Parallel Computing
Exercise 3 (June 8, 2021)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

TThe result is to be submitted by the deadline stated above via the Moodle interface. The result is submitted as a .zip or .tgz file which contains

- a single PDF (`.pdf`) file with
  - a cover page with the title of the course, your name(s), Matrikelnummer(s), and email-address(es),
  - the source code of the sequential program,
  - the demonstration of a sample solution of the program,
  - the source code of the parallel program,
  - the demonstration of a sample solution of the program,
  - a benchmark of the sequential and of the parallel program.
- the source (`.c`/`.cpp`/`.java`) files of the sequential and of the parallel program.

Distributed Memory Programming in MPI

The goal of this exercise to develop in MPI a distributed memory solution to one of the problems specified in Exercise 1. As the base of your parallel programming effort, you may use the sequential program you have developed in Exercise 1; you may also write a new sequential program or ask one of your colleagues for one. Our default assumption is that the programs for this assignment are written in C, using the official MPI binding for the parallel solution.

Having said this, you may also write your sequential program in Java and use for the parallelization one of the MPI bindings for Java provided by

- OpenMPI: `https://www.open-mpi.org/faq/?category=java`
- FastMPJ: `http://gac.udc.es/~rreye/fastmpj/`
- MPJ Express: `http://mpj-express.org`

OpenMPI is available at the course machine (see `module avail`), without guarantee of a functional Java interface; newer versions of this package respectively the other packages are to be installed on your own.

However, neither do we recommend to solve this assignment in Java nor will we be able or willing to give any support for the use of Java with MPI.

Benchmarking Benchmark the programs (both the sequential and the parallel one) as in Exercise 1; you may also use the MPI function `double MPI_Wtime()` which returns the current wall clock time in seconds. Report the results as in Exercise 1.

Contiguous Matrices If a matrix A of dimension $M \times N$ with values of type T is to be passed (respectively broadcast/scattered/gathered) among processes, make sure that A is represented by a contiguous block in memory. This can be achieved either by a global declaration `T A[M][N]` (which allocates the matrix in the data segment of the process, M and N have then to be compile-time constants) or by a declaration and initialization `T* A = malloc(M*N*sizeof(T))` (which allocates the matrix on the heap, N may then be variable; however, the element $A[i][j]$ is now denoted by the reference `A[i*M+j]`).

Alternative A: Matrix Inversion by Gauss-Jordan Elimination

In the MPI solution to this problem, you you may assume that the number P of processes divides the matrix dimension N exactly.

- The program starts by distributing matrix A *row-wise* among the P processes *in a round-robin fashion* (i.e. process 0 receives rows $0, P, 2P, \ldots$, process 1, receives rows $1, P+1, 2P+1, \ldots$, and so on). By this distribution, we ensure that the workload is evenly shared in the later phases of the diagonalization (when the remaining part of $(A|I)$ becomes small).

  To distribute A, process 0 constructs a correspondingly permuted version A' of the matrix and scatters its values among all processes (by a single call of `MPI_Scatter`).

- For performing the diagonalization, the program runs in N iterations, where in iteration i process $p = i\%P$ broadcasts row i to all other processes (`MPI_Bcast`). Each process then uses this row to update all the rows of the matrix for which it is responsible.

  To simplify the program, you may assume that $A(i, i)$ is different from 0 (if this should not be the case, you may abort the computation by `MPI_Abort`).

- Finally, process 0 gathers the permuted inverse matrix B' from all processes (by a single call of `MPI_Gather`) and constructs from this the actual inverse matrix B.

Alternative B: Solving the 15 Puzzle by A* Search

In the MPI solution to this problem, process 0 runs the sequential $A*$ search to determine *info* and *open* with at least P nodes with minimal path length estimations. The parallel search is executed by P processes in the following way:

- Process 0 broadcasts *info* to every process (`MPI_Bcast`). For this, the data structure has to be linearized appropriately, either into an array of a custom data type or into multiple arrays that hold plain integers that are broadcast separately (which is simpler but less efficient).

- Likewise process 0 constructs a linearized version of *open* with P nodes and scatters it among all processes (one or more calls of `MPI_Scatter`).

- Each process performs its local $A*$ search but regularly checks whether a message has arrived from process 0 that indicates that the search can be terminated (`MPI_Iprobe`); if this is the case, the process terminates the search and sends an acknowledgment back to process 0 (`MPI_Send`).

- If some process finds the goal node, it sends to process 0 the result data (`MPI_Send`). When process 0 receives that message (`MPI_Iprobe`), it sends to all other processes messages telling them to terminate their search (`MPI_Send`); process 0 then waits for their acknowledgments (`MPI_Receive`).

The execution time of the program measured in process 0 contains the time until it has received the final acknowledgment.