Parallel Computing
Exercise 1 (April 19, 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.

Shared Memory Programming in C/C++ with OpenMP or in Java

Develop a sequential and a parallel solution to *one* of the subsequently stated problems, *either* in C/C++ with OpenMP *or* in Java using the Java basic thread/high-level concurrency API.

Instrument the source code of your program to measure the real ("wall clock") time spent (only) in that part of your program that you are interested in (the core function without initialization of input data and output of results) and print this time to the standard output. In C/C++ with OpenMP, you can determine wall clock times by the function `omp_get_wtime()`, in Java you can determine it by `System.currentTimeMillis()`.

When running the parallel programs, make sure that threads are pinned to freely available cores; use `top` to verify the applied thread/core mapping and the thread's share of CPU time (which should be close to 100%). In a C/C++ solution with OpenMP, make sure that both your sequential and parallel program are compiled with all optimizations switched on (option `-O3`).

When benchmarking the parallel program, make sure that you run the parallel program with the same actual inputs (not only the same input sizes) as the sequential one by using the same random number generator seeds in the generation of inputs (if applicable).

Repeat each benchmark (at least) five times, collect all results, drop the smallest and the highest value and take the average of the remaining three values. For automating this process, the use of a shell script is recommended. For instance, a shell script `loop.sh` with content

```
#!/bin/sh
for P in 1 2 4 8 16 32 ; do
  echo $P
done
```

can be run as `sh loop.sh >log.txt` to print a sequence of values into file `log.txt`.

Present all timings in an adequate form in the report by

- a numerical table with the (average) execution times of sequential and parallel programs for varying input sizes and processor numbers, (absolute) speedups and (absolute) efficiencies;
- diagrams that illustrate execution times, speedups, and efficiencies with both linear and algorithmic axes, as shown in class (multiple runs should be shown in the same diagram by different curves, if the scales are comparable);
- ample verbal explanations that explain your compilation/execution settings, how you interpret the results, how you judge the performance/scalability of your programs.

Tip If you program in C/C++, the tool `valgrind`[1] is useful to debug invalid memory accesses; this package is included in many Linux distributions (Debian: `apt-get install valgrind`).

Presentation Please be prepared to give a short (10 min) presentation of your results on April 26; you will be notified by April 20 whether such a presentation is requested from you.

[1] http://valgrind.org/

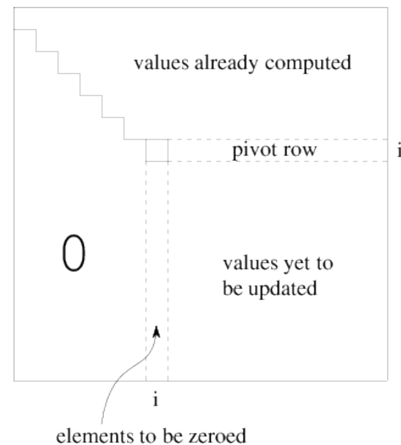
Alternative A: Gaussian Elimination

Gaussian Elimination is a well-known algorithm for solving a linear system $Ax = b$ of dimension N: given a matrix A of size $N \times N$ and a vector b of length n, we 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 $\widehat{A}x = e$ (i.e. all coefficients below the main diagonal of $\widehat{A}$ are zero) which has the same solution(s) as the original system.

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



Descriptions of the algorithm can be found in many sources, e.g., Wikipedia; the core idea is that in iteration i of the triangulation the element $A(i, i)$ serves as the pivot element: from every row $j > i$ we subtract the multiple $A(j, i)/A(i, i)$ of row i. 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 from a domain where the equation is to be solved *exactly* (as in computer algebra systems). In this assignment we will consider the domain $\mathbb{Z}/p = \{0, 1, \ldots, p - 1\}$ where p is a prime number and arithmetic is integer arithmetic modulo p as implemented by the following C functions:

```
static long mAdd(long a, long  b) { return (a+b)%p; }
static long mSub(long a, long b)  { return (a+p-b)%p; }
static long mMul(long a, long b)  { return (a*b)%p; }
static long mDiv(long a, long b)  { return mMul(a, mInv(b)); }
```

Here `mInv`(a) computes the modular inverse of a (i.e., the value a' for which $a \cdot a' \equiv 1 \pmod{p}$ holds) by applying the extended Euclidean algorithm:

```
static long mInv(long a)
{
  long r = p; long old_r = a;
  long s = 0; long old_s = 1;
  while (r != 0)
  {
    long q = old_r/r;
    long r0 = r; r = old_r-q*r; old_r = r0;
    long s0 = s; s = old_s-q*s; old_s = s0;
  }
  return old_s >= 0 ? old_s : old_s+p;
}
```

Sequential Program

Your first task is to implement a sequential program solving the problem for randomly generated equation systems of dimension N and $p = 982451653$ ($\approx 2^{30}$).

You may construct a "straight-forward" version of the algorithm that aborts with a corresponding message, if there is no solution or there exist multiple solutions. Since arithmetic is exact, 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).

Demonstrate the correctness of your program by solving a random 4×4 system for, e.g., $p = 11$, and giving the output of the program (system and solution). Benchmark the execution time of your solution (the time for solving the system not including the initialization time) for randomly initialized matrices with at least *two* different dimensions that let the program run at least 1 min and at least 3 min, respectively.

Parallel Program (Basic Version)

The much 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 outermost loop of the triangulization, all coefficients of A below and to the right of position (i, i) have to be processed; since this can be done independently for each coefficient, we can apply parallelism.

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

- C/C++: use OpenMP pragmas to ensure that the loop gets executed in parallel; do not forget to mark "private" variables appropriately. Compile the program with options `-O3 -openmp -openmp-report 1`. Experiment with different scheduling strategies respectively chunk sizes to determine the one that gives best performance.

- Java: use the high-level Java concurrency API for creating a thread pool among which tasks are scheduled each of which processes B rows of the matrix; experiment with suitable values for the block size B. Please note that the pool is created only once and reused in every iteration of the triangulation (you may use the method `invokeAll`[2] which blocks until all tasks have been processed).

Benchmark your program for $P = 1, 2, 4, 8, 16, 32$ cores (and potentially more).

Parallel Program (Advanced Version)

Most likely the basic program will not scale well beyond 16 cores (1 blade on the UV 1000) due to the higher latency of memory access across blades. In particular, in every outermost iteration of the algorithm, each matrix row may be accessed by a thread running on another node leading to a transfer of the row to another blade in every iteration step. Therefore write another

[2] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/ExecutorService.html#invokeAll(java.util.Collection)

version of the program that addresses this problem: every row is assigned to the same thread (pinned to a node blade) across multiple iterations: if we have P threads, thread 0 processes rows $0, P, 2P, \ldots$, thread 1 processes rows $1, P+1, 2P+1, \ldots$ and so on (rows are distributed to threads in a "round-robin" fashion).

In Java, this may be achieved by explicitly creating P threads that stay alive during the whole computation. Each thread allocates (in the heap of the node on which it runs) the matrix rows which it subsequently processes; after the allocation phase, the original main thread initializes the matrix with the coefficients. Then the program runs in n iterations where in every iteration each thread processes the rows it is in charge of. For synchronizing all threads after every iteration, you may use class `CyclicBarrier`[3].

In OpenMP this may be achieved by using (like in Java) a matrix representation that stores in an array the start addresses of each row and using repeated execution of `omp parallel` to let each thread process a part of a matrix: in the first execution each thread allocates the memory of the rows for which it is in charge, in every subsequent execution, it processes theses rows.

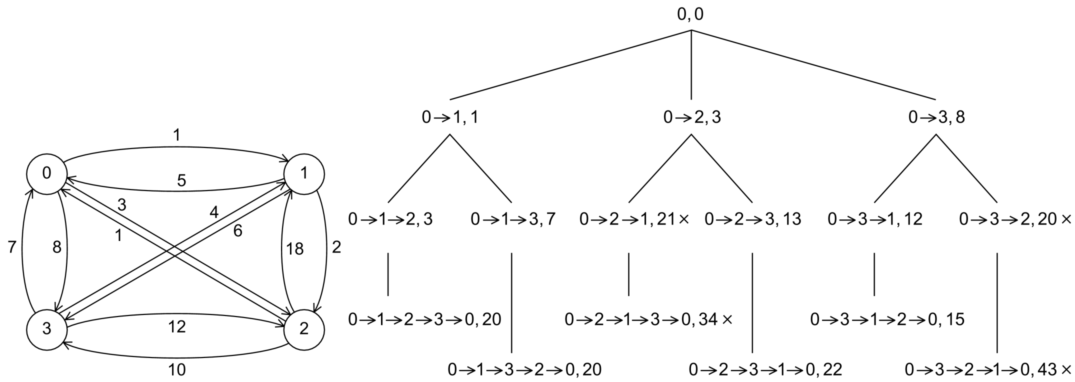
Benchmark the program in the same way as the original version.

Note: there is no guarantee that the advanced version of the program will indeed scale better than the basic version. However, the effort to achieve such an improvement and its evaluation will be judged.

[3] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CyclicBarrier.html

Alternative B: Traveling Salesman

The "traveling salesman" problem is as follows: given a directed graph with n nodes whose edges are labeled with positive lengths, find a cyclic path that contains all nodes and has minimum length. The nodes are identified with the numbers $0, \ldots, n-1$,the edges are represented by a distance matrix d such that, if $d(v_i, v_j) = w > 0$, then node v_i is connected to node v_j by an edge of length w (if $w = 0$, there is no edge connecting these nodes).



Since the problem is NP-complete, in practice algorithms are applied that determine heuristically "good" but not necessarily optimal solutions. We will, however, investigate an algorithm that indeed finds the optimal solution: the core idea is to traverse a search tree where every node is labeled by a path starting at node 0 and by the length of that path. The root of the tree is labeled with the singleton path 0 with length 0; every inner node of the tree with path $0 \to \ldots \to v_k$ and length w has as its children all those nodes whose path $0 \to \ldots \to v_k \to v_{k+1}$ extends the parent path by a node v_{k+1} that does not occur in $0 \to \ldots \to v_k$; the length of this path is $w + d(v_k, v_{k+1})$. The leaves of the tree are labeled with all cyclic paths $0 \to \ldots \to 0$ that contain every non-0 node only once and the length of that path. The shortest path is represented by the leaf with the shortest length.

The tree is constructed with the help of the set of all non-leaf nodes (i.e, paths and associated lengths); initially this set contains only the root. The algorithm proceeds by repeatedly removing one path/weight from the set; if this path consists of n nodes, the root 0 is added, and it is determined whether the resulting cyclic path is shorter than any previously found one; if yes, this path and its length are remembered. If the path has less than n nodes, it is extended in all possible ways by nodes that do not yet occur on the path; if the resulting path is at least as long as the length of a previously encountered cyclic path, it cannot lead to a shorter cyclic path any more and is dropped; otherwise, it is added to the set. The process repeats until the set becomes empty; the remembered cyclic path is then the shortest one. The core of a corresponding sequential C program can be thus as shown on the next page.

This algorithm belongs to the class of "branch and bound" algorithms: it keeps track of an upper bound on the quality of a solution (the length of the shortest cyclic path encountered so far); parts of the search tree that cannot lead to a better solution will be subsequently not investigated. A sequential algorithm processes this tree typically in depth-first order (by organizing the set of path as a stack to which new paths are pushed and from which paths are popped). However,

```
init_path(&path);                    // initialize first path
add_path(path);                      // add path to set
while (pool_number > 0) {            // set is not empty
  remove_path(&path);                // remove path from set
  if (path.number == N) {            // path contains all nodes
    update_result(path);             // possibly update result
    continue;
  }
  for (int i=1; i<N; i++) {          // extend partial path in all ways
    weight_t w = add_node(&path, i); // attempt to add node to a better path
    if (w < 0) continue;             // attempt failed
    add_path(path);                  // add new path to pool
    remove_node(&path, w);           // remove node for next attempt
  }
}
```

the tree can be also be investigated in parallel by multiple concurrent threads that independently remove paths from the set and investigate the corresponding subtrees in parallel. The only points of interaction between the threads are the set of partial paths (from which to remove and to which to add elements), the best solution found so far (which may have to be updated) and the length of the solution (a better bound established by one thread also reduces the subsequent search space of any other thread).

Sequential Program

First, implement *either* in C/C++ *or* in Java (choose here the same language that you use in the parallel solution) a sequential program that solves the problem for randomly generated graphs of dimension n (it should be configurable which fraction of the distance matrix d is not zero); the edge lengths may be represented as integer numbers. The set of paths can be organized as a stack; a little investigation reveals that there cannot be more than $n \cdot (n-1)/2$ (partial) paths on the stack; thus the stack can be represented by a preallocated array of this size. Since also paths have a limited length n, it is recommended to allocate all data structures in advance (rather than continuously allocating and freeing/garbage-collecting these).

Demonstrate the correctness of your implementation by the graph shown above.

The structure of a graph may significantly influence the runtime of the algorithm; thus benchmark the program for (at least) two inputs of different sizes that let the program run at least 1 min and at least 3 min, respectively.

Parallel Program in Java or OpenMP

Implement a shared memory version of the parallel program in Java or in OpenMP; here consider the following points:

- There shall be one shared set of paths from which each thread may remove an element for further processing; apparently access to this set must be synchronized. To ensure that

threads receive *big tasks* (i.e., short paths) this shared set shall be maintained as a *queue* rather than as a stack, i.e., new elements are added at the back rather than at the front.

- Each thread maintains an own local set of paths in the usual way as a stack; the thread has exclusive access to this stack (which thus needs not to be synchronized); the thread adds new paths only to its local stack (not to the shared queue). The memory for the stack shall be allocated by the thread itself to ensure that it resides on the same node as the thread.
- A thread processes paths primarily from its local stack; only when this stack becomes empty, the thread removes a path from the shared queue for further processing. However, if this is the last path in the queue, the thread extends the path in the usual way and leaves all but one of the extended paths in the shared queue for processing by other threads.
- Access to the best solution found so far and its length have to be synchronized. To allow independent processing, however, each thread maintains a local copy of the best length which it uses for bounding its local computation. Only when it thinks it may have found a better solution than the previous one (and thus attempts to update the solution), it also updates its local length by the (potentially better) shared length.
- If all threads find their local queues and the shared queue empty, the algorithm terminates.

OMP Solution The program may create T worker threads that simultaneously process the search tree:

```
#pragma omp parallel ... num_threads(T)
```

Access to shared variables has to be protected by critical sections or lock variables.

Java Solution You may create (as in the OMP-based solution) a fixed set of worker threads that operate on shared synchronized variables. Alternatively, the program may be based on the Java concurrency framework and create tasks for all non-leaf nodes of the search tree, potentially using the classes `ForkJoinTask` and `ForkJoinPool`[4].

Benchmarks Benchmark your program for $P = 1, 2, 4, 8, 16, 32$ cores (and potentially more).

[4] See https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html and the documentation of these classes