Introduction to
Parallel and Distributed Computing
Exercise 2 (May 9)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.uni-linz.ac.at

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

- a PDF file with
  - a cover page with the title of the course, your name, Matrikelnummer, 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 in the style of Exercise 1.
- the source (`.c`) files of the sequential and of the parallel program.

Exercise 2: OpenMP and Cilk

Gaussian Elimination is a well-known algorithm for solving a linear system $Ax = b$ of some dimension n: we are given a matrix A of size $n \times n$ and a vector b of length n and want to find that vector x of length n that makes the equation true.

The algorithm consists of two steps:

1. The system is converted to an upper-triangular system $Tx = e$ (i.e. all coefficients below the main diagonal of T are zero) which has the same solution(s) as the original system.

2. The new system $Tx = e$ is solved by backward substitution (we determine the solution $x_{n-1} = e_{n-1}$, and substitute the solution in the given system which produces a new upper-triangular system of dimension $n - 1$ which can be solved in the same way).

Details about the algorithm are given in the file `GaussianElimination.PDF` in the "Restricted Area" of the course site (Figure 9-13 describes triangulation, Figure 9-3 describes back-substitution).

While Gaussian Elimination is typically not used when the coefficients in A and b are floating point numbers (here mainly iterative methods are used for determining approximate solutions), it plays an important role if the coefficients are integer or rational numbers and the equation is to be solved *exactly* (as it is done in computer algebra systems).

Sequential Program

Your first task is to write in C/C++ a function `solve(A, b)` that returns the solution of the linear system determined by matrix A and vector b (or `null`, if there is no solution or there exist multiple solutions).

You may construct a "straight-forward" version of the algorithm where any non-zero coefficient may serve as a pivot element in the triangulization step (i.e. is not necessary to take the element with the maximum absolute value, as it is done in the algorithm of Figure 9-1).

Demonstrate the correctness of your program by solving a random 4×4 system and giving the output of the program (system and solution).

Benchmark the execution time of `solve` for randomly initialized matrices of dimensions $N = 256$ and $N = 378$ (in case these values take much too long or much too short, you may adapt N correspondingly); make sure that you compile the program with optimization option `-O3`.

Parallel Program

The more time-consuming part of Gaussian Elimination is the conversion of the system into upper-triangular form where in n iterations one row of the system after the other is converted into the right form. In iteration i of the algorithm, all coefficients of A below and to the right of position (i, i) have to be processed (see Figure 9-12); since this can be done independently for each coefficient, we have basically a parallel algorithm.

Based on this idea, modify the sequential program (if necessary) such that the iterations of the j-loop (processing different rows of the matrix) can be performed independently of each other.

For the actual parallelization, you may choose one of the options below (if you choose both, you will get 30% bonus grade). Whatever option you choose, demonstrate the correctness of your parallel program in the same way as for the original one.

Benchmark your program for the values of N given above and for $P = 1, 2, 4, 8, 16, 32$ processors; you should get substantial speedups. In your (sequential and parallel) benchmarks, apply the same strategy as explained in Exercise 1 (you may just perform three runs and take the average) and report the same results (execution times with linear and logarithmic scales, speedups and efficiency). Optionally you may work with different node affinity strategies (environment variables `GOMP_CPU_AFFINITY` or `KMP_AFFINITY` respectively command `cpuset`) to improve the performance.

Option A: OpenMP

Parallelize the program by annotating the j-loop with OpenMP pragmas such that the loop gets executed in parallel; do not forget to mark "private" variables appropriately. Compile the program with options `-O3 -openmp -openmp-report2` and explain the compilation output. Optionally you may experiment with different scheduling strategies (clause `schedule(runtime)`, environment variable `OMP_SCHEDULE`).

Option B: Cilk

Parallelize the program by using for the j-loop either the `_Cilk_for` statement (iterative version) or the _Cilk_spawn statement (recursive version) which you may write as `cilk_for` respectively `cilk_spawn` if you include the header file `<cilk/cilk.h>`. Compile the program with options `-O3 -liomp5 -lpthread`.

Benchmarking

In your implementation, you may use for coefficients and results simply floating point numbers (type `double`). However, in order to simulate the larger computation time of arbitrary precision arithmetic, use in your benchmarks the function

```
double smult(double a, double b) {
  double c = 0;
  for (int i=0; i<10000; i++) { c = 1-c*c; }
  return a*b+c;
}
```

as a "slow multiplication" operation whenever the multiplication of coefficients is required.

Bonus (30%): Rather than using floating point arithmetic and the "slow multiplication" operation `smult`, implement the program with arbitrary precision rational arithmetic using the GNU Multiple Precision Arithmetic Library[1] using type `mpq_t`; you have to include header `mkl_gmp.h` and compile with option `-lmkl`; adapt N to reasonable sizes.

[1] See https://gmplib.org/, section "Rational Number Functions".