ANIRUDH SETH
MAGNUS PIERRAU

# Homework Assignment 2

Parallel Computations for Large- Scale Problems

2020-02-26

1. (a) Algorithm 1 below shows the psuedo code for a one-to-all broadcasting operation. The function takes three arguments: the integer $D$, where $D$ is the logarithm of the number of processors; $s$, the index of the processor sending the message (the source); and $m$, the message to be sent.

   This operation handles the general case where any processor $s$ sends a message to all other processors. This is done by the help of a *virtual ID* (VID), in where the "real" processor indices are mapped to a virtual index, in which the source processor has virtual index $0$.

   The algorithm uses recursive doubling and thus finishes in $D$ steps.

   The variable `mask` initiated in line 4, help us discern which processor should be active (send or receive) during which iteration. The `mask` variable is initialized to $2^D - 1$ and for each iteration $i = D - 1 : 0$, the $i$th bit is flipped from $1$ to $0$. This helps us identify processors which are multiples of $2^i$ in each iteration. These processors will take part in sending and receiving in that iteration. For example, if $D = 3$, and $s = 0$, the first step will see processor $0$ send a message $m$ to processor $4$ ($0$ is not a multiple of $4$, but is the source and communicates in each step). In the subsequent step processors $0, 2, 4$ and $6$ will be active, all multiples of $2$.

   After determining which processors are to be active this iteration (line 7), we determine which processors will be sending and receiving messages respectively (line 8).

   Lines 9 and 13 determine the virtual index of the two processors to communicate and are translated into real indices in lines 10 and 14 respectively. The index of the processor to communicate with is found by bit flipping the $i$th bit in the VID.

**Algorithm 1** Psuedocode for general One-to-all Broadcasting operation

**Require:** $D$ ($2^D$ processors), $s$ (source of message), $m$ (message to be sent)

```
1:  function BROADCAST_ONE_TO_ALL(D, p, s, m)
2:      p = get_processor_index
3:      VID = p^source                          ▷ a^b denotes bitwise XOR operation between a and b
4:      mask = pow(2,D)−1                       ▷ pow(a,b) denotes denotes a^b
5:      for i ← D − 1 to 0 do
6:          mask = mask^pow(2,i)
7:          if (VID & mask)=0 then              ▷ a&b denotes bitwise AND operation between a and b
8:              if VID & pow(2,i)=0 then
9:                  v_dest=VID^pow(2,i)
10:                 q=v_dest^source
11:                 send(m,q)
12:             else
13:                 v_source=VID^pow(2,i)
14:                 q=v_source^pow(2,i)
15:                 receive(m,q)
16:             end if
17:         end if
18:     end for
19: end function
```

Figure 1 shows a one-to-all broadcast on an eight-node (three-dimensional) hypercube with node 0 as the source. In this figure, communication starts along the highest dimension (that is, the dimension specified by the most significant bit of the binary representation of a node label) and proceeds along successively lower dimensions in subsequent steps.

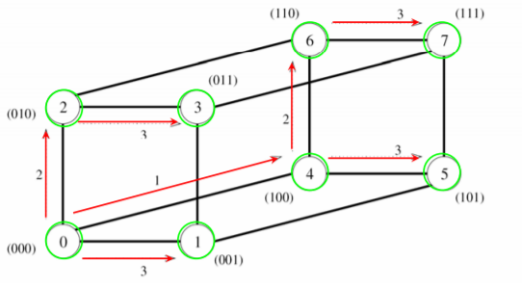



Figure 1: One-to-all broadcast on a three-dimensional hypercube. The binary representations of node labels are shown in parentheses

(b) Time performance analysis for Algorithm 1
If $P$ processes participate in the operation and the data to be broadcast or reduced contains $m$ words, then the broadcast procedure involves $\log P$ point-to-point simple message transfers, each at a time cost of $t_s + t_w m$. Therefore, the total time taken by the procedure is

$$\text{Number of steps: } \boldsymbol{D} = \log_2 \boldsymbol{P}$$

$$\text{Time per step: } t_s + t_w m$$

$$\text{Total time: } (t_s + t_w m) \log_2 \boldsymbol{P}$$

(c) In the scatter operation, a single node sends a unique message of size $m$ to every other node. Unlike one-to-all broadcasting, the source node starts with $P$ unique messages, one destined for each node. Although the scatter operation is semantically different from one-to-all broadcast, the scatter algorithm is quite similar to that of the broadcast. The communication patterns of one-to-all broadcast, as seen in Figure 1 and scatter (Figure 5) are identical. Only the size and the contents of messages are different. In Figure 5, the source node (node $0$) contains all the messages. The messages are identified by the labels of their destination nodes. In the first communication step, the source transfers half of the messages to one of its neighbors. In subsequent steps, each node that has some data transfers half of it to a neighbor that has yet to receive any data.There is a total of $\log P$ communication steps corresponding to the $\log P$ dimensions of the hypercube.

**Cost Analysis:** As Figure 5 illustrates, in each communication step of the scatter operations, data flow from one sub-cube to another. The data that a node owns before starting communication in a certain dimension are such that half of them need to be sent to a node in the other sub-cube. In every step, a communicating node keeps half of its data, meant for the nodes in its sub-cube, and sends the other half to its neighbor in the other sub-cube. The time in which all data are distributed to their respective destinations is

$$T_P = t_s \log P + t_w m(P-1)$$

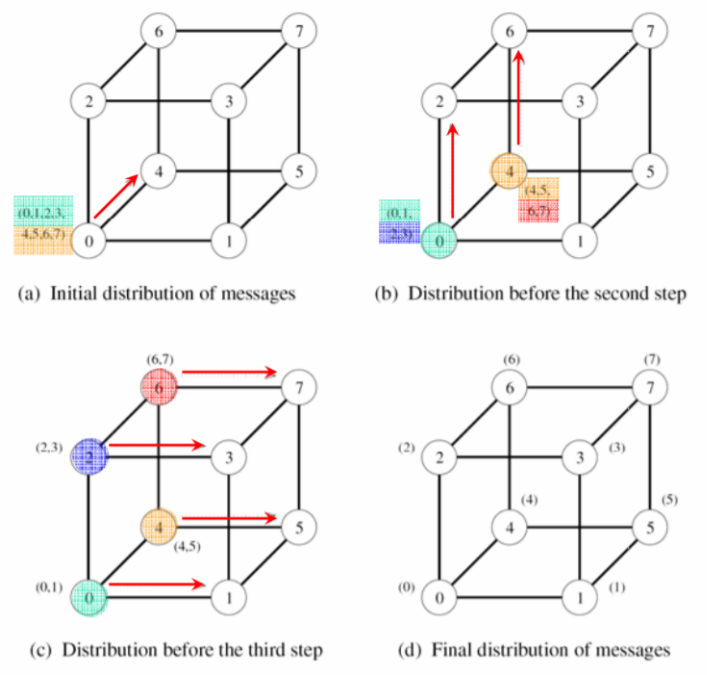



Figure 2: One-to-all broadcast on a three-dimensional hypercube. The binary representations of node labels are shown in parentheses

2. (a) The transpose $A^T$ of a matrix $A$ has the rows and columns interchanged. It can be thought of as flipping the matrix along the main diagonal. Figure 3 shows a matrix which is distributed on mesh of P processors. Notice the diagonal elements in transposition process remain unchanged. Each block on the processor can be transposed locally. This can be achieved if each element below the diagonal move up the diagonal and then towards the right to its correct place and the elements above the diagonal must move down and left. Figure 4 shows the two steps of the matrix transposition, as the elements are first broadcast between processors and then transposed locally.

   If the processor mesh is quadratic, with a total of $P$ processors, then the algorithm amounts to $2\sqrt{P}$ cyclical shifts between adjacent processors as illustrated in Figure 4.

   The algorithm consists of repeating two steps, indicated with thin and thick arrows respectively. In order for communication to be carried out without deadlocks a red-black communication pattern is necessary. In order for a processor to execute the correct action we will also need three colors, one for the processors on the main diagonal, one for those above and one for those above.

   Depending on which color the processor has, it will either send its message to the neighbor to the east, south, west or north. The message that is passed on is the sub-matrix stored in that processor.

   For example, processor $P_1$ will in the first step send its content to $P_5$, and will then receive the sub-matrix originating from $P_12$ via $P_{12} \rightarrow P_8 \rightarrow P_4 \rightarrow P_0$. This is the final positioning for that sub-matrix.

   The process continues in this manner and, given this coloring scheme, has at most $\sqrt{P}$ processes working at the same time.

   The elements that has to travel the furthest are the ones in the upper right and lower left corners, which must traverse along the entire column and then across the entire row, thus taking $2\sqrt{P}$ turns to arrive at the appropriate processor.

   Once all sub-matrices have been cycled through, they are locally transposed on each processor, which yields us the final transposition of the entire matrix.

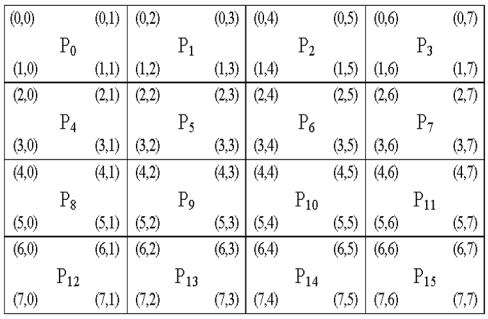



Figure 3: Block Check board partitioning of Matrix on P x P process mesh

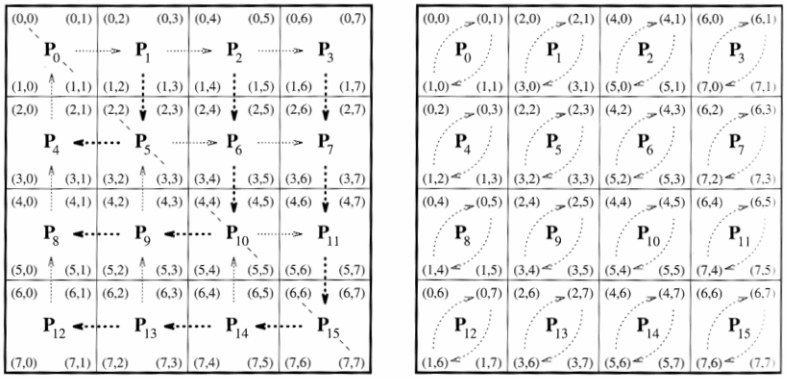


Figure 4: Transposition of matrix distributed on a P x P process mesh

(b) Each processor will hold a sub-matrix of size $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ for a total of $\frac{n^2}{P}$ elements. Thus, each communication step will take $t_s + \frac{n^2}{P} t_w$ timesteps.
If we consider that the time that it takes for the most distant elements to arrive at their correct position is $2\sqrt{P}$, then the time taken to execute this parallel algorithm will be

$$\begin{aligned} T_P &= \frac{n^2}{2P} + 2\sqrt{P}t_s + 2\sqrt{P}t_w \frac{n^2}{P} \\ &= \frac{n^2}{2P} + 2\sqrt{P}t_s + 2\frac{n^2}{\sqrt{P}t_w} \end{aligned}$$

Here the $\frac{n^2}{2P}$ is the cost of the local transposition of a $\frac{n}{\sqrt{P}} \times \frac{n}{\sqrt{P}}$ matrix, which is done in parallel by all processors.
The corresponding computation time for a sequential algorithm is

$$T_S* = \mathcal{O}(n^2)$$

3. The parallel algorithm to solve the equation using Jacobi iteration was implemented in C (See appendix for source code). The functions used for the problem are :

$$\begin{aligned} u(x) &= x^2(x-1) \\ r(x) &= -x \\ f(x) &= 2 - x * x(x-1) \end{aligned}$$

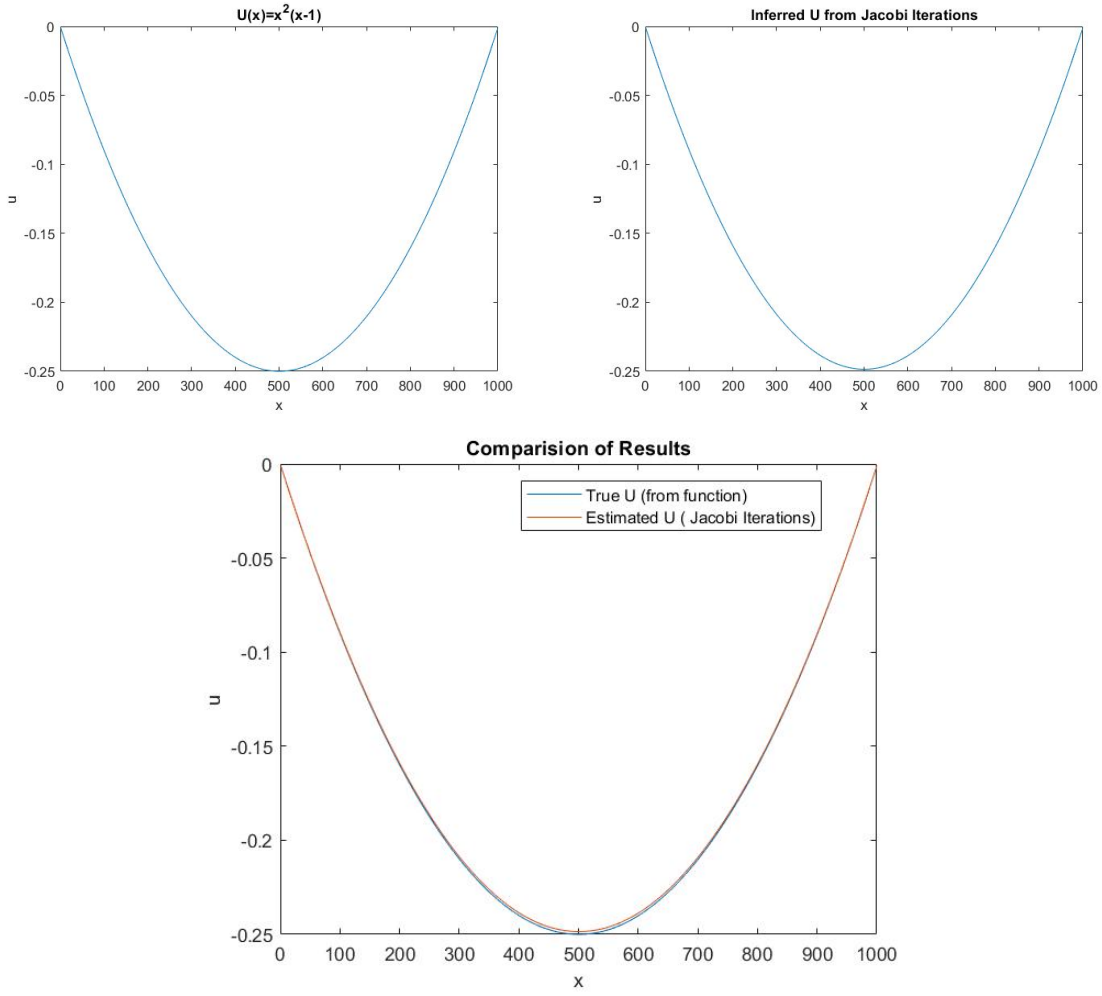
The conditions are satisfied:

$$\begin{aligned} r(x) \leq 0 \; \forall \; x \; \epsilon \; [0,1] \\ u(0) = u(1) = 0 \end{aligned}$$

The analytic solution can also be verified in Matlab by using the below script

```
syms y(x)
ode = diff(y,x,2) ==x*y+2-x *x *(x-1)
cond1 = y(0) == 0
cond2 = y(1) == 0
conds = [cond1 cond2];
ySol(x) = dsolve(ode,conds)
```

The true value of $u$ (obtained using the function derived analytically ), the inferred value of u (derived after running 1 Million Jacobi Iterations )and their comparison together can be seen in Figure 3.




The plot below shows the mean squared error between the true value of u and the inferred value of u after every 10000 iterations.It can be observed that the error is monotonically decreasing with the number of iterations.

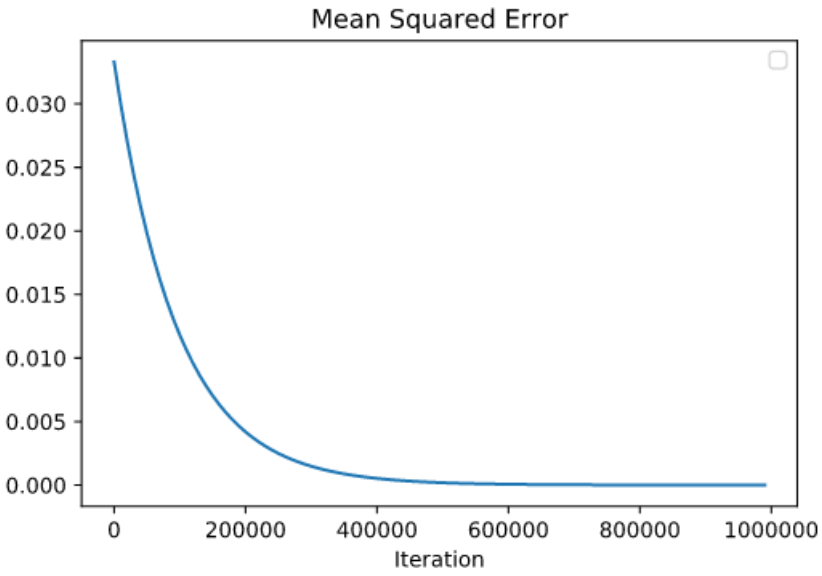



Figure 5: Mean Squared Error after every 10,000 Jacobi Iterations

## References

[1] Barry Wilkinson and Michael Allen. 2004. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition). Prentice-Hall, Inc., USA.

[2] http://parallelcomp.uw.hu/index.html

[3] http://people.cs.aau.dk/ adavid/teaching/MVP-08/09+10a-MVP08.pdf

[4] http://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR93411.pdf

# Appendix

```c
/* Reaction-diffusion equation in 1D
 * Solution by Jacobi iteration
 * simple MPI implementation
 *
 * C Michael Hanke 2006-12-12
 */

#define MIN(a, b)((a) < (b) ? (a) : (b))

/* Use MPI */
#include "mpi.h"
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

/* define problem to be solved */
# define N 1000 /* number of inner grid points */
# define SMX 1000000 /* number of iterations */

/* implement coefficient functions */
extern double r(const double x);
extern double f(const double x);

double r(const double x) {
  return -x;
}

double f(const double x) {
  return 2 - x * x * (x - 1);
}

/* We assume linear data distribution. The formulae according to the lecture
   are:
      L = N/P;
      R = N%P;
      I = (N+P-p-1)/P;    (number of local elements)
      n = p*L+MIN(p,R)+i; (global index for given (p,i)
   Attention: We use a small trick for introducing the boundary conditions:
      - The first ghost point on p = 0 holds u(0);
      - the last ghost point on p = P-1 holds u(1).
   Hence, all local vectors hold I elements while u has I+2 elements.
*/
/* returns global index for a given p and i */
int getGlobalIndex(int L, int R, int p, int i) {
  return p * L + MIN(R, p) + i;
}
int main(int argc, char * argv[]) {
  /* local variable */
  int P, p, M, L, R, I, tag, step;
  double h;
  /* Initialize MPI */
  MPI_Status status;
  MPI_Init( &argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &P); /* Number of processors*/
  MPI_Comm_rank(MPI_COMM_WORLD, &p); /* Current processors*/
  if (N < P) {
    fprintf(stdout, "Too few discretization points...\n");
    exit(1);
  }
```

```
60    tag = 100;
61    /* Compute local indices for data distribution */
62    L = N / P;
63    R = N % P;
64    I = p < R ? L + 1 : L;
65    h = 1.0 / (N + 1);
66
67    /* arrays */
68    double *unew = (double * ) calloc(I, sizeof(double));
69    double *rr = (double * ) calloc(I, sizeof(double));
70    double *ff = (double * ) calloc(I, sizeof(double));
71    double *u = (double * ) calloc(I + 2, sizeof(double));
72
73    for (int i = 0; i < I; i++) {
74      int z = getGlobalIndex(L, R, p, i);
75      double xn = z * h;
76      ff[i] = f(xn);
77      rr[i] = r(xn);
78    }
79
80    /* Jacobi iteration */
81    for (step = 0; step < SMX; step++) {
82      /* RB communication of overlap */
83      if (p % 2 == 0) { // red
84        if (p != P - 1) { // check if no processor on the right
85          MPI_Send(&u[I], 1, MPI_DOUBLE, p + 1, tag, MPI_COMM_WORLD);
86          MPI_Recv(&u[I + 1], 1, MPI_DOUBLE, p + 1, tag, MPI_COMM_WORLD,&status);
87        }
88        if (p != 0) { // check if no processor on the left
89          MPI_Send(&u[1], 1, MPI_DOUBLE, p - 1, tag, MPI_COMM_WORLD);
90          MPI_Recv(&u[0], 1, MPI_DOUBLE, p - 1, tag, MPI_COMM_WORLD,&status);
91        }
92      } else { // black
93        if (p != 0) {
94          MPI_Recv(&u[0], 1, MPI_DOUBLE, p - 1, tag, MPI_COMM_WORLD,&status);
95          MPI_Send(&u[1], 1, MPI_DOUBLE, p - 1, tag, MPI_COMM_WORLD);
96        }
97        if (p != P - 1) {
98          MPI_Recv(&u[I + 1], 1, MPI_DOUBLE, p + 1, tag, MPI_COMM_WORLD,&status);
99          MPI_Send(&u[I], 1, MPI_DOUBLE, p + 1, tag, MPI_COMM_WORLD);
100       }
101     }
102     /* local iteration step */
103     for (int i = 0; i < I; i++) {
104       unew[i] = (u[i] + u[i + 2] - h * h * ff[i]) / (2.0 - h * h * rr[i]);
105     }
106     for (int i = 0; i < I; i++) {
107       u[i + 1] = unew[i];
108     }
109
110       /* The code below saves the u after 10000 step and is used for plotting the error */
111   //   if(step%10000==0)
112   //   {
113   // const char * filename = "outputError.txt";
114   // double writeSignal = 1.0;
115   // FILE * f;
116   // if (p == 0) {
117   //   f = fopen(filename, "a");
118   //   //  fprintf(f,"\n%d\n", step);
119   //   for (size_t i = 0; i < I; i++) {
120   //     fprintf(f, "%f ", unew[i]);
```

```
    //   }
    //   fclose(f);
    //   MPI_Send(&writeSignal, 1, MPI_DOUBLE, p + 1, tag, MPI_COMM_WORLD);
    // } else {
    //   MPI_Recv(&writeSignal, 1, MPI_DOUBLE, p - 1, tag, MPI_COMM_WORLD, &status);
    //   f = fopen(filename, "a");
    //   for (size_t i = 0; i < I; i++) {
    //     fprintf(f, "%f ", unew[i]);
    //   }
    //   if(p==P-1){
    //   fprintf(f, "\n\n");}
    //   fclose(f);
    //   if (p != P - 1) {
    //     MPI_Send(&writeSignal, 1, MPI_DOUBLE, p + 1, tag, MPI_COMM_WORLD);
    //   }
    // }

    //   }

    }

    /* output for graphical representation */
    /* Instead of using gather (which may lead to excessive memory requirements
     on the master process) each process will write its own data portion. This
     introduces a sequentialization: the hard disk can only write (efficiently)
     sequentially. Therefore, we use the following strategy:
     1. The master process writes its portion. (file creation)
     2. The master sends a signal to process 1 to start writing.
     3. Process p waites for the signal from process p-1 to arrive.
     4. Process p writes its portion to disk. (append to file)
     5. process p sends the signal to process p+1 (if it exists).
  */
    const char * filename = "output.txt";
    double writeSignal = 1.0;
    FILE * f;
    if (p == 0) {
      f = fopen(filename, "w");
      for (size_t i = 0; i < I; i++) {
        fprintf(f, "%f ", unew[i]);
      }
      fclose(f);
      MPI_Send(&writeSignal, 1, MPI_DOUBLE, p + 1, tag, MPI_COMM_WORLD);
    } else {
      MPI_Recv(&writeSignal, 1, MPI_DOUBLE, p - 1, tag, MPI_COMM_WORLD, &status);
      f = fopen(filename, "a");
      for (size_t i = 0; i < I; i++) {
        fprintf(f, "%f ", unew[i]);
      }
      fclose(f);
      if (p != P - 1) {
        MPI_Send(&writeSignal, 1, MPI_DOUBLE, p + 1, tag, MPI_COMM_WORLD);
      }

    }
    /* That's it */
    MPI_Finalize();
    exit(0);
  }
```