Parallel Computing
Exercise 3 (May 27, 2025)

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, matriculation number, and email address,
  - 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://docs.open-mpi.org/en/v5.0.x/features/java.html`
- 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 June 3; you will be notified by May 29 whether such a presentation is requested from you.

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). In such a way we generally ensure that the work load of a process is not influenced by a particular distribution of data in the matrix.

  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: Bin Packing

Implement an MPI solution to the bin packing problem where process 0 serves as a manager that distributes tasks to additional P worker processes (speedups and efficiencies are computed relative to P, not $P + 1$).

- The manager holds a pool of (descriptions of) open tasks, which are the arguments of calls of `binpack` to be executed. For every worker, the manager records which task the worker currently processes or whether the worker is idle. The manager also records the values *rlen* and *result* of the currently best packing (initially a non-optimal one computed by a fast algorithm).
- The manager starts the execution of the parallel algorithm by sending the description of the initial task (the initial call of `binpack`) to some worker (and recording this).
- A task description sent to a worker consists of the arguments of a call of `binpack` except:
  - The partial result *result0* is not included but assumed to be the empty sequence (thus the worker only determines the sequence of new bins to be added to *result0*).
  - The length *rlen* of the best packing currently known to the manager is included (the worker may use this value to prune its search).
- Each worker iteratively waits for a task, receives one from the manager, executes this task (by executing `binpack` with the given arguments), informs the manager when it has completed the execution, and then waits for another task.
- If the worker (by executing `binpack`) creates a new task (for the parallel execution of a recursive call of `binpack`), it sends this task to the manager. When the manager receives a task from some worker, it forwards the task to some idle worker and records this. If there is no idle worker, the task is placed into the pool.
- Whenever a worker has has found a shorter packing, it sends this packing (*rlen* and *result*) to the manager. The manager checks whether the new value *rlen* indeed is smaller than its own value (which may have been decreased in the mean time). If this is the case, the manager updates its own value of *rlen* and replaces *result* by the concatenation of *result0* (which was not transmitted to the worker) and the value of *result* received from the worker.
- When a worker reports the completion of a task, the manager sends it another task from its pool or (if the pool is empty) records the worker as idle.
- When the task pool becomes empty and all workers are idle, the manager terminates.

By above scheme, a worker processing a task is not immediately informed about improvements of *rlen* by other workers. We may possibly optimize the scheme such that, when the manager receives an improved packing from some worker, it sends the improved value of *rlen* to all other workers. Every worker may regulary poll for such messages (`MPI_Iprobe`) and use them to keep its own value of *rlen* up to date.

Please experiment with this (or a similar kind of) optimization and report whether it decreases (or rather increases) the execution time; use for your benchmarking that version of the program that gives the best performance.