Parallel Computing
Exercise 1 (April 20, 2021)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above via the Moodle interface. The result is submitted 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`/`.cpp`/`.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 develop a C/C++ program, the tool `valgrind`[1] is useful to debug invalid memory accesses; this package is included in many GNU/Linux distributions (Debian: `apt-get install valgrind`).

Please be prepared to give a short (10 min) presentation of your results on April 27; you will be notified by April 21 whether such a presentation is requested from you.

[1] http://valgrind.org/

Alternative A: Matrix Inversion by Gauss-Jordan Elimination

Given a regular matrix $A = (a_{ij})$ with n rows and n columns, our goal is to compute the *inverse* of A, i.e., that matrix $B = (b_{ij})$ with n rows and n columns that satisfies $A \cdot B = I$ where I is the identity matrix:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots \\ b_{n1} & \dots & b_{nn} \end{pmatrix} = \begin{pmatrix} 1 & \dots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \dots & 1 \end{pmatrix}$$

This problem can be solved by *Gauss-Jordan Elimination*[2], a variant of *Gaussian Elimination* that transforms in a sequence of steps the matrix $(A|I)$ with n rows and $2n$ columns into the corresponding matrix $(I|B)$:

$$\left(\begin{array}{ccc|ccc} a_{11} & \dots & a_{1n} & 1 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} & 0 & \dots & 1 \end{array}\right) \rightsquigarrow \dots \rightsquigarrow \left(\begin{array}{ccc|ccc} 1 & \dots & 0 & b_{11} & \dots & b_{1n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & 1 & b_{n1} & \dots & b_{nn} \end{array}\right)$$

The algorithm proceeds in n iterations where in iteration i, for $i = 1, \dots n$, the element a_{ii} serves as the *pivot* element: we divide line i by a_{ii} and subtract from every row $j \neq i$ the multiple a_{ji}/a_{ii} of row i; thus column i gets 1 in row i and 0 in every row $j \neq i$. Please note that A is already in diagonal form in every column $j < i$ such that it is thus not necessary to consider this part any more. For instance, for $n = 6$ and $i = 4$, we have the following situation:

$$\left(\begin{array}{cccccc|cccccc} 1 & 0 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 1 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 1 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & a_{4,4} & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{array}\right)$$

While Gauss-Jordan Elimination is typically not used when the matrix coefficients are floating point numbers (here mainly iterative methods are used for determining approximate solutions), it may play a 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 $\mathbb{Z}/p = \{0, 1, \dots, p-1\}$ 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:

[2] https://de.wikipedia.org/wiki/Inverse_Matrix#Gau%C3%9F-Jordan-Algorithmus
https://en.wikipedia.org/wiki/Gaussian_elimination#Finding_the_inverse_of_a_matrix

```
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 a randomly generated matrix A and $p = 982451653$ ($\approx 2^{30}$).

You may construct a "straight-forward" version of the algorithm that aborts with a corresponding message, if matrix A is not regular. Since arithmetic is exact, any non-zero coefficient may serve as a pivot element in the diagonalization (i.e., is not necessary to take the element with the maximum absolute value).

Demonstrate the correctness of your program by inverting a random 4×4 matrix 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.

Please note that the input of the algorithm is A and its output of B; it is the task of the algorithm to convert A into the intermediate form $(A|I)$ and $(I|B)$ into B; the times for these conversions therefore have to be attributed to the algorithm.

Parallel Program (Basic Version)

In iteration i of the outermost loop, all coefficients of $(A|I)$ in all rows and all columns $j \geq i$ have to be processed; this can be done independently for each coefficient, i.e., in parallel. A simple strategy to increase the task granularity is to utilize the parallelism just across rows: consequently modify the sequential program (if necessary) such that the iterations of the loop that runs over 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 a block of 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`[3] 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 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`[4].

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/ExecutorService.html#invokeAll(java.util.Collection)

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

Problem B: Solving the 15 Puzzle by A* Search

We are given a reachable state of the *15 puzzle*[5], i.e., 15 tiles positioned on a 4×4 board as derived by some sequence of moves from the original state. Our goal is to compute a sequence of moves that transforms the puzzle back into its original state (which we call the "goal"). For instance:

$$\begin{bmatrix} 15 & 2 & 1 & 12 \\ 8 & 5 & 6 & 11 \\ 4 & 9 & 10 & 7 \\ 3 & 14 & 13 & \end{bmatrix} \rightsquigarrow \begin{bmatrix} 15 & 2 & 1 & 12 \\ 8 & 5 & 6 & 11 \\ 4 & 9 & 10 & \\ 3 & 14 & 13 & 7 \end{bmatrix} \rightsquigarrow \dots \rightsquigarrow \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & \end{bmatrix}$$

In every state of the puzzle, we may slide a tile that borders the "hole" in horizontal or vertical direction into the position of the hole; thus in a given state 2–4 moves are possible:

$$\begin{tikzcd}
 & \begin{bmatrix} 15 & 2 & 1 & 12 \\ 8 & 5 & 6 & 11 \\ 4 & 9 & 10 & \\ 3 & 14 & 13 & 7 \end{bmatrix} \arrow[dl] \arrow[d] \arrow[dr] & \\
\begin{bmatrix} 15 & 2 & 1 & 12 \\ 8 & 5 & 6 & \\ 4 & 9 & 10 & 11 \\ 3 & 14 & 13 & 7 \end{bmatrix} & \begin{bmatrix} 15 & 2 & 1 & 12 \\ 8 & 5 & 6 & 11 \\ 4 & 9 & & 10 \\ 3 & 14 & 13 & 7 \end{bmatrix} & \begin{bmatrix} 15 & 2 & 1 & 12 \\ 8 & 5 & 6 & 11 \\ 4 & 9 & 10 & 7 \\ 3 & 14 & 13 & \end{bmatrix}
\end{tikzcd}$$

All possible moves of the puzzle may be therefore organized as a directed graph whose root node is the given initial state; every node has as its successors the states resulting from all possible moves in that state (since the same state may be reached from multiple predecessors, the graph is not necessarily a tree; since by a number of moves we may reach a previous state again, the graph also contains cycles). The goal is to find a path in the graph from the root to the goal node.

Sequential Algorithm For problems such as the given one, the *A* algorithm*[6] implements an effective search heuristics on the basis of two data structures (see Algorithm 1):

- A map *info* that records for every node the minimal *depth* d known so far (the minimal number of moves performed to reach the current puzzle state from the initially given one) and a lower *bound* b for its distance to the desired goal (the original puzzle state). A suitable representation of *info* is that of a "map" or "dictionary".
- A set *open* that records all newly encountered nodes. Operations required on this set are a lookup of the node with minimum value $d + b$ (this node most likely is on the shortest path from the root towards the goal), the removal of this node from the set, and the addition of new nodes. A suitable representation of *open* is that of a "priority queue".

An admissible lower bound $bound(n)$ for the distance of node n to the original situation is the sum of the *Manhattan distances*[7] of the current positions of all tiles from their original positions

[5] https://en.wikipedia.org/wiki/15_puzzle
[6] https://en.wikipedia.org/wiki/A*_search_algorithm
[7] https://en.wikipedia.org/wiki/Taxicab_geometry

Algorithm 1 A* algorithm

```
function A*($root$)        ▷ returns distance of goal node from root ($\infty$, if goal is not reachable)
    $info[root] \leftarrow \langle 0, bound(root) \rangle$; $open \leftarrow \{root\}$
    while $open \neq \emptyset$ do
        choose $node \in open$ with minimal $d + b$ where $\langle d, b \rangle = info[node]$
        $\langle d, b \rangle \leftarrow info[node]$
        if $node$ is goal then return $d$
        $open \leftarrow open \backslash \{node\}$
        for $next \in successors(node)$ do
            $\langle d', b' \rangle \leftarrow info[next]$
            if $\langle d', b' \rangle$ is undefined or $d + 1 < d'$ then
                $info[next] \leftarrow \langle d + 1, bound(next) \rangle$; $open \leftarrow open \cup \{next\}$
            end if
        end for
    end while
    return $\infty$
end function
```

(here the hole is *not* considered as a tile); the Manhattan distance of two positions is simply the sum of the absolute values of the the Cartesian coordinate differences.

Algorithm 1 returns the number of moves performed to reach the goal. However, if we store in *info* also the predecessor of each node, we may reconstruct the path from the goal to the root.

Parallel Algorithm

The A^* algorithm is guaranteed to determine a path of minimal length; for the parallel version, however, we contend ourselves with an approximation of the algorithm that does not necessarily determine the shortest path but is more scalable. This parallel version runs in two phases:

1. For a desired thread number T, the algorithm runs sequentially an A^* search until either the goal is encountered or the set *open* contains at least T nodes with minimal $d + b$, i.e., we have at least T candidates that equally likely sit on an optimal path towards the goal.

2. The main thread starts T auxiliary threads each of which receives from *open* a candidate node n with minimal value $d + b$. Each auxiliary thread creates a local copy of the map *info* of the main thread and creates a local version of *open* that is initialized as $\{n\}$. All threads then run in parallel a sequential A^* search using their local versions of *info* and *open*; the first thread that encounters the goal delivers its result to the main thread and notifies all other threads which subsequently terminate.

The auxiliary threads operate independently on their own data and thus do not require synchronization, which is beneficial for the performance of the execution. Howveer, this comes at the price that the result returned by the first thread does not necessarily represent the shortest path.

Furthermore, the speedup that is actually achieved by a parallel execution depends on how “lucky” a thread is to find from its starting node n a path to the goal; the execution may result in a superlinear speedup, but also in no speedup at all, or even a slow-down; this depends on the trade-off between performing multiple searches in parallel and performing these searches in a sub-optimal way (however, very often substantial speedups can be observed).

Implementation First, implement in C/C++ or Java a sequential solution with a function *seqSolve*(S, N) which solves a random puzzle. This puzzle is generated by using a random number generator with seed S: from the original puzzle state, it performs N random moves and thus derives the initial state. Using as S the current time, you can generate random puzzles. If you print out S, you can later reproduce the the same initial state by using the same S again. Demonstrate the correctness of your solution by depicting the sequence of moves for some solution (for this, store in *info* for every node also its predecessor on the path towards the root; for running larger examples, however, you may omit that information in order to save space).

For an implementations in Java, the generic classes `HashMap`/`TreeMap` and `PriorityQueue` may be suitable to implement *info* and *open*; in C++, the template classes `unordered_map` and `priority_queue` are suitable. In plain C, you either have to implement your own data structures or use some open source library[8]. In any case, you may use the attached source code `Puzzle15.java` as an inspiration for your own solution.

Then implement a parallel solution with a function *parSolve*(S, N, T) where the parameter T denotes the number of threads. You may also provide additional parameters such as a parameter M that denotes the minimum number of nodes in *open* that must have the same minimal path length estimation $b + d$ before starting the parallel search (for small values of T it may be advantageous to use $M \gg T$ to avoid the waste of threads by poor initial estimations of path lengths).

If you implement your solution in OpenMP, you can use the pragma

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

to create T worker threads. If you implement your solution in Java, you may directly create T threads or use features of the Java concurrency framework, as desired. Please note that for some input instances, the algorithm may require a lot of memory; if the heap size is too small, garbage collection times will dominate the execution time and the algorithm will not show speedups. Therefore choose by the Java option `-XMs`M the largest heap size that fits into the memory; e.g., on Zusie $M = 100g$ (100 GB) lets the heap occupy most of the blade memory. Benchmark your program for $P = 1, 2, 4, 8, 16, 32$ cores.

Please note that that the run time of the algorithm is very sensitive to the length of the shortest path from the given initial situation to the root; for randomly generated initial situations, this length is $\simeq 50$, but it it may become $\simeq 80$ in the worst case. A path of length 48 may be detected in some milliseconds, while a path of length 50 may already take hundreds of seconds, and the computation of a path of length 52 might take an extremely long time and ultimately run out of memory. Thus perform many runs until you have at least three examples with reasonable computation times (e.g., a small one that runs 10–20 seconds, and two significantly larger ones). Please note that you have to perform $N \approx 250$ random moves to get an initial situation whose minimum distance from the root is $\simeq 50$ (because a move may return to a previous puzzle state).

[8]For instance: https://github.com/yudi-matsuzake/libgenerics