Lab Course
Efficient Programming of Multicore Processors and Supercomputers

Report 5: Parallelization of Heat with MPI

Group 1: Jonas Mayer, Paul Preißner, Konrad Pröll

Fakultät für Informatik
Technische Universität München

June 29th 2017

General notes

- Task divisible into smaller subtasks with different problems
  - Parsing of inputfile
  - Initializing of array
  - Relaxation
  - Coarsening of output

Parsing of input file

- Different approaches:
  - "Root process" parses input, then broadcast
  - Every process parses input
- First approach is very easy for simple types
- Heatsrc is an array of structs!
- Second approach might lead to problems if every process reads at same time
- Idea: Use "token" to indicate whether the file is free or not

```
1   // check input file
2   int file_free=0;
3   {
4     if (myid==root)
5     file_free=1;
6     else
7     MPI_Recv(&file_free, 1, MPI_INT, myid-1, 1,
          MPI_COMM_WORLD, &status);
8     if (file_free==1)
9     {
10      // parse input
11    }
12    if (myid!=nprocs-1)
13    {
14      MPI_Send(&file_free, 1, MPI_INT, myid+1, 1,
            MPI_COMM_WORLD);
15    }
16  }
```

Initialization of the processes

- Consists of two subtasks
  - Calculation arraysizes and initializion of them
  - Initialization of heatsources
- Needed: Cartesian topology

```
1   if (((act_res)%(proc_x))==0)
2   (arraysize_x)=(len_x)=(act_res)/(proc_x);
3   else {
4     if ((col)==((proc_x)-1))
5     {
6       //last processor in dim x
7       (arraysize_x)=(act_res/proc_x)+1;
8       (len_x)=(act_res)%(arraysize_x);
9     } else {
10      (arraysize_x)=(len_x)=((act_res)/(proc_x))+1;
11    }
12  }
13  if (((act_res)%(proc_y))==0)
14  (arraysize_y)=(len_y)=(act_res)/(proc_y);
15  else {
16    if ((row)==((proc_y)-1))
17    {
18      //last processor in dim y
19      (arraysize_y)=((act_res)/(proc_y))+1;
20      (len_y)=(act_res)%(arraysize_y);
21    } else {
22      (arraysize_y)=(len_y)=((act_res)/(proc_y))+1;
23    }
24  }
```

- Check if border of array is actually border for global grid
- Adjust distance to array's position in global grid

```
1        int c_x=(col)*(arraysize_x);
2        int c_y=(row)*(arraysize_y);
3        for( i=0; i<numsrcs; i++ )
4        {
5          // top row
6          if ((row)==0)
7          {
8            for( j=0; j<spx; j++ ) //was np
9            {
10             dist = sqrt (pow((double)(j+c_x)/(double)(np-1)
                   -
11               heatsrcs[i].posx, 2)+
12               pow(heatsrcs[i].posy, 2));
13               if( dist <= heatsrcs[i].range )
14               {
15                 (u)[j] +=
16                 (heatsrcs[i].range-dist) /
17                 heatsrcs[i].range *
18                 heatsrcs[i].temp;
19               }
20           }
21         }
22         //other borders
```

Notes

- Very easy to mix up x and y
- Incredibly difficult to debug just looking at the residual
- Work with a small arraysize and print the arrays out!

Parallelization of the jacobi relaxation

- Three tasks: Reduction of global residual, communication of borders and adjustment of calculation to different arraysizes and to interleave communication and calculcation
- First task is pretty simple

```
1    MPI_Allreduce(&residual, &globresid, 1, MPI_DOUBLE,
         MPI_SUM, MPI_COMM_WORLD);
```

- For communication between processes, we need to determine neighbors

```
1       int north, south, west, east;
2       MPI_Cart_shift(comm_2d, 1, 1, &north, &south);
3       MPI_Cart_shift(comm_2d, 0, 1, &west, &east);
```

- We also need types for communication, as the layout of columns and lines in the memory vary

```
1   MPI_Type_contiguous(npx-2, MPI_DOUBLE,
        &north_south_type);
2   MPI_Type_vector(npy-2, 1, npx, MPI_DOUBLE,
        &east_west_type);
```

```
1    MPI_Request reqs[8];
2    MPI_Isend(&u[1+sizex] , 1, *north_south_type, north, 9,
         comm_2d, &reqs[0]);
3    MPI_Isend(&u[1+(sizey-2)*sizex] , 1, *north_south_type,
         south, 9, comm_2d, &reqs[1]);
4    MPI_Irecv(&u[1], 1, *north_south_type, north, 9,
         comm_2d, &reqs[2]);
5    MPI_Irecv(&u[1+(sizey-1)*sizex], 1, *north_south_type,
         south, 9, comm_2d, &reqs[3]);
6    MPI_Isend(&u[1+sizex], 1, *east_west_type, west, 9,
         comm_2d, &reqs[4]);
7    MPI_Isend(&u[2*sizex-2], 1, *east_west_type, east, 9,
         comm_2d, &reqs[5]);
8    MPI_Irecv(&u[sizex], 1, *east_west_type, west, 9,
         comm_2d, &reqs[7]);
9    MPI_Irecv(&u[2*sizex-1], 1, *east_west_type, east, 9,
         comm_2d, &reqs[6]);
```

Interleaving of calculation and communication

- Only the outermost layer of calculated gridpoints depends on values from other processes
- Start communication, calculate all but the outermost layer, then synchronize and calculate the outermost layer

```
1     // calculate inner points, which do not depend on
          borders
2     for( i=2; i<len_y+0; i++ ) {
3       for( j=2; j<len_x+0; j++ )
4       {
5         // Calculate new values
6       }
7     }
8     MPI_Waitall(8, reqs, MPI_STATUS_IGNORE);
9     for (i=1; i<len_x+1; i++)
10    {
11      // top row
12      unew = 0.25 *
            (u[i]+u[sizex-1+i]+u[sizex+1+i]+u[2*sizex+i]);
13      diff = unew - u[sizex+i];
14      utmp [sizex+i] = unew;
15      sum+=diff*diff;
16      // .. bottom row,
17    }
18    for (i=2; i<len_y; i++)
19    {
20      //leftmost and rightmost column
21    }
```

Coarsening

- State: Every process has its own, potentially very large array
- Goal: Writeout the heatdistribution of the entire body in a reasonable resolution uvis
- Obviously, gathering the whole array is no option
- In general, there is MPI_IO to writeout distributed array, but due to calculation of pixmap, this might be pretty difficult

- Coarse al local arrays to uvis/arraysize and gather those arrays
- One root process then writes the entire picture
- Issues:
  - Padded arrays: We stretch those to size of normal arrays.
  - Also, resolution might not be divisible through number of processes in dimension. We just increase the resolution in those cases
- Might lead to wrong output for very small resolutions, but works very fine for reasonable sizes
- We need to define a MPI_Datatype, so that receiving process knows datalayout of global uvis

```
1     MPI_Datatype subarray, subarray_resized;
2     int sizes[2];
3     sizes [0] = param.visresgloby;
4     sizes [1] = param.visresglobx;
5     int subsizes[2];
6     subsizes [0] = param.visresy;
7     subsizes [1] = param.visresx;
8     int starts[2]= {0,0};
9     int order;
10    int* displs=(int*) malloc (nprocs*sizeof(int));
11    int* counts=(int*) malloc (nprocs*sizeof(int));
12    int cords[2];
```

```
1     for (a=0; a<nprocs; a++)
2     {
3       counts[a]=1;
4       MPI_Cart_coords(comm_2d, a, 2, cords);
5       displs[a]=cords[0]*param.visresx+cords[1]*param.visresy*para
6     }
7     MPI_Type_create_subarray(2, sizes, subsizes, starts,
          MPI_ORDER_C, MPI_DOUBLE, &subarray);
8     MPI_Type_commit(&subarray);
9     MPI_Type_create_resized(subarray, 0, 1*sizeof(double),
          &subarray_resized);
10    MPI_Type_commit(&subarray_resized);
11    MPI_Gatherv(uloc, param.visresx*param.visresy,
          MPI_DOUBLE, param.uvis, counts, displs,
          subarray_resized, root, comm_2d);
```

Measurements

- Sadly, due to SuperMUC maintenance we could not run a lot of tests on our implementation
- Speedup fpr square topologies looks like it is limited by memory bandwidth

Measurement	Floprate	Speedup
Sequential (Baseline)	3,896.2 MFlop/s	1
Auto parallelization 8 threads	13,208.1 MFlop/s	3.39
OMP 16 Threads	22,247.3 MFlop/s	5.71
MPI 64 processes, blocking	74,064.5 MFlop/s	19.01
MPI 64 processes, nonblocking	80,339.4 MFlop/s	20.62

Thank you for your attention!