Introduction to Parallel and Distributed Computing Exercise 1 (April 25)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.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 single PDF (`.pdf`) file with
  - a cover page with the title of the course, your name, Matrikelnummer, and email-address,
  - a section with the source code of the program benchmarked, the output of the parallelizing compiler, and an explanation of the output,
  - a section with the raw data of the benchmarks,
  - a section with a summary table and graphical diagrams of the benchmarks.
- the source (`.c`) file(s) of the programs.

Exercise 1: Automatic Parallelization and Benchmarking

Algorithm The "all-pairs shortest paths" problem is to compute, for a weighted directed graph and every pair of vertices in that graph, the length of the shortest path from one vertex to the other one (this length is considered ∞, if there is no such path).

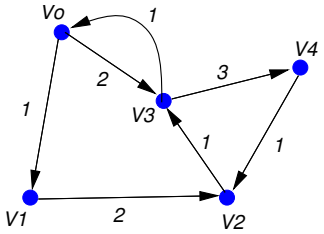
In more detail, we assume that the graph has $N > 0$ vertices and is represented by a $N \times N$ "weight matrix" W where for all vertex indices i, j with $0 \leq i, j < N$ we have

$$W[i][j] = \begin{cases} 0 & \text{if } i = j \\ w > 0 & \text{if } i \neq j \text{ and there is an edge of weight } w \text{ from } i \text{ to } j \\ \infty & \text{if } i \neq j \text{ and there is no edge from } i \text{ to } j \end{cases}$$

The goal is then to compute a "distance matrix" D where for all vertex indices i, j we have

$$D[i][j] = \begin{cases} 0 & \text{if } i = j \\ d > 0 & \text{if } i \neq j \text{ and the shortest path from } i \text{ to } j \text{ has length } d \\ \infty & \text{if } i \neq j \text{ and there is no path from } i \text{ to } j \end{cases}$$

This problem is demonstrated by the following sample graph:



W	v_0	v_1	v_2	v_3	v_4
v_0	0	1	∞	2	∞
v_1	∞	0	2	∞	∞
v_2	∞	∞	0	1	∞
v_3	1	∞	∞	0	3
v_4	∞	∞	1	∞	0

$\rightsquigarrow$

D	v_0	v_1	v_2	v_3	v_4
v_0	0	1	3	2	5
v_1	4	0	2	3	6
v_2	2	3	0	1	4
v_3	1	2	4	0	3
v_4	3	4	1	2	0

A simple algorithm solving this problem proceeds by repeated "multiplications"

$D \leftarrow W$
for n **from** 1 **to** $N - 2$ **do**
  $D \leftarrow D \times W$
end for

(please note that D is updated *after* the "product" $D \times W$ has been computed). When n multiplications have been performed, the matrix D shall describe the shortest paths with at most $n + 1$ edges; therefore, since a shortest path cannot have more than $N - 1$ edges, at most $N - 2$ multiplications are sufficient. Here a "multiplication" $C \leftarrow A \times B$ is is defined as

for i **from** 0 **to** $N - 1$ **do**
  for j **from** 0 **to** $N - 1$ **do**
    $C[i, j] \leftarrow A[i, j]$
    for k **from** 0 **to** $N - 1$ **do**
      $C[i, j] \leftarrow \min(C[i, j], A[i, k] + B[k, j])$
    end for
  end for
end for

For all i, j this subalgorithm attempts to improve the shortest path from i to j found so far by an alternative path which runs from i through some intermediate node k to j: it thus compares the length $C[i, j]$ of the shortest path found so far with the length $A[i, k] + B[k, j]$ of every alternative path and selects the shorter of the two.

The algorithm can be further improved by repeated "squaring" that only requires $\lceil \log_2(N-1) \rceil$ "multiplications":

$D \leftarrow W$
for n **from** 1 **to** $\lceil \log_2(N-1) \rceil$ **do**
  $D \leftarrow D \times D$
end for

Our goal is to implement this algorithm with a parallel version of the "squaring" $D \leftarrow D \times D$.

Implementation Use for this exercise the default installation of the Intel compiler `icc` (`module load intelcompiler`) and compile with options `-O3 -lrt`.

Sequential Program You may use as the starting point the given sample program `matmult.c` for matrix multiplication and adapt it to a solution of the "all pairs shortest paths" problem.

You may use for the matrices statically allocated arrays of fixed dimension (`double W[N][N];`). The arrays should be declared globally (they are then allocated in the data segment of the process), not locally in a function (this allocates the matrix on the process stack, which may let the stack overflow and the program crash). Please note that N must then denote a compile-time constant.

Alternatively, you may also allocate a matrix on the heap as a linear array (`double *W = malloc(n*n*sizeof(double));`); here n may be a variable value. Please note that then the matrix is accessed at index i, j by the reference `W[i*n+j]`.

Do *not* use a representation of the matrix as an array of array pointers (`double **W`); this may considerably slow down the execution and also let the automatic parallelization fail.

You may initialize W with random values in an interval $]0, M]$ for some maximum M and use e.g. -1 to represent ∞.

Instrumentation Instrument the source code of the program to measure the real ("wall clock") time spent (only) in the execution of the algorithm (without initialization, input, or output) and print it to the standard output. Determine the wall clock time (in ms) elapsed for a computation as follows:

```
#include <time.h>
...
struct timespec t1, t2;
clock_gettime(CLOCK_REALTIME, &t1);
...
clock_gettime(CLOCK_REALTIME, &t2);
int t = (t2.tv_sec-t1.tv_sec)*1000+(t2.tv_nsec-t1.tv_nsec)/1000000;
```

Benchmark Sequential Execution Compile the program with the Intel compiler and optimization option `-O3`. Benchmark the program for two significantly different values of N; for at least one value of N the execution shall run at least one minute.

Benchmark Parallel Execution Compile the program with the Intel compiler and parallelization option `-O3 -parallel -par-report2`. Investigate and explain the output of the compiler (you may also use `-par-report3` for more detailed information).

Benchmark the parallel program for the same values of N chosen in the sequential program (and the same contents of $W!$) with $T = 1, 4, 8, 16, 32, 64$ threads binding each thread to a physical core, using environment variable settings such as

```
export OMP_DYNAMIC=FALSE
export OMP_NUM_THREADS=32
export GOMP_CPU_AFFINITY="256-287"
```

Be sure to use for your benchmark unoccupied processors of the machine.

Repetition Repeat all benchmarks 5 times and collect all results. 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 4 8 16 32 64 ; 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 (explaining which node affinity strategy you have used).

Summary Construct a summary table that reports for each value of N

- the average execution time of three runs of the sequential program (excluding those two runs that took shortest and longest) and

and for each value of N and T

- the average execution time of three runs of the parallel program (excluding those two runs that took shortest and longest),
- the average speedup, i.e., the average sequential execution time divided by the average parallel execution time, and
- the average efficiency, i.e., the average speedup divided by the number of threads used.

Illustrate the execution times, speedups, and efficiencies of the parallel program also by graphical diagrams; use both a linear scale and a logarithmic scale for the axes.