Verena Horak
3rd Homework
HPSC 2004

Exercise 3.2:
Implement a two dimensional finite difference algorithm on a square domain of dimension $n \times n = N$ for 5 points stencil using MPI. Measure performance on one to four parallel computers, and use performance models to explain your results. Plot execution time as a function of the number of points (N) and as a function of the number of processors (P) for N up to 100×100 and P up to 4 processors and for 10^4 time steps.

As an example for five points stencil, we will implement an iterative solution for the Laplace Equation $u_{xx} + u_{yy} = 0$. We will set boundary conditions to be one at the left and right margin of the $n \times n$ square domain and zero otherwise. We will denote the value of each spatial mesh point (i, j) after iteration k by $u^k_{i,j}$ $\forall\ 0 \leq i, j \leq n-1$. Values will be initialized by boundary conditions and zero for all inner mesh points.
From lecture, we know that an iterative solution for the Laplace Equation can be obtained by the formula

$$u^{k+1}_{i,j} = \frac{u^k_{i-1,j} + u^k_{i+1,j} + u^k_{i,j-1} + u^k_{i,j+1}}{4} \quad \forall\ 1 \leq i, j \leq n-2\ .$$

Note that values need only to be updated for the inner mesh points as values at the margin remain constant due to boundary conditions.
We can parallelize this algorithm by data domain distribution among multiple processors. Thus, every process will be responsible for a submatrix of the global square domain. For P processors, the submatrices would have dimension $\frac{n}{m} + 2 \times n$, as some additional place has to be provided in order to store values from neighbouring processes that are needed for calculating the updated values $u^{k+1}_{i,j}$. Note that the first or the last row of this submatrix may remain unused if the process has rank 0 or rank $P-1$. Values at the boundary of each process' domain have to be exchanged with neighbouring processes after every iteration step.
This method is quite a bad approach towards a numerically good approximation to the solution of the Laplace Equation, as it is slow to converge. As our focus in this example is put on the possibility of parallelizing algorithms and the effect on execution time, we don't care any further about this fact.
The implementation of this algorithm might look like this:

```
int n, P     // define size of data domain and number of processors
int m        // total number of processes participating in computation
int rank     // number of local process;
```

```
Start parallel computation with m processes;

// the number of rows in the submatrix of the global square domain
// is in general (if P is not a divisor of n) given by
// ( (n + n mod P)/P + 2);
// for process number P-1, some rows may remain unused
double[(n + n mod P)/P + 2, n] u  // submatrix of global square domain
double[(n + n mod P)/P + 2, n] uold

// Initialize submatrices u and uold
// with boundary conditions and initial values
for (i = 0; i <= (n + n mod P)/P+1; i++) {
  for (j = 0; j <= n-1; j++) {
    if (j==0 or j==n-1)
      {u[i, j] = 1; uold[i, j] = 1;}
    else
      {u[i, j] = 0; uold[i, j] = 0;}
  }
}

// Start iteration for 10000 time steps
for (k = 1; k <= 10000; k++) {
  for (i=1; i <= (n + n mod P)/P; i++) {
    for (j=1; j <= n-2; j++) {
      u[i, j] = (uold[i-1, j] + uold[i+1, j] +
                 uold[i, j-1] + uold[i, j+1]) / 4;
    }
  }
  // Exchange data with neighbouring processes
  If (rank > 0) {
    Send row u[1] to process number rank-1;
    Receive data from process number rank-1 and write to u[0];
  }
  If (rank < m-1) {
    Send row u[(n + n mod P)/P] to process number rank+1;
    Receive data from process number rank+1 and write to
      u[(n + n mod P)/P + 1];
  }
  uold = u;
}

// Local submatrix u represents approximate solution to Laplace
// Equation for data domain that was assigned to the process;
// if necessary, all data may be combined to get complete solution
// in a single process
```

Testing the program with up to four parallel computers, the following values for execution time (seconds) were measured:

	$N = 12 \times 12$	24×24	48×48	96×96	300×300
$P = 1$	0.1	0.3	1.1	4.2	41.1
$P = 2$	0.9	1.1	1.8	4.1	25.2
$P = 3$	1.3	1.7	2.6	5.2	20.7
$P = 4$	1.4	1.8	2.8	5.2	18.0

The table above may support two conclusions:

- For small values of N, that is for few data to calculate, the time needed for communication and synchronisation among multiple processes causes the parallel algorithm to perform worse than computation on a single processor.
- For larger values of N, parallelizing the algorithm leads to improved performance. The amount of time saved by working in parallel on different data domains grows, whereas the effort for communication becomes neglible. In other words, the overhead for communication remains nearly constant for growing N, so that the time saved by parallel execution has a considerable impact.

The source code of the program used is cited below:

```
#include <stdio.h>
#include <math.h>
#include "mpi.h"

    int main(int argc, char **argv){
      int        rank, value, size, i, j, count, n;
      MPI_Status status;
      double     atime, etime;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank( MPI_COMM_WORLD, &rank );
      MPI_Comm_size( MPI_COMM_WORLD, &size );
      n=300;

      double     uold[(n+n%size)/size+2][n];
      double     unew[(n+n%size)/size+2][n];
```

```
   for (i = 0; i <=(n+n%size)/size + 1; i++){
      for (j = 0; j <= n - 1; j++){
         if (j==0 || j==(n-1)){
            unew[i][j] = 1; uold[i][j] = 1;
         }
         else{
            unew[i][j] = 0; uold[i][j] = 0;
         }
      }
   }

   atime = MPI_Wtime();
   for (count = 1; count <= 10000; count++){
      for (i = 1; i <= (n+n%size)/size; i++){
         for (j = 1; j <= n-2; j++) {
            unew[i][j] = (uold[i][j+1] + uold[i][j-1] +
                          uold[i+1][j] + uold[i-1][j]) / 4.0;
         }
      }

      if (rank < size - 1){
         MPI_Send(unew[(n+n%size)/size], n, MPI_DOUBLE, rank+1,
         0, MPI_COMM_WORLD );
      }
      if (rank > 0){
         MPI_Send(unew[1], n, MPI_DOUBLE, rank-1, 1, MPI_COMM_WORLD );
         MPI_Recv(unew[0], n, MPI_DOUBLE, rank-1, 0, MPI_COMM_WORLD,
         &status );
      }
      if (rank < size - 1){
         MPI_Recv(unew[(n+n%size)/size+1], n, MPI_DOUBLE, rank+1, 1,
         MPI_COMM_WORLD, &status );
      }

      for (i = 0; i <= (n+n%size)/size+1; i++){
         for (j = 0; j <= n-1; j++) {
            uold[i][j]=unew[i][j];
         }
      }
   }
   etime = MPI_Wtime();
   if (rank == 0)
      printf("Time needed for %d Points: %0.1f\n",n*n,etime-atime);
   MPI_Finalize( );
   return 0;
}
```