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 on a variety of ways, i.e., use better numerical algorithms, minimize time spent on communication, adjust the domain decomposition or partitioning. This implies that it is important to know how much time the program spends in the various phases, and how this time depends on the various parameters like number of processes and problem size.

1.2 Exercises, modifications and performance aspects

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

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.

You should, after having completed the previous exercises, 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 uses besides red-black iteration also 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 value of the grid point (i,j) after the nth 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 determine 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 also algorithms like the CG method that converge about equally fast and do not need a parameter w 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 there is difference in performance if one make 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 grids with the following numbers of gridpoints: 200×200, 400×400, 800×800. (or as much as is feasible within at most 1 minute of computing time per run). Use ω= 1.95 For this experiment it is not necessary to iterate until convergence is reached. It is more practical to perform a fixed number of iterations to find out the time needed per iteration. The time $t(n)$ needed for n iterations can be parametrized as

$$t(n)=\alpha+\beta n \qquad (3)$$

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 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 is needed in order 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 that this number of iterations increases for increasing problem size?

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 or will not convergence within the next few iterations. Adjust the code such that the global error is calculated and communicated only every 10 iterations. The collective communication with MPI_Allreduce can thus easily be reduced with a large factor. The computational price one has to pay is that one may perform a few (at most 9) iterations more than is 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 1 red/black sweep in between two border exchanges. By such a modification the numerical algorithm is changed. Determine the number of iterations, and the time it takes before the test problem has converged to the desired accuracy. Performs simulations where the number of sweeps in between 2 communication steps with Exchange_Borders is 1, 2, 3, ... 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, one can try to optimize it by (better) exploiting this knowledge. Investigate whether performance goes up if the inner for loop in Do_Step is modified such 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 a function of the number of processes. For which number of processes and/or problem size do you expect that this time will 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 situations, 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. How much of the time spent within Exchange_Borders can be marked as latency or overhead and how large is the bandwidth for point-to-point communication. Hence, you have to calculate also the total amount of data that is communicated. Do your experiments with varying problem sizes. Use the same configurations and gridsizes 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 that is communicated is reduced with 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.

Analyse 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 to grid points that have to be exchanged,

depends on the phase in the calculation (just before a red-seep or a black sweep), the size of the subdomain, the border to which the grid points belongs 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 such that it uses the Conjugate Gradient Method instead of Jacobi or SOR. Because the emphasis is not on numerical algorithms, we will give 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 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.

```
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);
}
```

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