Lab exercises (1.2)

After building a parallel code you now will perform some 'experiments' with it. The main goal of these experiments is to improve performance. This can be done in various ways, such as using better numerical algorithms, minimizing time spent on communication, and adjusting the domain decomposition or partitioning. It is important to know how much time the program spends in the various phases and how this time depends on the different parameters such as the number of processes and problem size.

1.2 Exercises, modifications, and performance aspects

This exercise requires you to report on several experiments. Use the following symbols and notation in your report.

n: the number of iterations
g: gridsize
t: time needed in seconds
pt: processor topology in form pxy, where
p: number of processors used
x: number of processors in x-direction
y: number of processors in y-direction
For example, 414 means 4 processors in a 1×4 topology.

After completing the previous exercises, you should now have a parallel code that can easily be modified or transformed into another variant of the Poisson solver.

1.2.1

We start with an algorithmic improvement. It has nothing to do with parallelization. Change the code such that it also uses beside red-black iteration over-relaxation (SOR). Hereto the iterations are rewritten as follows

$$c_{i,j}^n = \frac{\phi_{i-1,j}^n + \phi_{i,j-1}^n + \phi_{i,j+1}^n + \phi_{i+1,j}^n + h^2 S_{i,j}}{4} - \phi_{i,j}^n \qquad (1)$$

where $c_{i,j}^n$ is the change in the value of the grid point (i,j) after the n^{th} red-black iteration? Updating the grid points is rewritten as

$$\phi_{i,j}^{n+1} = \phi_{i,j}^n + \omega c_{i,j}^n \qquad (2)$$

where ω is a relaxation parameter. Depending on the size of the problem ω has a value close to 2 for the fastest convergence. For the test problem, the number of iterations can be reduced by about a factor of 10.

1.2.2

Empirically determine the optimal value for w and the required number of iterations for the test problem (gridsize 100, 4 processors, 4×1 topology). Try strategically chosen values between 1.90 and 1.99 (5 runs maximal). There exist algorithms like the CG method that converge about equally fast and do not need a parameter ω that depends on the problem size.

1.2.3

Now that you have a faster code it is easier to investigate its scaling behaviour. Investigate whether the performance differs if one makes slices only in the x-direction or only in the y-direction. Do this with 4 processes in the following configurations: 4×1 and 2×2. For both configurations use problems with the following numbers of gridpoints: 200×200, 400×400, 800×800. (or as much as is feasible within at most 3 minutes of computing time per run). Use $\omega = 1.95$ for this experiment, and it is not necessary to iterate until convergence is reached. It is more practical to perform a fixed number of iterations to determine the time needed per iteration. The time *t(n)* needed for *n* iterations can be parametrized as

$$t(n) = \alpha + \beta n \tag{3}$$

where α and β are some constants. Both α and β depend on the problem size, and possibly also on the way the domain is partitioned. Adjust the file input.dat if you want to change the problem size and/or the maximum number of iterations.

1.2.4

How would you divide the domain in the x and y-direction: 16×1, 8×2, 4×4, 2×8 or 1×16? Hence determine α and β also in this case. Indicate this without actually running the experiments.

1.2.5

The number of iterations that are needed to reach convergence also depends on the problem size. Determine this number of iterations for various problem sizes. Use the following gridsizes 200×200, 400×400, 800×800 or larger (e.g., 1000×1000). Only adjust the problem size in input.dat, not the stopping criterion. How do you expect this number of iterations will increase when the problem size increases?

1.2.6

Monitor, for the 800×800 situation, the behaviour of the error as a function of the iteration number.

1.2.7 (optional)

Since you probably see a trend in the error as a function of the iteration count, you can also easily decide whether the problem will converge within the next few iterations. Adjust the code so that the global error is calculated and communicated only every 10 iterations. This way, the collective communication with MPI_Allreduce can be significantly reduced. The price is that you may perform a few (at most 9) iterations more than strictly necessary. Can you gain performance by evaluating the global error only every 10 (or 100) iterations? In other words, how much time is involved in obtaining the global error?

1.2.8

In another attempt to reduce communication, you can simply perform more than one red/black sweep

between two border exchanges. By making this modification, the numerical algorithm is changed. Determine the number of iterations and the time it takes for the test problem to converge to the desired accuracy. Perform simulations where the number of sweeps between two communication steps with Exchange_Borders is 1, 2, 3, and so on. A sweep over 'red' points is always followed by a sweep over 'black' points.

1.2.9

In Do_Step the sweeps through the lattice are over all grid points, whereas only half of them are updated. Instead of changing the numerical algorithm, you can try to optimize it by better exploiting this knowledge. Investigate whether performance improves if the inner for loop in Do_Step is modified so that the parity check in the if-statement is no longer needed.

1.2.10 (optional)

Measure the amount of time spent within Exchange_Borders as a function of problem size and as the number of processes. For which number of processes and/or problem size do you expect this time to be about the same as the time spent doing useful calculations?

1.2.11

Though the time spent in Exchange_Borders is small in the current situation, it can easily become an important fraction of the total time if more processes are active. Hence, it is relevant to investigate the behaviour of Exchange_Borders in more detail. Determine how much time spent within Exchange_Borders can be attributed to latency or overhead and measure the bandwidth for point-to-point communication. You also need to calculate the total amount of data that is communicated. Conduct your experiments with varying problem sizes, using the same configurations and grid sizes as in item 1.2.3.

1.2.12

In Exchange Borders, each time one communicates twice as many data points as necessary, since in each sweep only half the grid points along a boundary are updated. Hence, performance may increase if the amount of data communicated is reduced by a factor of two. However, this 'optimization' is rather tricky to implement, especially with the current data structure where all red and black grid points are stored in the same data structure.

Analyze the various situations that may occur. Consider how

- the address of the first point to exchange,
- the number of points to exchange, and
- the number of points (in the data structure) in between grid points that have to be exchanged,

depends on the phase in the calculation (just before a red-sweep or a black-sweep), the size of the subdomain, the border to which the grid points belong, and the color of the grid point &phi[0][]0]. Is it worth the effort to minimize the communication load and increase the complexity of the code?

1.2.13 (Optional)

Change the code to use the Conjugate Gradient Method instead of Jacobi or SOR. Since the emphasis is not on numerical algorithms, we will provide an algorithm here. It replaces the Do_Step routine. Note that the function prototype has changed.

```
void Do_Step()
{
  int x, y;
  double a, g, global_pdotv, pdotv, global_new_rdotr, new_rdotr;

  /* Calculate "v" in the interior of my grid (matrix-vector
  multiply) */ for (x = 1; x < dim[X_DIR] - 1; x++)
    for (y = 1; y < dim[Y_DIR] - 1; y++)
    {
      vCG[x][y] = pCG[x][y];
      if (source[x][y] != 1)      /* only if point is not fixed */
        vCG[x][y] -=  0.25 *(pCG[x + 1][y] + pCG[x – 1][y] +
                    pCG[x][y + 1] + pCG[x][y - 1]) ;
    }

  pdotv = 0;
  for (x = 1; x < dim[X_DIR] - 1; x++)
    for (y = 1; y < dim[Y_DIR] - 1; y++)
      pdotv += pCG[x][y] * vCG[x][y];

  MPI_Allreduce(&pdotv, &global_pdotv, 1, MPI_DOUBLE,
                MPI_SUM, grid_comm);

  a   = global_residue / global_pdotv;

  for (x = 1; x < dim[X_DIR] - 1; x++)
    for (y = 1; y < dim[Y_DIR] - 1; y++)
      phi[x][y] += a * pCG[x][y];

  for (x = 1; x < dim[X_DIR] - 1; x++)
    for (y = 1; y < dim[Y_DIR] - 1; y++)
      rCG[x][y] -= a * vCG[x][y];

  new_rdotr = 0;
  for (x = 1; x < dim[X_DIR] - 1; x++)
    for (y = 1; y < dim[Y_DIR] - 1; y++)
      new_rdotr += rCG[x][y] * rCG[x][y];

  MPI_Allreduce(&new_rdotr, &global_new_rdotr, 1, MPI_DOUBLE,
                MPI_SUM, grid_comm);

  g = global_new_rdotr / global_residue;
  global_residue = global_new_rdotr;

  for (x = 1; x < dim[X_DIR] - 1; x++)
    for (y = 1; y < dim[Y_DIR] - 1; y++)
      pCG[x][y] = rCG[x][y] + g * pCG[x][y];
}
```

After each iteration global_residue, as calculated within Do_Step is used as a stop criterion. The following global variables

```
double **pCG, **rCG, **vCG;
double global_residue;
```

are needed and memory is allocated for the three arrays in InitCG, similar to the allocation of memory for phi in Setup_Grid. The heart of the Solve routine now becomes

```
InitCG()
while (global_residue > precision_goal && count < max_iter)
{
  Exchange_Borders();
  Do_Step();
  count++;
}
```

The variable global_delta as used in the previous codes is replaced by global_residue since it has a different meaning. Moreover, in Exchange_Borders, not phi, but pCG has to be exchanged, see the first do in Do_Step. Finally, the routine InitCG() is needed to properly start the conjugate gradient algorithm.

In the CG algorithm, all processes have to communicate with each other to evaluate the two global dot products. It also requires more memory than the SOR or red-black Gauss-Seidel algorithms. However, the code issimplified since there is no longer a need for a relaxation parameter ω. Additionally, all bookkeeping to determine the parity of each point in each subdomain is now obsolete. Compare the time needed per iteration for the test problem with that of the best previous algorithm. As a first check for the correctness of your implementation, note that the required number of iterations for the test problem is 125.

```
void InitCG()
{
  int x,  y;
   double rdotr=0;

   /* allocate memory for CG arrays*/
   pCG = malloc(dim[X_DIR] * sizeof(*pCG));
   pCG[0] = malloc(dim[X_DIR] * dim[Y_DIR] * sizeof(**pCG));
   for (x = 1; x < dim[X_DIR]; x++) pCG[x] = pCG[0] + x * dim[Y_DIR];

   rCG = malloc(dim[X_DIR] * sizeof(*rCG));
   rCG[0] = malloc(dim[X_DIR] * dim[Y_DIR] * sizeof(**rCG));
   for (x = 1; x < dim[X_DIR]; x++) rCG[x] = rCG[0] + x * dim[Y_DIR];

   vCG = malloc(dim[X_DIR] * sizeof(*vCG));
   vCG[0] = malloc(dim[X_DIR] * dim[Y_DIR] * sizeof(**vCG));
   for (x = 1; x < dim[X_DIR]; x++) vCG[x] = vCG[0] + x * dim[Y_DIR];

   /* initiate rCG and pCG */
   for (x = 1; x <  dim[X_DIR]  -  1;  x++)
     for (y = 1; y < dim[Y_DIR] - 1; y++)
     {
        rCG[x][y] = 0;
        if (source[x][y] != 1)
           rCG[x][y] = 0.25 *(phi[x + 1][y] + phi[x - 1][y] +
                              phi[x][y + 1] + phi[x][y - 1]) - phi[x][y];
       pCG[x][y] = rCG[x][y];
        rdotr += rCG[x][y]*rCG[x][y];
     }

   /*  obtain  the  global_residue  also  for  the  initial  phi    */
   MPI_Allreduce(&rdotr, &global_residue, 1, MPI_DOUBLE, MPI_SUM, grid_comm);
}
```