -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla poiss
srun ./poisson2d
Jacobi relaxation Calculation: 4096 x 4096 mesh
[...]
Num GPUs: 4.
4096x4096: 1 GPU:  18.9390 s, 4 GPUs:  18.8453 s, speedup:
MPI time:   0.0001 s, inter G
```

Make Targets:

`run:`	run `poisson2d` (default)
`poisson2d:`	build `poisson2d` binary
`profile:`	profile with `pgprof`
`*.solution:`	same as above with solution (`poisson2d.solution.*`)

Task 2: Hide MPI communication time

Look for TODOs

Start copy loop asynchronously

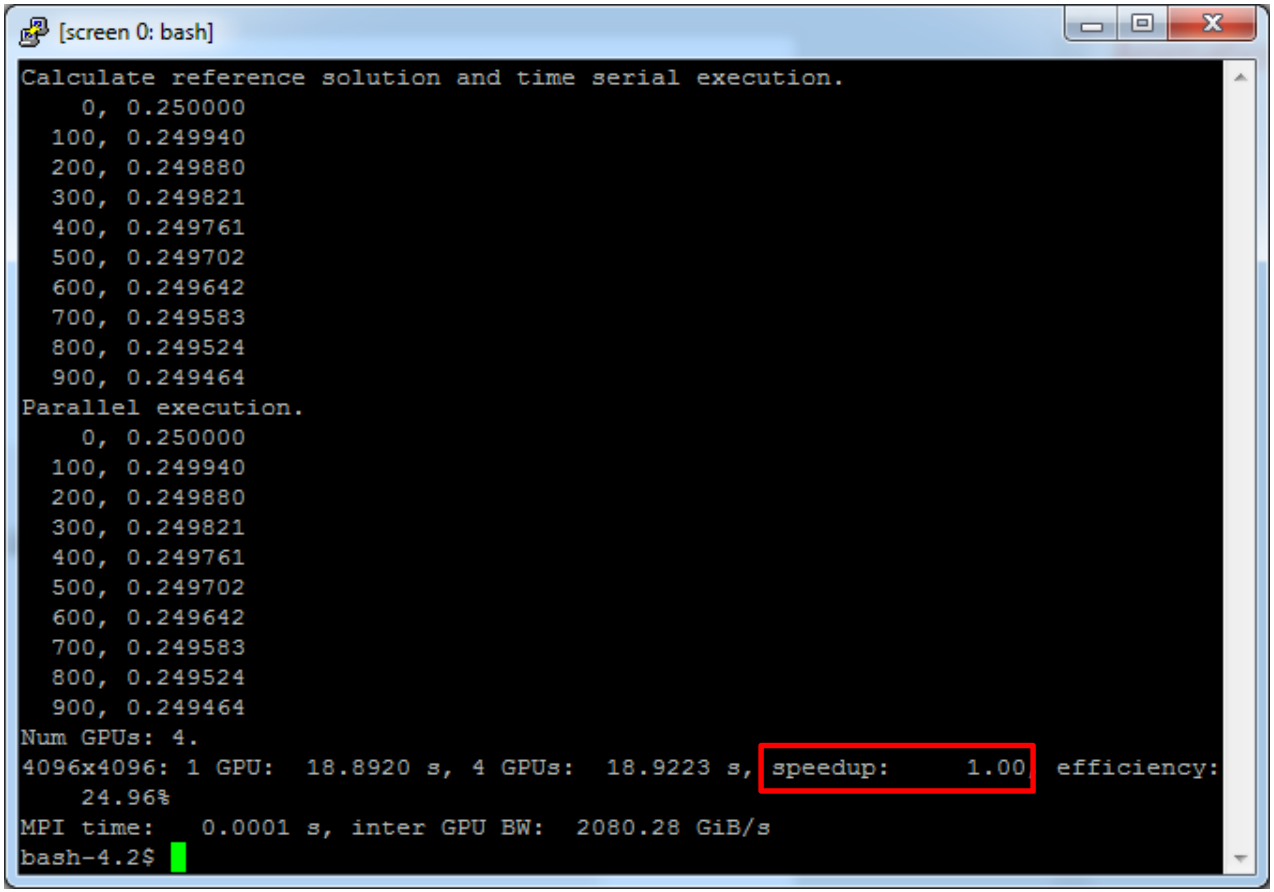
Wait for async copy loop after MPI comm. is done

```
$ make
mpicc -c -DUSE_DOUBLE -Minfo=accel -fast -acc -ta=tesla poiss
srun ./poisson2d
Jacobi relaxation Calculation: 4096 x 4096 mesh
[...]
Num GPUs: 4.
4096x4096: 1 GPU:   4.6780 s, 4 GPUs:   1.3290 s, speedup:
MPI time:   0.0846 s, inter G
```

Make Targets:

`run:`	run `poisson2d` (default)
`poisson2d:`	build `poisson2d` binary
`profile:`	profile with `pgprof`
`*.solution:`	same as above with solution (`poisson2d.solution.*`)

Task 1: Initial Version



Task 1: Solution

```
//Initialize MPI and determine rank and size
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

#pragma acc set device_num( rank )

real* restrict const A = (real*) malloc(nx*ny*sizeof(real));
real* restrict const Aref = (real*) malloc(nx*ny*sizeof(real));
real* restrict const Anew = (real*) malloc(nx*ny*sizeof(real));
real* restrict const rhs = (real*) malloc(nx*ny*sizeof(real));
```

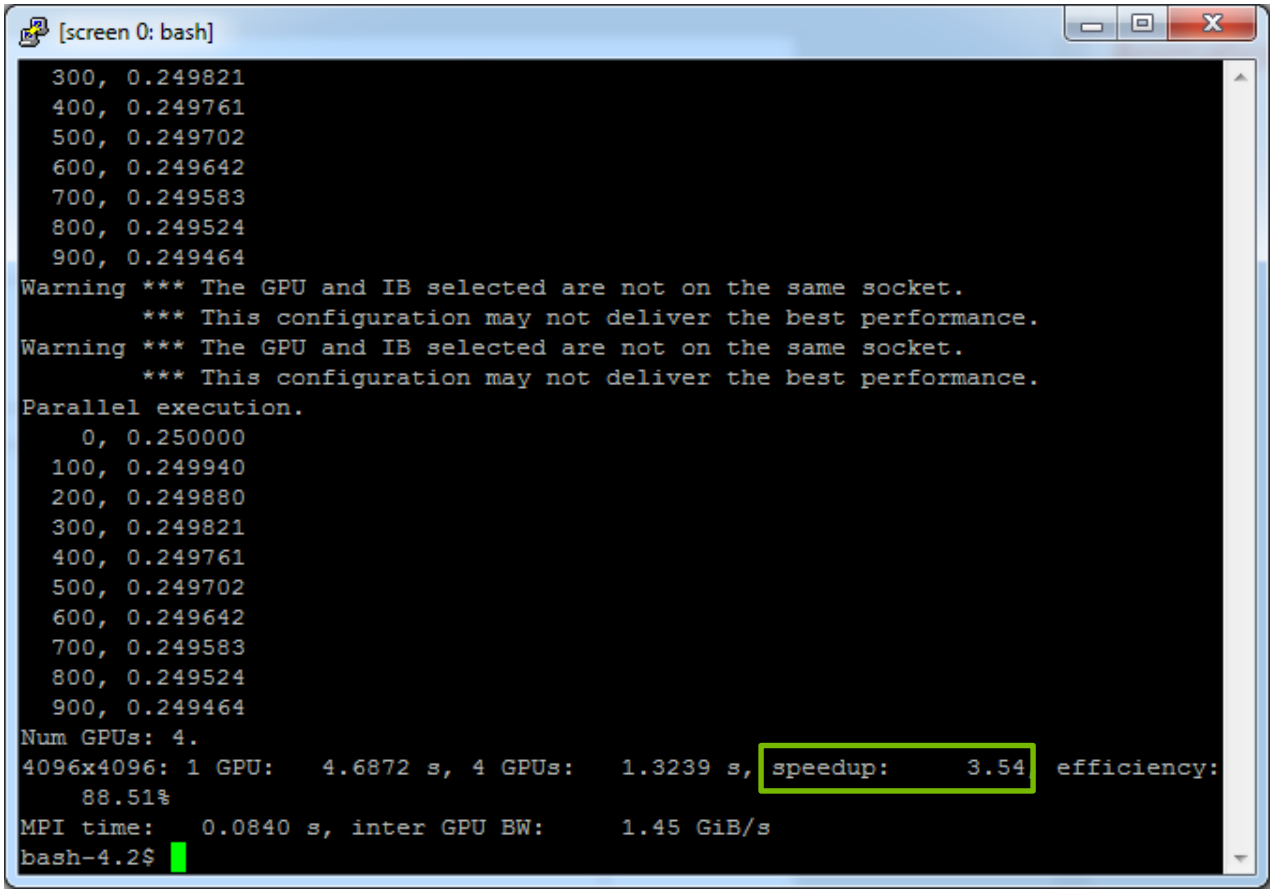
Task 1: Solution

```
// Ensure correctness if ny%size != 0
int chunk_size = ceil( (1.0*ny)/size );
int iy_start = rank * chunk_size;
int iy_end = iy_start + chunk_size;
// Do not process boundaries
iy_start = max( iy_start, 1 );
iy_end = min( iy_end, ny - 1 );
```

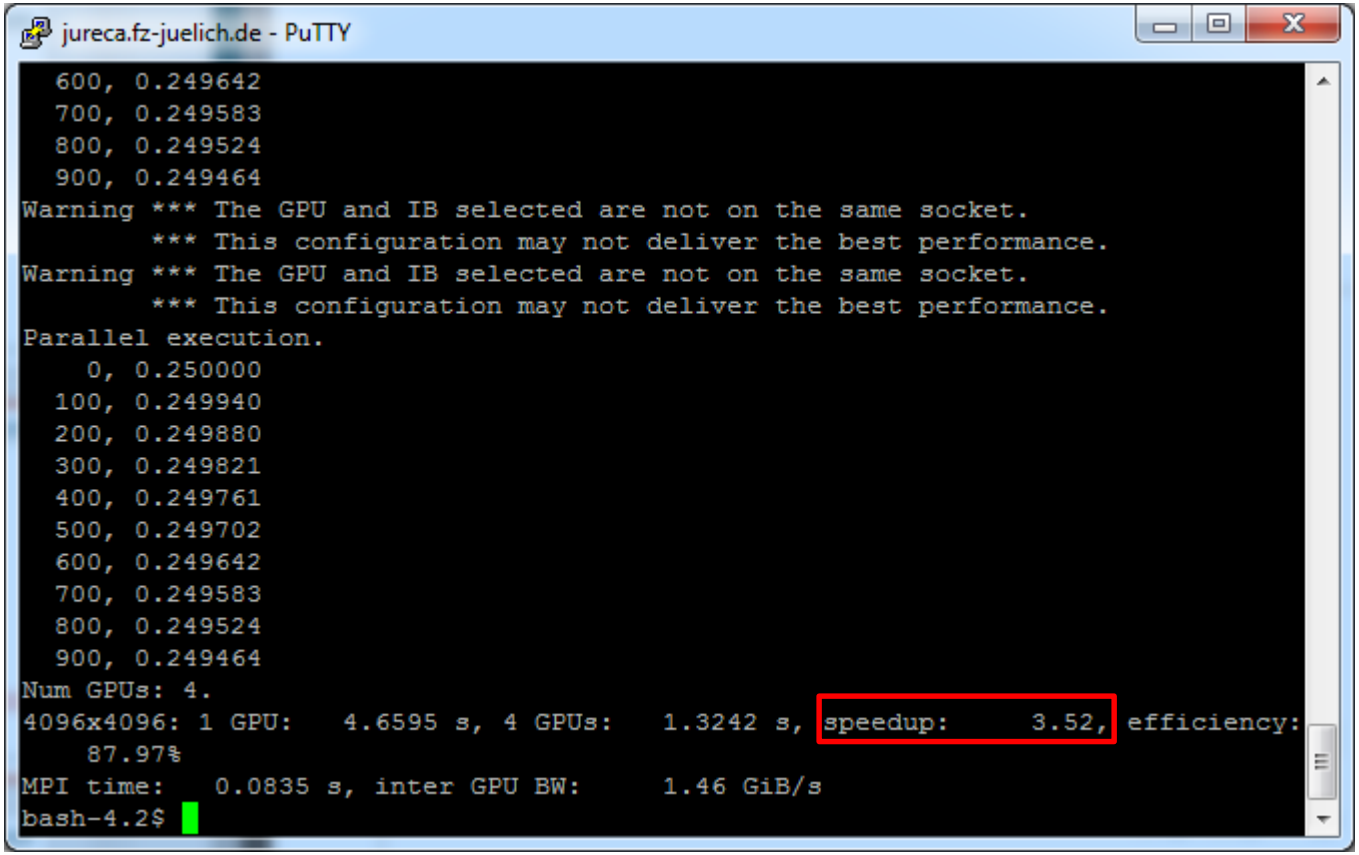
Task 1: Solution

```
int top = (rank == 0) ? (size-1) : rank-1;
int bottom = (rank == (size-1)) ? 0 : rank+1;
#pragma acc host_data use_device( A ) {
  //1. Sent row iy_start (first modified row) to top receive lower boundary (iy_end)
  //from bottom
  MPI_Sendrecv( A+iy_start*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, top , 0,
                A+iy_end*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, bottom, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE );
  //2. Sent row (iy_end-1) (last modified row) to bottom
  //receive upper boundary (iy_start-1) from top
  MPI_Sendrecv( A+(iy_end-1)*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, bottom, 0,
                A+(iy_start-1)*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, top , 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
```

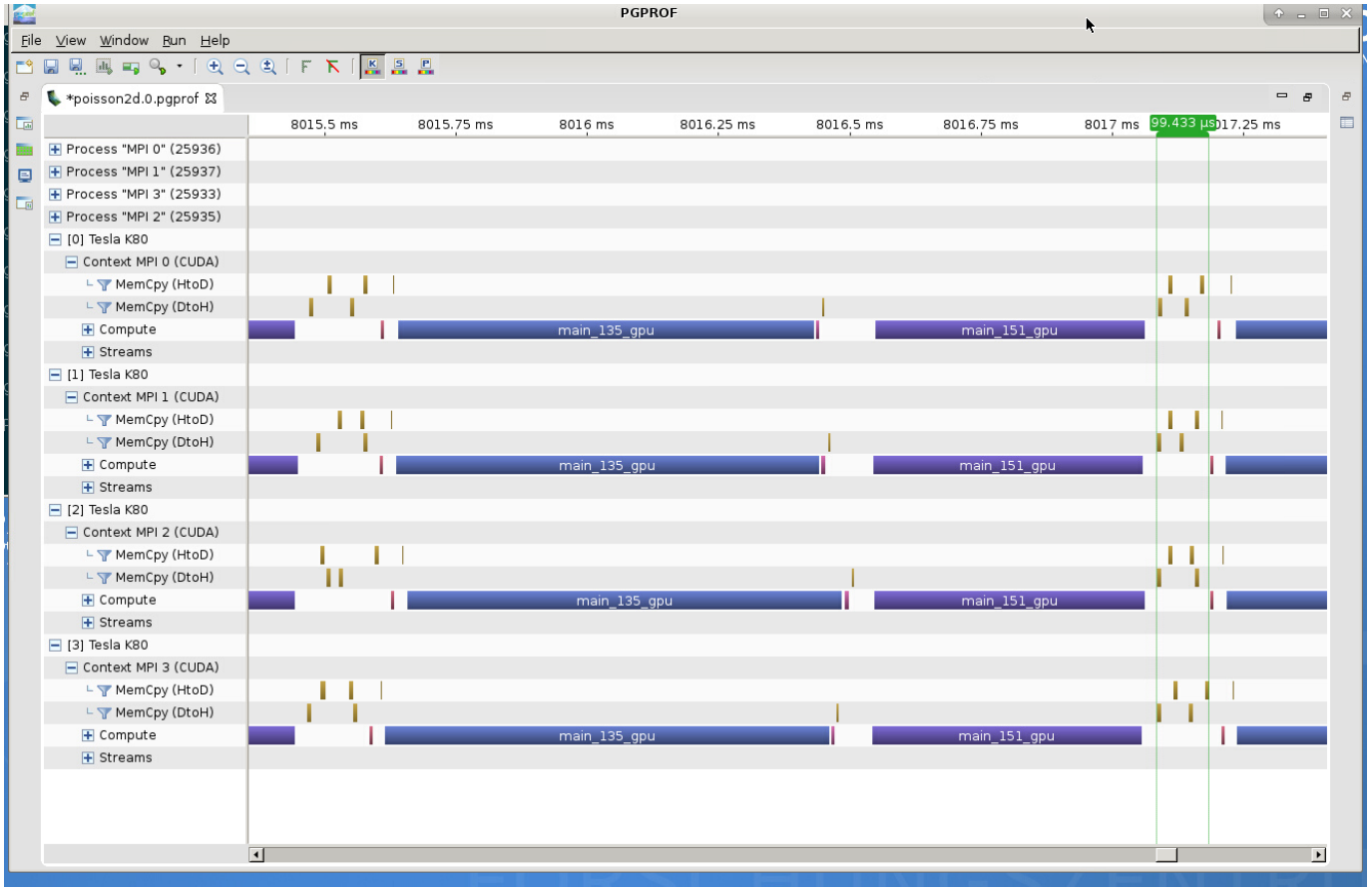
Task 1: Solution



Task 2: Initial Version



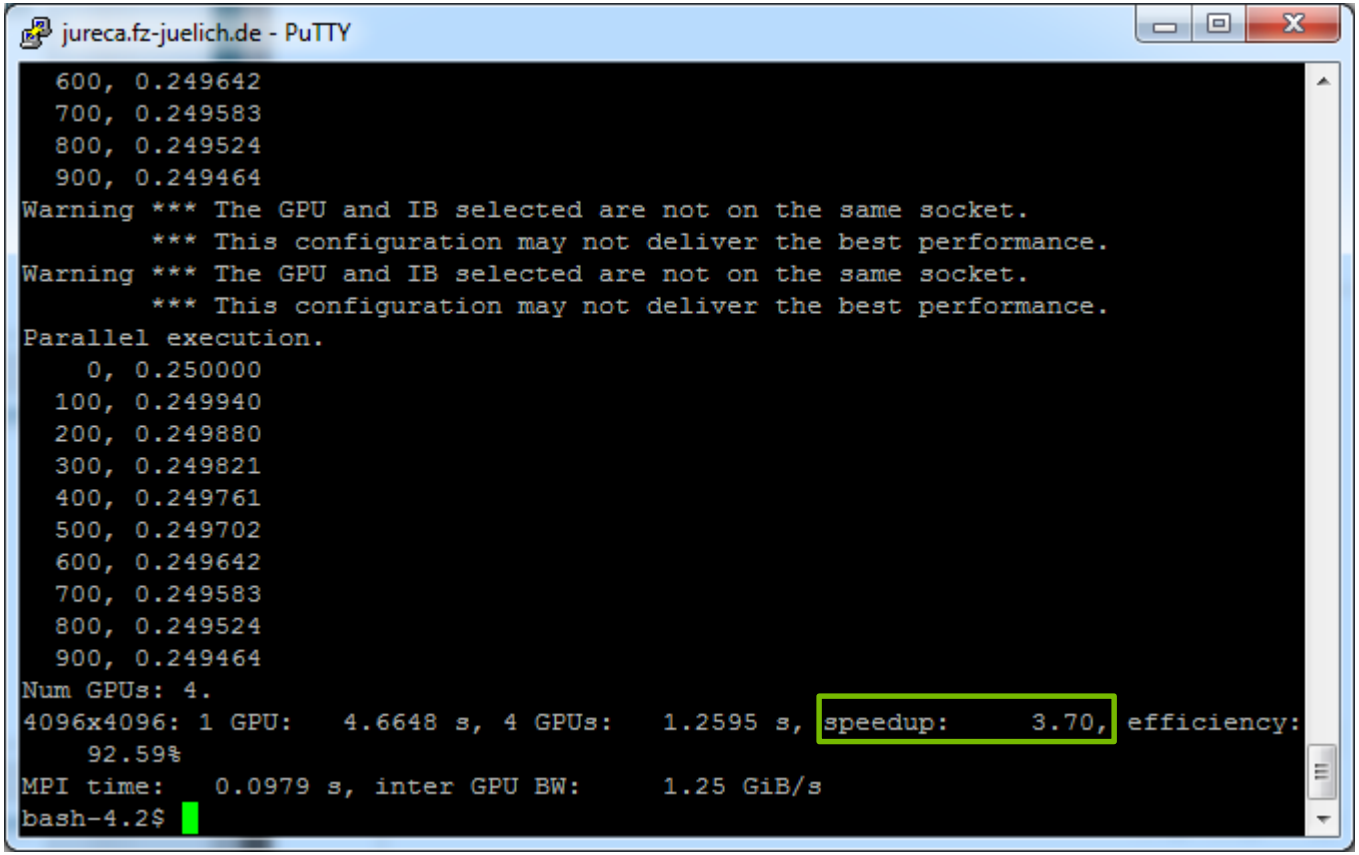
Task 2: Initial Version



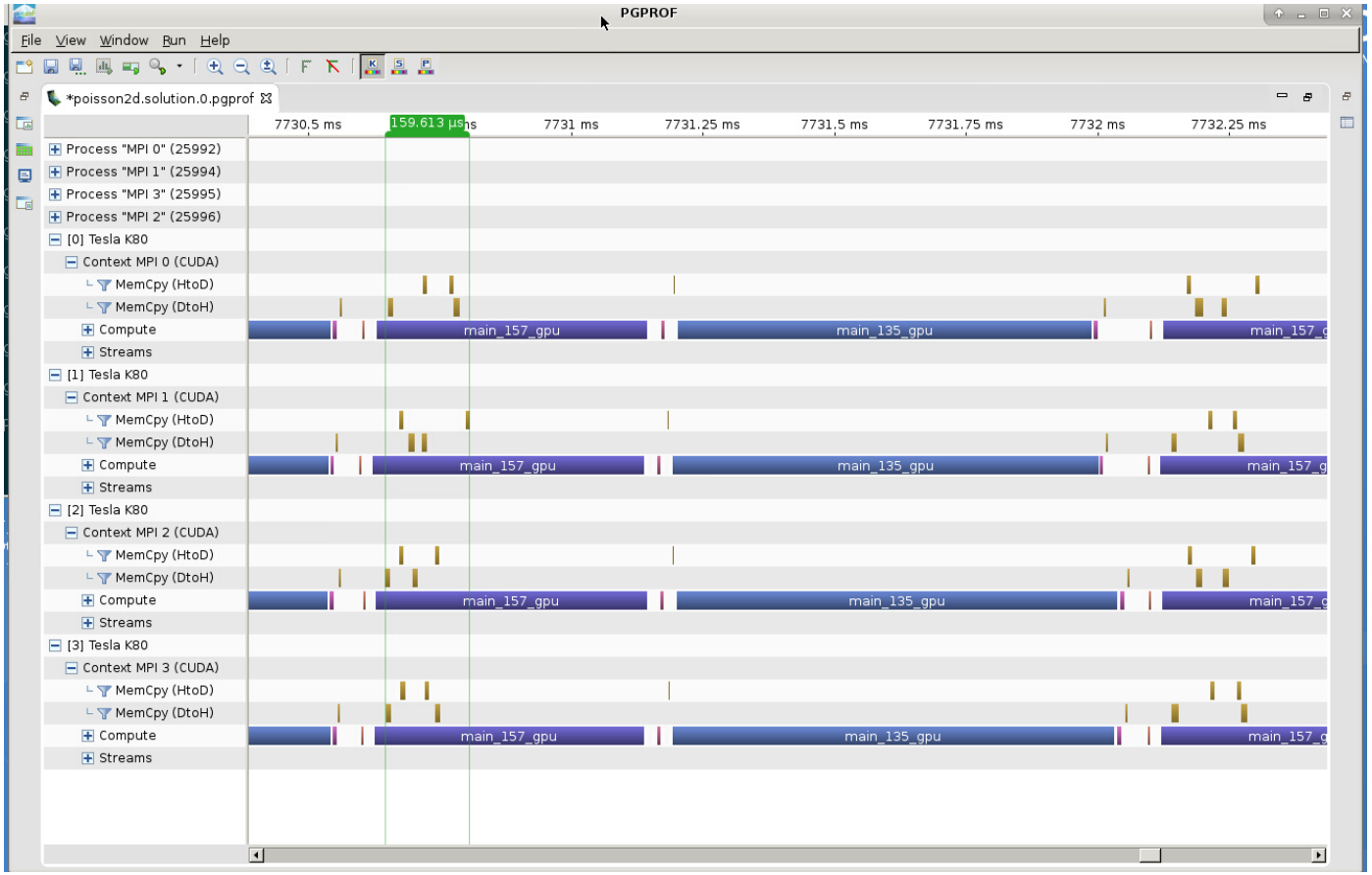
Task 2: Solution

```
#pragma acc parallel loop present(A,Anew)
for( int ix = ix_start; ix < ix_end; ix++ ) {
  A[(iy_start)*nx+ix] = Anew[(iy_start)*nx+ix];
  A[(iy_end-1)*nx+ix] = Anew[(iy_end-1)*nx+ix];
}
#pragma acc parallel loop present(A,Anew) async
for (int iy = iy_start+1; iy < iy_end-1; iy++) {
for( int ix = ix_start; ix < ix_end; ix++ ) {
  A[iy*nx+ix] = Anew[iy*nx+ix];
} }
int top = (rank == 0) ? (size-1) : rank-1;
int bottom = (rank == (size-1)) ? 0 : rank+1;
#pragma acc host_data use_device( A )
{
  MPI_Sendrecv( A+iy_start*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, top , 0,
                A+iy_end*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, bottom, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE );
  MPI_Sendrecv( A+(iy_end-1)*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, bottom, 0,
                A+(iy_start-1)*nx+ix_start, (ix_end-ix_start), MPI_REAL_TYPE, top , 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE );
}
#pragma acc wait
```

Task 2: Solution



Task 2: Solution



Communication + Computation Overlap