Parallel Computing
Exercise 2 (May 17, 2022)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above via the Moodle interface.
The submitted result is 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`/`.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://mpjexpress.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]`).

Presentation Please be prepared to give a short (10 min) presentation of your results on May 24; you will be notified by May 18 whether such a presentation is requested from you.

Alternative A: Gaussian Elimination

For this alternative, you you may assume that the number P of processes divides the matrix dimension N exactly. Furthermore, the distribution of data among processes becomes simpler if the system A, b is represented as a single matrix that holds in an additional column the vector b.

- The program starts by distributing the system A, b *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 triangulation (when the non-zero part of A becomes small).

  To distribute A, process 0 constructs a correspondingly permuted version A', b' of the system to scatter the values among all processes (by a single call of `MPI_Scatter`).

- For performing the triangulization, 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 system 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`).

- For performing the back-substitution, the program runs in N iterations where in each iteration the process $p = N\%i$ that holds the newly computed result $x[i]$ broadcasts this value to all other processes (`MPI_Bcast`). Each process then uses this value to remove one unknown from all the rows of the system for which it is responsible.

- Finally, since process 0 has received all values that were broadcast during back-substitution, it can determine the result x.

Alternative B: Traveling Salesman

For this alternative, implement a version where the root serves as a manager of the tasks to be distributed among additional *P* worker processes:

- The root manages the global pool of paths from which each worker may query new work, i.e., the worker receives a partial path which it has to extend in all possible ways for a potential optimum solution using a local pool of paths.
- Whenever a worker queries the manager for a partial path, the manager provides as part of its answer the length of the shortest cycle found so far which the worker may use to prune its search.
- Whenever a worker finds a potentially shorter cycle, it informs the manager about this cycle and receives as an answer the length of the current shortest cycle which it may subsequently use for pruning.
- If the manager has only one partial path left in its pool, it extends this path in all possible ways such that subsequent queries by multiple workers may be addressed.
- If the manager runs out of work; it informs each client upon its next request about this fact, such that the clients may subsequently terminate; when all clients have been informed, the manager may terminate.