Parallel Computing
Exercise 5 (Open End)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

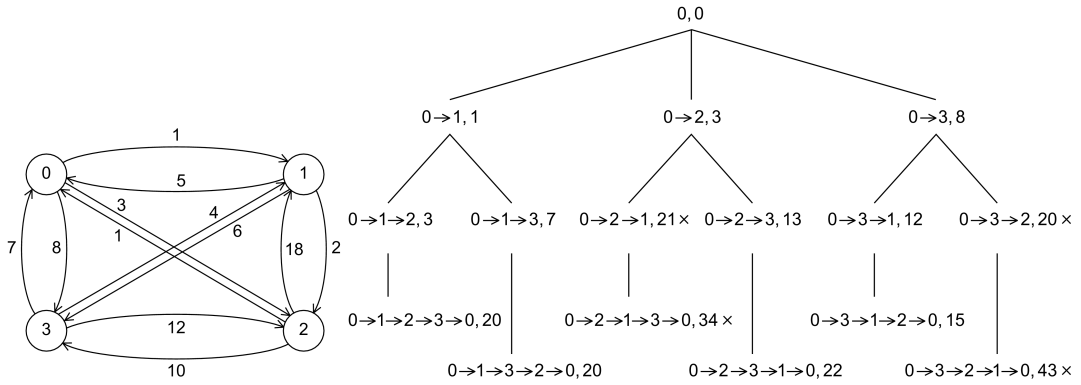
The result is to be submitted by the deadline stated above via the Moodle interface. **The result has to be individually elaborated (no collaboration).**

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 files of the sequential and of the parallel program.

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).

Your task is to implement a sequential version of this program and *one* of the parallel variants described below.

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 5 min, respectively.

Variant A: 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.

In OpenMP, the parallel program can be implemented by parallel blocks and critical sections. In Java, the parallel program can be implemented either by explicitly created threads and synchronization on objects or by using the concurrency API and the Fork/Join framework.

Benchmark Evaluation 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`).

Benchmark your program for $P = 1, 2, 4, 8, 16, 32$ cores.

Variant B: Parallel Program in MPI

Implement a distributed version of the parallel program in MPI. Here the root process 0 serves as a manager of the tasks to be distributed among additional P worker processes:

- The root manages the global pool of paths from which each worker may query new work, i.e., the worker receives a partial path which it has to extend in all possible ways for a potential optimum solution using a local pool of paths.
- Whenever a worker queries the manager for a partial path, the manager provides as part of its answer the length of the shortest cycle found so far which the worker may use to prune its search.
- Whenever a worker finds a potentially shorter cycle, it informs the manager about this cycle and receives as an answer the length of the current shortest cycle which it may subsequently use for pruning.
- If the manager has only one partial path left in its pool, it extends this path in all possible ways such that subsequent queries by multiple workers may be addressed.
- If the manager runs out of work; it informs each client upon its next request about this fact, such that the clients may subsequently terminate; when all clients have been informed, the manager may terminate.

For the implementation, please note that, if a matrix A of dimension $M \times N$ with values of type T is to be passed (respectively broadcast/scattered/gathered) among processes, you have to make sure that A is represented by a contiguous block in memory. This can be achieved either by

[1] http://valgrind.org/

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]`).

Benchmark Evaluation Benchmark the MPI program as in Variant A; 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.