

# Parallel Computing

## Exercise 4 (June 16, 2020)

Wolfgang Schreiner  
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above via the Moodle interface. If the assignment has been elaborated in a collaboration of two students, only one of them shall upload the assignment (indicating of course on the cover page the collaboration partner).

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 is 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: Gaussian Elimination

For this alternative, 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  $0, P, 2P, \dots$ , process 1, receives rows  $1, P + 1, 2P + 1, \dots$ , 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).

- 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  $x$ .

## Problem B: The Number of Satisfying Assignments of a Formula

In the MPI solution to this problem the process 0 serves as a manager of a pool of tasks which it distributes among additional  $P$  worker processes (thus in total  $P + 1$  processes are used):

- The manager holds a pool of partial assignments still to be processed (initially only the empty assignment  $a = []$ ), the number of satisfying assignments reported by the workers so far (initially 0), and, for each worker, the information whether the worker is currently working or waiting for a new assignment to be processed (initially all workers are waiting).
- To each worker that is waiting the manager attempts to send a partial assignment from its pool. However, before the manager gives a worker the last assignment in its pool (such that the pool would become empty), the manager attempts to extend this assignment to  $2^t$  partial assignments of which it forwards one to the worker and keeps the other ones.
- When a worker has processed the partial assignment(s) on its local stack such that this stack becomes empty, it returns the number of satisfying assignments to the server as a result and waits for a new assignment to be processed.
- Whenever a worker returns as a result a number of satisfying assignments, the manager records this information and attempts to send the worker a new assignment. If this is not possible, the manager records the worker as waiting.
- From time to time (e.g., when it has popped a certain number of assignments from its local stack), every worker sends some of its oldest assignments (those with the most variables not yet assigned) from the top of its local stack to the server.
- If a server receives from a worker some assignments (rather than a result), it puts them into its pool and attempts (as described above) to send assignments to all waiting workers.
- If the manager detects that all clients are waiting and its pool is empty, it reports the total number of satisfying assignments and informs all workers. The manager and the workers may then terminate.

Please note that in this solution, since each worker may send without request additional assignments to the server, the number of assignments that the manager keeps in its pool is not bounded by  $2^t - 1$ ; thus the pool may have to be dynamically expanded.