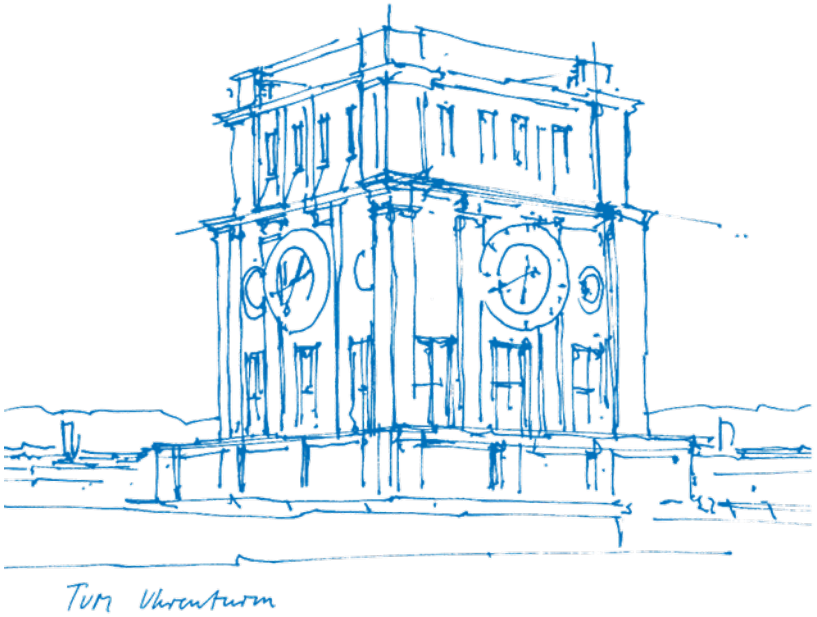
Parallel Programming Tutorial - Q/A on MPI

M.Sc. Amir Raoofy
Technical University of Munich
09. Juli 2018



Organization

Organization

- Speedup requirements for assignment 7 (SIMD) is relaxed to 2.5.
- Final Exam is on July 24th, 16:00 to 17:30 in MW 2001.
- **Date for repetition Exam might change from October 5th to October 12th.**
  - **Double check the final date in TUMOnline.**
- We will have no more Q/As with tutors.
- On Wednesday, we will have a dedicated Q/A session for exam preparation.
- We are processing the results of your submissions to our server.
  - Hopefully, we will publish the results by next week.

Hints for assignment 07, SIMD

Hints for assignment 07 - SIMD (Optional)

- Vectorization of a simple matrix matrix multiplication using intrinsics
  - In fact for obvious reasons, it is a matrix-transposed-matrix multiplication.
- Try vectorizing the most inner loop (k loop)
- Typically the vectorization is done on the most inner loop.
- You have to try vectorizing the inner-products.
- Look into our guest lecture by Dr. Michael Klehm. There you can find out how to do it.
- Look into Intel intrinsic guide as a reference for intrinsic instructions.

Solution to assignment 11 - profiling with mpiP

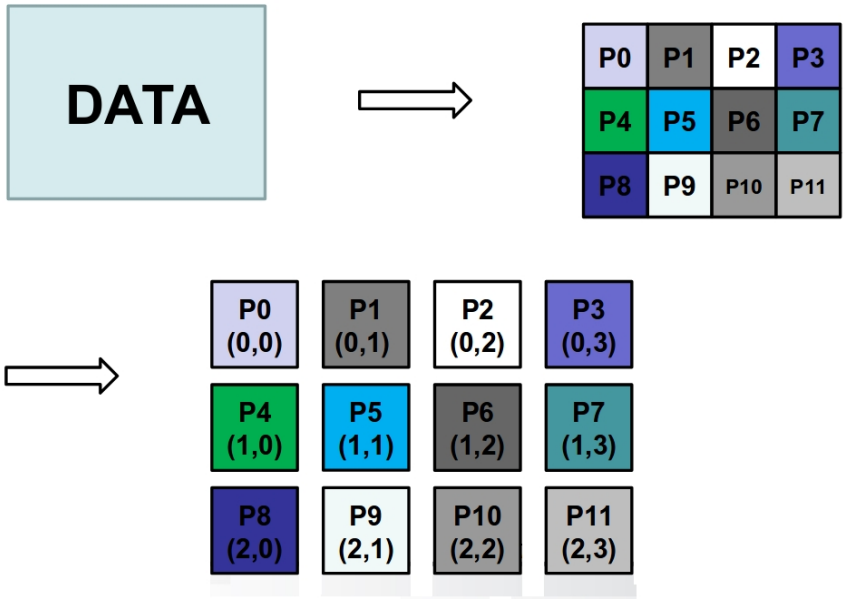
Solution to assignment 11 - profiling with mpiP

- You only need to modify Makefile in CoMD:
  - `OTHER_LIB = -L <path to shared object> -lmpiP -ldl -lm -lunwind`
  - You could have used **rpath** or **LD_LIBRARY_PATH** in case you had a problem with shared library.
- You could have used `LD_PRELOAD=<path to shared object>/libmpiP.so`.
- In case you could not resolve the source look up properly you could use:
  - **export MPIP=-k0** before the execution of the program.
  - This sets the depth of stack trace to zero
  - Look in here for more information: here

- You can find specific Callsites in your codes.
- Find the communication bottlenecks and target them for further optimizations.
- You can find statistics about MPI messages.

Solution to assignment 10

Assignment 10 - Virtual grid topology



Assignment 10 - setting up grid topology and domain decomposition

- We introduce a virtual grid topology.
- Each MPI process needs to know where in the grid it is located.
- Each MPI process needs which other tasks are its neighbor in the virtual grid.
- Each MPI process gets a small chunk of global heat problem according to its location on the virtual grid topology

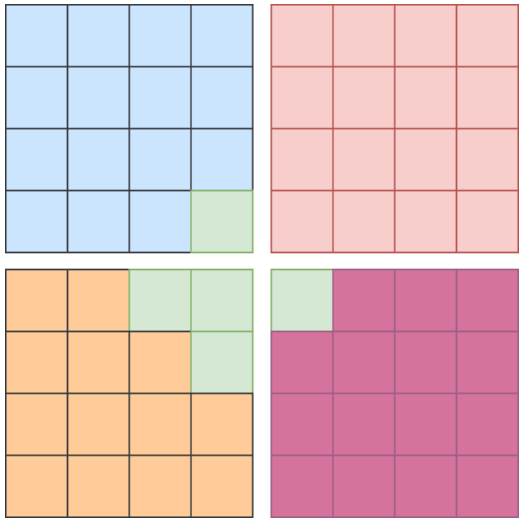
```
// determine my coordinates (x,y) -- r=x*a+y in the 2d grid topology of processors
int rx = rank % px;
int ry = rank / px;

// determine my four neighbors
int north = (ry-1)*px+rx; if(ry-1 < 0)   north = MPI_PROC_NULL;
int south = (ry+1)*px+rx; if(ry+1 >= py) south = MPI_PROC_NULL;
int west= ry*px+rx-1;     if(rx-1 < 0)   west = MPI_PROC_NULL;
int east = ry*px+rx+1;    if(rx+1 >= px) east = MPI_PROC_NULL;

// decompose the domain
int bx = n/px;               // block size in x
int by = n/py;               // block size in y
int offx = rx*bx;            // offset in x
int offy = ry*by;            // offset in y
```

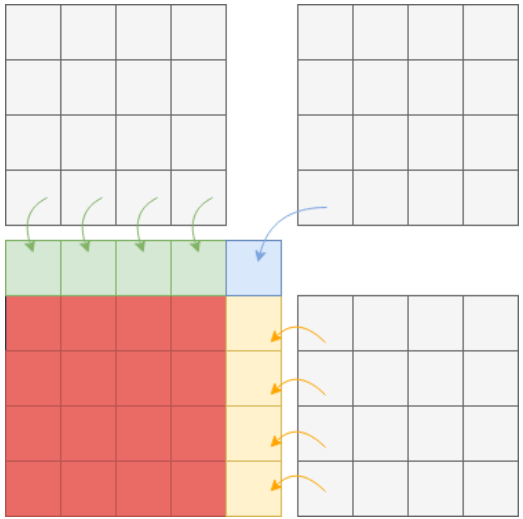
Assignment 10 - domain decomposition

- Why do we need additional halos?
- Where do we need communication?



Assignment 10 - halos

- Why do we need additional halos?
- Where do we need communication?



Assignment 10 - domain decomposition again

- Each process allocates memory for its chunk of problem + additional halo zones
- Since we use non-blocking communication, We need additional buffers for packing and unpacking MPI messages

```
// memory allocation in each MPI process
h_old = (double*)calloc(1,(bx+2)*(by+2)*sizeof(double)); // extended with halos of width 1
h_new = (double*)calloc(1,(bx+2)*(by+2)*sizeof(double)); // extended with halos of width 1
double *tmp;

// allocate communication buffers
// send buffers
double *sbufnorth = (double*)calloc(1,bx*sizeof(double));
double *sbufsouth = (double*)calloc(1,bx*sizeof(double));
double *sbufeast = (double*)calloc(1,by*sizeof(double));
double *sbufwest = (double*)calloc(1,by*sizeof(double));
// receive buffers
double *rbufnorth = (double*)calloc(1,bx*sizeof(double));
double *rbufsouth = (double*)calloc(1,bx*sizeof(double));
double *rbufeast = (double*)calloc(1,by*sizeof(double));
double *rbufwest = (double*)calloc(1,by*sizeof(double));
```

Assignment 10 - implementation of communications

- At each iteration, each MPI process:
  - Packs the sending messages, namely the borders, into send buffers.
  - Calls nonblocking MPI_ISends.
  - Calls nonblocking MPI_IRecvs.
  - Waits for non-blocking communication requests to finish.
  - Unpacks receiving messages from receiving buffers and copy them to halos.
  - update h_new array.

```
MPI_Request reqs[8];
for (int iter = 0; iter < niters; ++iter)
{
    // packing messages
    for(int i=0; i<bx; ++i) sbufnorth[i] = h_old[map(i+1,1,bx+2)];
    for(int i=0; i<bx; ++i) sbufsouth[i] = h_old[map(i+1,by,bx+2)];
    for(int i=0; i<by; ++i) sbufeast[i] = h_old[map(bx,i+1,bx+2)];
    for(int i=0; i<by; ++i) sbufwest[i] = h_old[map(1,i+1,bx+2)];
    // sending messages
    MPI_Isend(sbufnorth, bx, MPI_DOUBLE, north, 9, comm, &reqs[0]);
    MPI_Isend(sbufsouth, bx, MPI_DOUBLE, south, 9, comm, &reqs[1]);
    MPI_Isend(sbufeast, by, MPI_DOUBLE, east, 9, comm, &reqs[2]);
    MPI_Isend(sbufwest, by, MPI_DOUBLE, west, 9, comm, &reqs[3]);
```

Assignment 10 - implementation of communications (Cont.)

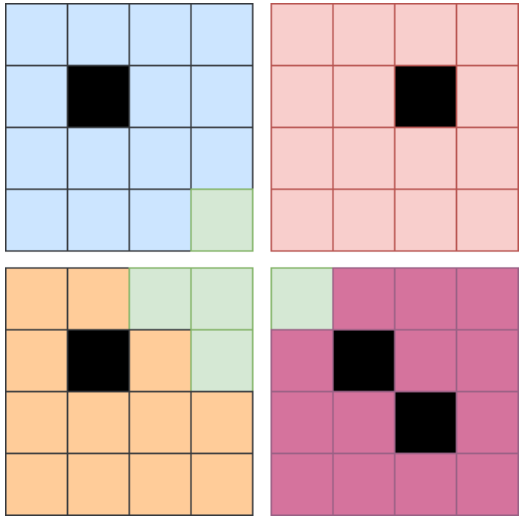
```
    // receiving messages
    MPI_Irecv(rbufnorth, bx, MPI_DOUBLE, north, 9, comm, &reqs[4]);
    MPI_Irecv(rbufsouth, bx, MPI_DOUBLE, south, 9, comm, &reqs[5]);
    MPI_Irecv(rbufeast, by, MPI_DOUBLE, east, 9, comm, &reqs[6]);
    MPI_Irecv(rbufwest, by, MPI_DOUBLE, west, 9, comm, &reqs[7]);

    // wait for non-blocking operations to finish
    MPI_Waitall(8, reqs, MPI_STATUSES_IGNORE);

    // unpack receiving messages into halos
    for(int i=0; i<bx; ++i) h_old[map(i+1,0,bx+2)] = rbufnorth[i];
    for(int i=0; i<bx; ++i) h_old[map(i+1,by+1,bx+2)] = rbufsouth[i];
    for(int i=0; i<by; ++i) h_old[map(bx+1,i+1,bx+2)] = rbufeast[i];
    for(int i=0; i<by; ++i) h_old[map(0,i+1,bx+2)] = rbufwest[i];

    // update domain
    for(int j=1; j<by+1; ++j)
        for(int i=1; i<bx+1; ++i)
            h_new[map(i, j ,bx+2)] = h_old[map(i, j, bx+2)] / 2.0 + (h_old[map(i - 1, j, bx+2)] \
                                   + h_old[map(i + 1, j, bx+2)] + h_old[map(i, j - 1, bx+2)] \
                                   + h_old[map(i, j + 1, bx+2)]) / 4.0 / 2.0;
    tmp=h_new; h_new=h_old; h_old=tmp;
}
```

Assignment 10 - what about the list of sources?



Assignment 10 - what about the list of sources? (Cont.)

- That is the tricky part. You need to parallelize the list of sources
- This means that each process needs to know which sources are located in its domain.
- We introduce a local array of sources for each process and fill it in with sources that are located in each process.
- We replace the local array with the global one in program so that each process only iterates over the local elements of the sources.

```
int locnsources=0;                                  // number of sources in my area
int locsources[nsources][2];                        // sources local to my rank
// loop over all sources to determine which sources are in my part
for (int i=0; i<nsources; ++i) {
    int locx = sources[i][0] - offx;
    int locy = sources[i][1] - offy;
    if(locx >= 0 && locx < bx && locy >= 0 && locy < by) {
        locsources[locnsources][0] = locx+1;        // offset by halo zone
        locsources[locnsources][1] = locy+1;        // offset by halo zone
        locnsources++;
    }
}
```

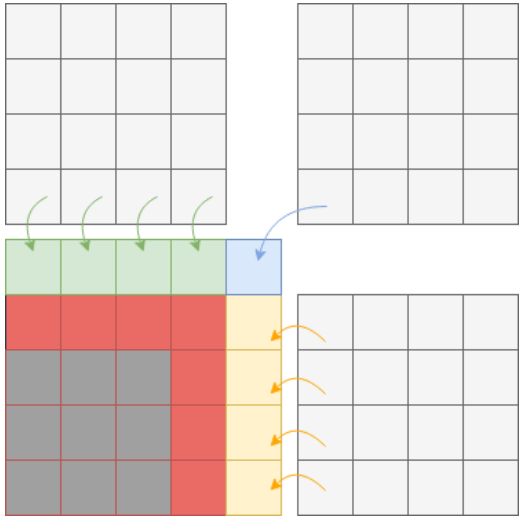
Assignment 10 - how can we further optimize the program?

- What is the use of non-blocking communication if we do not really use it?
- Why do we use non-blocking if we immediately call waits after the communications?!
- We should do as much as possible while MPI is executing the communications.
- The idea is to split the domain update iterations into two parts:
  - Borders right next to the halos where you need updated halos in each iteration
  - Inner parts of domain where you can already start updates without needing updated halos.
- Of course this re-orders the iterations and **might** we end up with different results as floating point operations are not associative. e.g., gauss seidel method.

- Code and illustration in next slides.

- further optimizations?
  - Shared-memory parallelization; assignment 12.
  - We can use one-sided communication for optimization of the communication.
  - We can use MPI neighbor collectives.

Assignment 10 - how can we further optimize the program? (Cont.)



Assignment 10 - how can we further optimize the program? (Cont.)

```
// update inner grid points
for(int j=2; j<by; ++j) {
    for(int i=2; i<bx; ++i) {
        // ...
    }
}

// communication has to be finished at this point
MPI_Waitall(8, reqs, MPI_STATUSES_IGNORE);

// update outer grid points
for(int j=1; j < by+1; j+=by-1) {
    for(int i=2; i<bx; ++i) { // north, south
        // ...
    }
}
for(int j=1; j < by+1; ++j) {
    for(int i=1; i<bx+1; i+=bx-1) { // east, west
        // ...
    }
}
```

Solution to assignment 12

Solution to assignment 12 - Hybrid MPI+OpenMP

- It is very easy. You just need to use OpenMP for parallelization in addition to MPI.
- The idea is to have OpenMP for shared-memory parallelization and MPI for distributed memory.
  - Although, you can already use MPI for shard memory for good.
- Also note the initialization of MPI in main:
  - `MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);`

```
for (int iter = 0; iter < niters; ++iter)
{
    // MPI stuff
    // ...

    #pragma omp parallel for num_threads(num_threads)
    for(int j=1; j<by+1; ++j)
        for(int i=1; i<bx+1; ++i)
            h_new[map(i, j ,bx+2)] = h_old[map(i, j, bx+2)] / 2.0 + (h_old[map(i - 1, j, bx+2)] + h_old[map(i +

    // the rest of stuff
    // ...
}
```