High Performance Computing for Science and Engineering I.

Exercise Set 4

Florian Mahlknecht

2019-11-21

Contents

1 2D Diffusion **1**

1.1 Diagnostics ouptput 3

1.2 Histogram 3

2 Bug Hunting **5**

3 Pipelining **7**

3.1 Intel Pipeline 7

3.2 Pipeline example from lecture 7

1 2D Diffusion

Even though it was not requested, Listing 1 shows the non-blocking MPI communication implementation in the `advance` routine.

fmahlknecht@student.ethz.ch
19-945-351

```
MPI_Request req[4];

if (m_procs > 1) {
  // *** start MPI part ***
  auto nextRank = (m_procs + m_rank + 1)%m_procs;
  auto prevRank = (m_procs + m_rank - 1)%m_procs;

  auto pSendLower = m_rho.data() + (1)*m_realN;
  auto pSendUpper = m_rho.data() + (m_localN)*m_realN;
  auto pSaveLower = m_rho.data() + (0)*m_realN;
  auto pSaveUpper = m_rho.data() + (m_localN+1)*m_realN;

  MPI_Irecv(pSaveUpper, m_realN, MPI_DOUBLE, nextRank,
            CommTags::Lower, MPI_COMM_WORLD, req);
  MPI_Irecv(pSaveLower, m_realN, MPI_DOUBLE, prevRank,
            CommTags::Upper, MPI_COMM_WORLD, req+1);
  MPI_Isend(pSendLower, m_realN, MPI_DOUBLE, prevRank,
            CommTags::Lower, MPI_COMM_WORLD, req+2);
  MPI_Isend(pSendUpper, m_realN, MPI_DOUBLE, nextRank,
            CommTags::Upper, MPI_COMM_WORLD, req+3);
}

auto applyStencil = [&](int i) {
  for (int j = 1; j <= m_N; ++j) {
    m_rho_tmp[i*m_realN + j] = m_rho[i*m_realN + j] + m_fac * (
          + m_rho[i*m_realN + (j+1)]
          + m_rho[i*m_realN + (j-1)]
          + m_rho[(i+1)*m_realN + j]
          + m_rho[(i-1)*m_realN + j]
          - 4.*m_rho[i*m_realN + j]
        );
  }
};

/* Central differences in space, forward Euler in time, Dirichlet BCs */
for (int i = 2; i < m_localN; ++i) {
  applyStencil(i);
}

if (m_procs > 1)
  MPI_Waitall(4,req,MPI_STATUSES_IGNORE);

applyStencil(1);
applyStencil(m_localN);

/* Use swap instead of rho_ = rho_tmp__. This is much more efficient,
           because it does not copy element by element, just replaces storage
           pointers. */
using std::swap;
swap(m_rho_tmp, m_rho);
```

Listing 1: Communication implementation

Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

1.1 Diagnostics ouptput

The diagnostics show the output given in fig. 1. As expected the density decreases over time as expected.

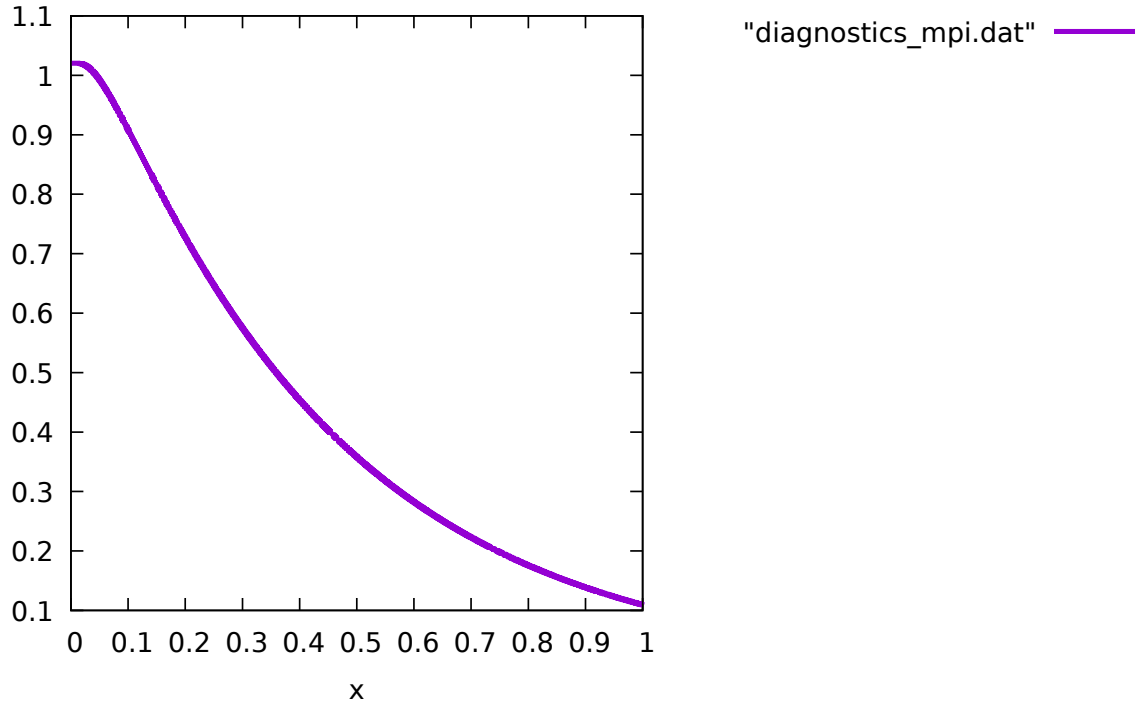

Figure 1: Diagnostics output

1.2 Histogram

The MPI related parts of the histogram computation are implemented as follows:

```
1  // *** start MPI part ***
2  // ...
3
4  /* compute local extrema... */
5
6  MPI_Allreduce(&lmin_rho, &min_rho, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
7  MPI_Allreduce(&lmax_rho, &max_rho, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
8
9  /* compute local bin counts ... */
10
11 // *** start MPI part ***
12 MPI_Reduce(&hist, &g_hist, M, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
13 // *** end MPI part ***
14 // *** end MPI part ***
```

Listing 2: Histogram MPI reductions implementation

After 0.5 s, i.e. with a Δt of 0.000 05 and 10000 iterations, the results shown in fig. 2 are obtained.

fmahlknecht@student.ethz.ch
19-945-351

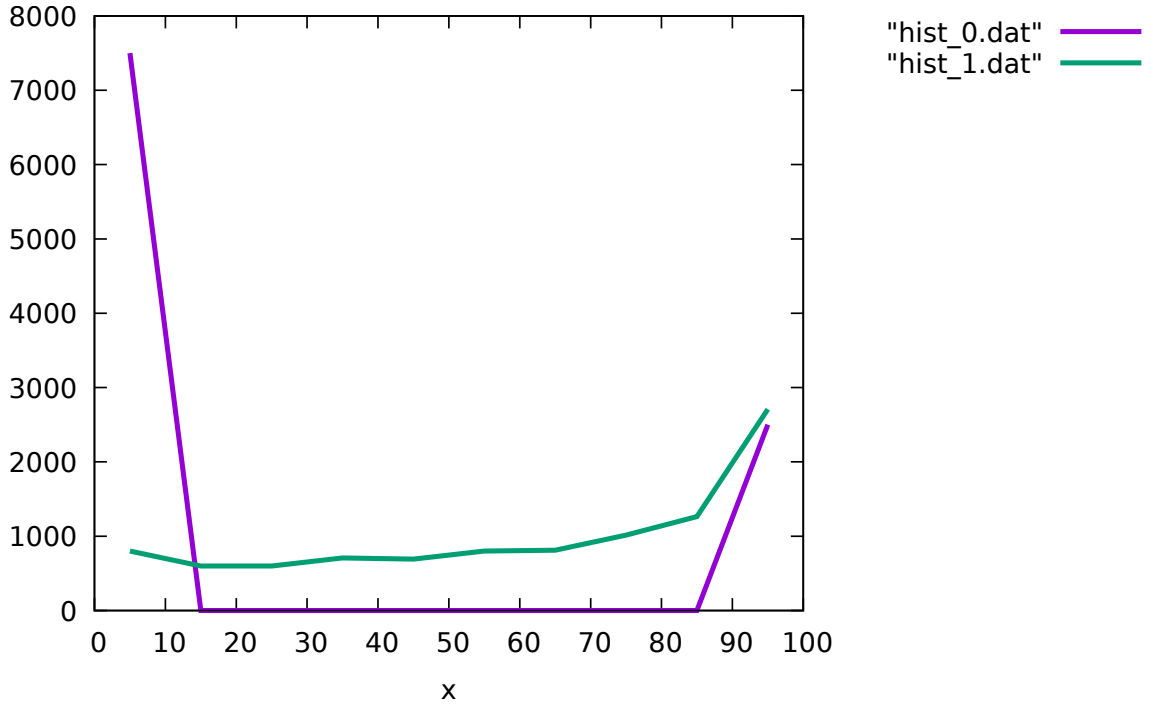

Figure 2: Histogram after 0.5 seconds

Note that this is a sort of normalized version of the histogram, to be able to compare them. The actual histogram for precise understanding is given in fig. 3.

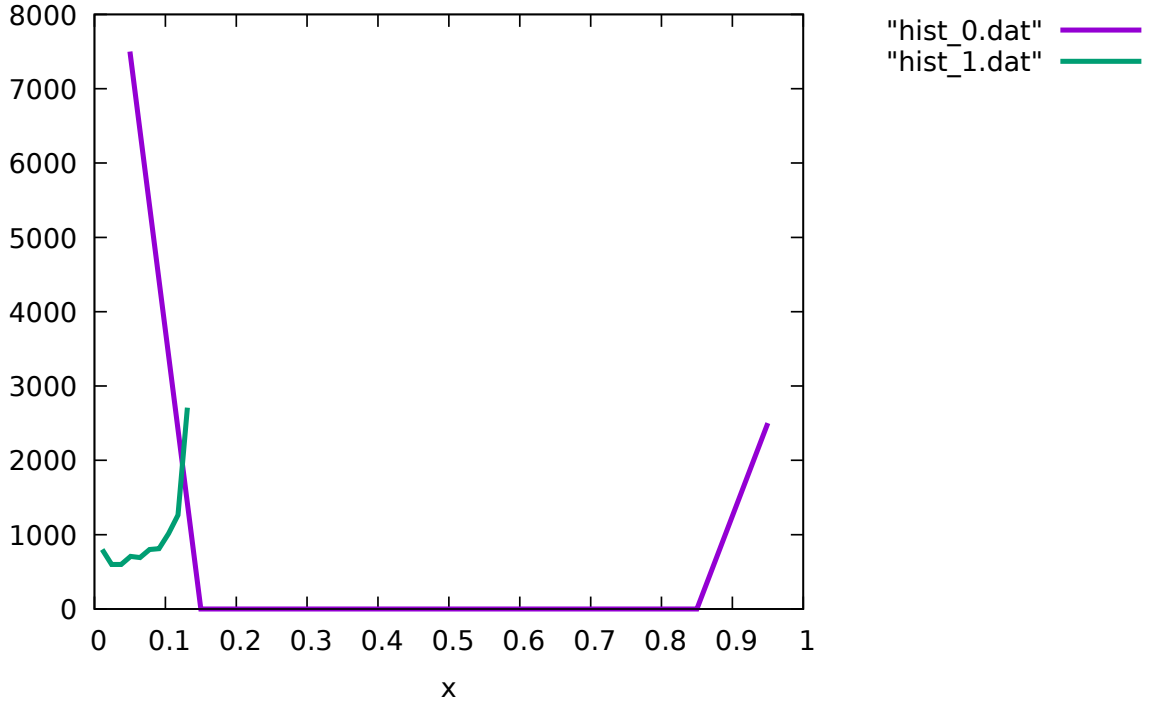

Figure 3: Unscaled histogram after 0.5 seconds

Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

2 Bug Hunting

```
const int N = 10000;
double result = new double[N];
//do a very computationally expensive calculation
//...

// write the result to a file
std::ofstreamfile("result.txt");

for (int i = 0; i < N; ++i){
  file<<result[i]<<std::endl;
}

delete[] result;
```

Listing 3: Code Snippet 1

In listing 3, under the assumption that the program requires MPI, we can say that:

1. `MPI_Init()` and `MPI_Finalize()` are missing
2. When writing to an output file, we need to make sure that either
   - the filename is unique
   - only the root process (e.g.) writes the file (i.e. `rank == 0`)
3. before accessing the filesystem resources, `MPI_Barrier` should be called to guarantee a synchronization of data before output

```
//only 2 ranks: 0,1
double important_value ;
//obtain the important value
//.. .
//exchange the value
if(rank == 0)
  MPI_Send(&important_value,1, MPI_DOUBLE,1,123, MPI_COMM_WORLD) ;
else
  MPI_Send(&important_value,1, MPI_DOUBLE,0,123, MPI_COMM_WORLD) ;

MPI_Recv(&important_value,1,MPI_INT,MPI_ANY_SOURCE,
        MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
// do other work
```

Listing 4: Code Snippet 2

In listing 4 we can spot immediately a potential deadlock. In most implementation this will not be visible, because one double value probably triggers a buffered send, but in any case this

fmahlknecht@student.ethz.ch
19-945-351

needs to be avoided. Either change the code to asynchronous communication or exchange the send / receive order for different ranks.

Furthermore the same buffer for sending and receiving is used, which must be avoided.

```
MPI_Init (&argc, &argv) ;
int rank,size;
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
int bval ;
if(0==rank )
{
  bval=rank ;
  MPI_Bcast(&bval,1,MPI_INT,0,MPI_COMM_WORLD) ;
}
else
{
  MPI_Statusstat ;
  MPI_Recv(&bval,1,MPI_INT,0,rank, MPI_COMM_WORLD, &stat) ;
}
cout << "[" << rank << "]" << bval <<endl ;
MPI_Finalize();
return 0;
```

Listing 5: Code Snippet 3

In listing 5 we can analyze the program for the case where there is only 1 rank started. The else branch is never executed and the broadcast goes well, since there are no other processes which need to read the value.

However, when there is more than just 1 rank, the code tries to receive a value by calling `MPI_Recv`. This is wrong and will never return, because broadcasting makes part of MPI's collective communication. This means that to read the broadcasting value, all processes are required to call `MPI_Bcast` (ideally) at the same time. So in this case, all ranks will be stuck, i.e. it results pending and will never complete.

Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351

3 Pipelining

3.1 Intel Pipeline

There are two ports which can execute a load (micro-) operation and 1 port which can execute a store (micro-) operation. This means that the ratio is $\frac{2}{1}$.

From the lecture slides we know that in [1] we can find an estimate of average of relative occurance of specific intel 80x86 instructions. For store and load they are:

Load 22 %

Store 12 %

So as we can see the ratio is approximately the same, i.e. $\approx \frac{2}{1}$ which motivates their design choice.

3.2 Pipeline example from lecture

Without a pipeline one instruction takes 15 ns.

The shortest possible cycle for the pipelining case is 6 ns.

With this pipeline executing N instructions while starting with an *empty* pipeline takes $((N-1)+5)\,T_{cycle}$, since we need to wait for the first instructions to fill up the pipeline. The first instruction is completed after 5 stages, and the remaining $(N-1)$ will take just the cycle time. So after all:

$$(N+4)\,6\,\text{ns} \tag{1}$$

However, this only holds if no pipelining hazard occurs, i.e. no pipeline stalls occur. Under the same assumption we get the following speedup w.r.t. the non-pipelined implementation:

$$s = \frac{N\,15\,\text{ns}}{(N+4)\,6\,\text{ns}} \tag{2}$$

$$\lim_{N\to\infty} s = \frac{15}{6} = 2.5 \tag{3}$$

References

[1] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

fmahlknecht@student.ethz.ch
19-945-351