Parallel Computing
Exercise 1 (April 12)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above via the Moodle interface. If the assignment has been elaborated in a collaboration of two students, only one of them shall upload the assignment (indicating of course on the cover page the collaboration partner).

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 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`).

Benchmark your sequential program and your parallel program for at least three inputs of significantly different sizes (the sequential computation for at least one of the inputs must run at least one minute) and for $T = 1, 2, 4, 8, 16, 32$ threads. 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 (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.

Please be prepared to give a short (10 min) presentation of your results on April 17; you will be notified by April 13 whether such a presentation is requested from you.

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`).

[1] http://valgrind.org/

Problem A: Scale-free Networks and Google PageRank

The goal of this problem is to develop a program for

1. computing a scale-free network (representing a web of linked pages), and
2. assigning to each node in this network (i.e., to each page in the web) a rank according to the Google PageRank algorithm.

Both subproblems are to be solved by parallel computation.

Scale-Free Network The degree of a scale-free network (approximately) follows a power distribution, i.e., every node in the network has probability $P(k) = k^{-\gamma}$ to have k links to other nodes (for some $\gamma > 0$). It can be observed that many real-life networks have this property, in particular the Internet and the World-Wide Web.

In the following, we describe a variant of the Barabási–Albert (BA) algorithm[2] for generating a random scale-free network with N nodes. We assume that the graph is represented by an array of N nodes where each node holds its *outdegree* (the number of links to other nodes), its *indegree* (the number of links from other nodes) and (a pointer to) an array *inrefs* that contains the indices of all nodes that have links to this node (this pointer is initially null; the array is allocated on demand and grows dynamically).

The BA algorithm proceeds in two steps:

1. Generate a seed network with nodes $0, \ldots, K-1$ where $K \ll N$
2. Extend the seed network to the scale-free network by adding nodes $K, \ldots, N-1$

In the first step, we will use $K := (\ln N)^2$ and link every node i to every node $j < i$ (thus every node i receives $outdegree(i)$ and $indegree(i) = N - i$; the nodes with small indices and thus initially high indegrees will become the "hubs" of the subsequent scale-free network).

The second step extends the seed network to the full scale-free network by the "preferential attachment" strategy shown in Algorithm 1. Here every new node i is linked by the j-loop to M previously created nodes; the probability of a node k to receive a new link from i is proportional to one plus the number of incoming links that node k already has (the base value one ensures that also nodes with zero incoming links have a chance). For this purpose, the algorithm assigns to each node k a "weight" of one plus the number $indegree(k)$ of incoming links; it also determines the current total weight of the network as the sum *total* of the number i of nodes created so far and of the number #*links* of links created so far (which has to be tracked); the total weight is thus the sum of the node weights. Then the algorithm determines a random number r with $0 \le r < total$ and finds the smallest node k with $\sum_{k'=0}^{k}(indegree(k') + 1) > r$; this node is thus chosen with a probability that equals its relative weight with respect to the total weight of the network.

A parallel variant of the algorithm may proceed by splitting the iterations of the i-loop into a sequence of intervals ("blocks"). All iterations within a block are processed in parallel to determine new links (without updating the graph yet); after a block has been processed, the

[2] https://en.wikipedia.org/wiki/Barab%C3%A1si%E2%80%93Albert_model

Algorithm 1 Preferential Attachment

```
for i from K to N − 1 do
    create node i
    total ← i + #links
    for j from 0 to M − 1 do
        choose random r ∈ [0, total[
        sum ← 0
        k ← (−1)
        repeat
            k ← k + 1
            sum ← sum + 1 + indegree(k)
        until sum > r
        link node i to node k
    end for
end for
```

graph is updated with the new links and the computation proceeds to the next block. This naturally yields an outer loop whose index i indicates the start of a block and an inner loop whose index $i0$ indicates an index within that block. Each iteration $i0$ writes into the slot $L[i0][j]$ of a shared array L (that holds pointers to separate arrays for each value of i_0) the node k to which node $i + i0$ is to be linked by iteration j of the corresponding loop. After a block has been processed, the graph is updated by creating, for every $i0$ and j, a link from node $i + i0$ to node $k = L[i0][j]$.

The size of the parallel execution block should be on the one side large enough to ensure that there is sufficently much work for each thread; on the other side it must be also significantly smaller than N to ensure that by multiple iterations of the other loop the network "in essence" becomes scale-free. As a rule of thumb, choose the smallest block size that does not significantly hurt the performance of the parallel execution. In an OpenMP solution, the inner loop can be executed in parallel by an annotation `#pragma omp parallel for`; in the Java concurrency API, the method `invokeAll` may be used to create the tasks for processing the loop iterations and to wait for the completion of their execution by a previously created thread pool.

A problem arises in the parallel use of the random generator by multiple threads: every random generator has a local state that is updated by the computation of the next random number. This makes the C standard function `rand()` not thread-safe; instead the function `rand_r(`s`)` must be used which receives a pointer s to an integer variable that holds the state to be used and updated by the random number generator. Thus an integer array S is to be allocated and initalized such that each parallel iteration executes `rand_r(S+i0)` with a different state pointer. Likewise, in the Java solution, the class `java.util.concurrent.ThreadLocalRandom` should be used to avoid the sharing of states across multiple threads.

Algorithm 2 PageRank (numerical solution)

$PR(i) \leftarrow 1/N$, for $0 \leq i \leq N-1$
repeat
  $e \leftarrow 0$
  for i **from** 0 **to** $N-1$ **do**
    $pr \leftarrow 0$
    for $j \in inrefs(i)$ **do**
      $pr \leftarrow pr + PR(j)/outdegree(j)$
    end for
    $PR'(i) \leftarrow (1-D)/N + D \cdot pr$
    $e \leftarrow e + (PR'(i) - PR(i))^2$
  end for
  $PR \leftarrow PR'$
until e < E

PageRank Google's PageRank algorithm[3] assigns to every page i in the network the rank

$$PR(i) = (1-D)/N + D \cdot \sum_{j \in inrefs(i)} PR(j)/outdegree(j)$$

Rank $0 \leq PR(i) \leq 1$ denotes the probability that a user reaches page i by a random click (a normalized page rank $0 \leq PR(i) \leq N$ is determined by dropping the factor $1/N$). The *damping factor* D denotes the probability that a user does not stop clicking; usually $D := 0.85$ is assumed.

This equation describes a recursive equation system which can be numerically solved by iterative approximation as shown in Algorithm 2. A parallel version of this algorithm can compute all iterations of the i-loop in parallel (each iteration may indepently compute its local error $e := (PR'(i) - PR(i))^2$ which may be then subsequently reduced to the global error).

In the OpenMP solution both parallel execution and global reduction may be achieved by a single pragma `#pragma omp parallel for ... reduction(+:e)`. In the Java concurrency API, the method `invokeAll` may be used to create the tasks for processing the individual loop iterations and to wait for the completion of their execution by a previously created thread pool.

Benchmarking The absolute execution times, the relative fraction of the times required for network generation and the PageRank computation, and the achievable speedups depend very much on suitable choices of N and M; from preliminary experiments, it seems that $N \in [100000, 200000]$ and $M \in [50, 5000]$ may give acceptable results ("unrealistically large" values of M are needed to get good results for comparatively small values of N). To get a feeling for the dynamic behavior of the algorithm, use debugging output to determine how quickly new pages are linked to the scale-free network and how quickly the page rank computation converges.

When benchmarking the program, determine *separately* the times, efficiencies, speedups of the generation of the scale-free network and of the computation of the page rank; however, also give the combined measures. For the numerical approximation of the page rank choose a suitable error bound not less than 10^{-6}.

[3] https://en.wikipedia.org/wiki/PageRank

Problem B: The Number of Satisfying Assignments of a Formula

The goal of this problema is to develop a parallel program for determining the number of satisfying assignments of a propositional formula in conjunctive normal form.

Preliminaries

A *propositional formula* F in *conjunctive normal form* is a conjunction of clauses where each *clause* C is a disjunction of literals and each *literal* L is a positive or negative occurrence of a propositional *variable* V. F is thus formed according to the following extended BNF grammar:

$$\begin{array}{lcl} F & ::= & [\ C\ (\wedge\ C)^*\ ] \\ C & ::= & [\ L\ (\vee\ L)^*\ ] \\ L & ::= & V\ \mid\ \neg V \end{array}$$

An example of such a formula is

$$(x \vee \neg y \vee z) \wedge (\neg x \vee y) \wedge (y \vee \neg z).$$

which can be also represented by the set

$$\big\{\{x, \neg y, z\}, \{\neg x, y\}, \{y, \neg z\}\big\}.$$

If a clause has both a positive occurrence V and a negative occurrence $\neg V$ of the same variable V, it is equivalent to "true" and can be discarded; we thus may assume that no clause has such conflicting occurrences.

An *assignment* A is a set of n literals such that every variable V occurs in A either in positive or in negative form: $V \in A$ indicates that variable V is assigned the truth value "true" while $\neg V \in A$ indicates that it is assigned the truth value "false". An assignment A *satisfies* a formula F, if the formula becomes true when all the variables are replaced by the truth values indicated by the assignment. In more detail, assignment A satisfies clause C, if there exists a literal L in A that also occurs in C; A satisfies F, if it satisfies every clause in C. For instance, above formula is satisfied by the assignment $A = \{x, y, z\}$.

In order to determine whether there exists any satisfying assignment for a formula F with n variables, the DPLL algorithm[4] depicted (in a simplified form as) Algorithm 3 may be applied: if F is empty, the formula represents "true" and is thus satisfiable; if F has an empty clause, this clause represents "false", and the formula is not satisfiable. Otherwise, we choose some literal L that occurs in F and apply the algorithm recursively, first to formula $F \wedge L$ and, if necessary, also to formula $F \wedge \neg L$. Each of these formulas has $n - 1$ variables, because the computation of $F \wedge L$ removes every negative occurrence of L from F and removes every clause C from F that has a positive occurrence of L (dually for the computation of $F \wedge \neg L$). Thus the recursive call DPPL($F \wedge L$) determines whether there exists any satisfying assignment for F which sets L to "true"; the recursive call DPPL($F \wedge \neg L$) determines whether there exists any satisfying assignment for F which sets L to "false". The execution of the algorithm can be described by a binary tree with 2^n leaves that represent all possible assignments; it thus represents an exhaustive search for a satisfying assignment in the space of all possible assignments.

[4] https://en.wikipedia.org/wiki/DPLL_algorithm

Algorithm 3 DPLL Algorithm (simplified)

```
function DPLL(F, n)                                   ▷ Formula F has n variables
    if F is empty then
        return true
    else if F has an empty clause then
        return false
    else
        choose literal L in F
        return DPPL(F ∧ L, n − 1) or DPPL(F ∧ ¬L, n − 1)
    end if
end function
```

The algorithm can be easily extended to return the *number* of satisfying assignments by the following changes:

1. In first base case, it returns 2^n (if there are still n unassigned variables, there are 2^n possible assignments to these variables).

2. In the second base case, it returns 0 (since there is no assignment).

3. In the recursive case, it returns the sum of the values returned by the two recursive calls.

In the actual implementation of the algorithm by a computer program, a formula F with c clauses in v variables can be represented by a $c \times v$ matrix of elements $\{-1, 0, +1\}$ where $F[i, j]$ has value $+1$, if clause i has a positive occurrence of variable j, value -1, if it has a negative occurrence, and value 0, if variable j does not occur in clause i. Above formula can be thus represented by the matrix

$$\begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & 0 \\ 0 & 1 & -1 \end{bmatrix}.$$

Rather than constructing a new formula matrix F for every recursive invocation of the algorithm, we may pass to every invocation just a triple c', l, a where

- c' is the number of clauses that are still in F,
- l is an array of length c where $l[i]$ is the number of literals in clause i of F (the special value $l[i] = -1$ indicates that clause i has been deleted from F; there are $c - c'$ such lines);
- a represents the *partial assignment* computed so far: it is a vector of length v of values $\{-1, 0, +1\}$ (initially 0 everywhere): $a[i] = +1$ indicates that variable i has been assigned the truth value "true", $a[i] = -1$ indicates that it has been assigned the truth value "false"; $a[i] = 0$ indicates that it has not been assigned a truth value yet.

All necessary operations of the algorithm may be expressed in terms of these three variables.

Sequential Solution Implement a sequential solution that solves the problems for parameters c, v, d, s, where c is the number of clauses, v is the number of variables, d is a density value between 0 and 1, and s is an integer that represents the seed for the random number generator. The program shall generate a random formula matrix of dimension $c \times v$ of which a fraction d is filled with random literals using seed s for the random number generator (to save space, use type `byte` for the elements of the matrix). Demonstrate the correctness of your solution by showing the result for some (small) formula and then benchmark the solution for three different inputs of reasonable size.

The parallel solution shall also use a parameter T that representes the number of concurrent threads to be applied for the solution. Demonstrate also the correctness of the parallel solution by showing that it computes the same result as the sequential one.

Java-Based Parallel Solution The program may be based on the Java concurrency framework using the classes `ForkJoinTask` and `ForkJoinPool`[5].

However, you may also apply the strategy sketched for the OMP-based solution below which lets a fixed set of worker threads operate on a shared stack (this may or may not be more efficient).

OMP-based Parallel Solution The program creates T worker threads that simultaneously process the space of possible solutions:

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

The space of possible solutions to be processed is represented by:

- a global stack of partial assignments (access to this stack is protected by a critical section or a lock variable);
- for every thread a local stack of partial assignments (since the thread has exclusive access to its stack, the lockal stacks need not be protected).

Initially, the global stack holds only the empty assignment $a = []$ and all local stacks are empty.

Each thread primarily processes assignments from its local stack by a doubly nested loop:

1. in the outer loop, the thread pops a partial assignment a from its local stack and determines from F, n, a the values l and c with which it starts the execution of the inner loop;
2. in the inner loop, the thread proceeds as follows:
   a) the thread performs the checks for the "base cases" and only proceeds with the loop, if they are not successful;
   b) for one of the "recursive" cases, the thread pushes a new partial assignment a' to its local stack;
   c) the other "recursive" case with assignment a'' is processed by the thread itself in the next iteration of the loop;

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

3. the thread leaves the inner loop when it reaches the "base case" where it can determine the number of total assignments arising from the current partial assignment a''; the thread correspondingly increments a local counter that keeps track of the number of satisfying assignments that the thread has found so far.

If the local stack becomes empty, the thread pops instead an assignment a from the global stack and continues with processing this assignment. However, if by the removal of a the global stack becomes empty, the thread extends a by setting $t \geq 1$ previously unassigned variables to all possible combinations of their truth values; it thus generates 2^t new assignments from which the thread keeps itself one for further processing and leaves the other ones on the global stack for processing by the other threads (thus $2^t - 1 \geq T$ is advisable).

Furthermore, to prevent an early permanent idling of some threads, every busy thread checks from time to time (e.g. after it has popped a certain number of assignments from its local stack), whether the global queue is empty. If yes, it fills the global stack with the $2^t - 1$ oldest assignments (the assignment with the fewest variables set) from its local stack (the local stack should be thus implemented with the help of two index variables as a cyclic queue).

If a thread runs out of assignments to process and also finds the global stack empty, it increments a protected shared variable that denotes the number of threads that are waiting on the global stack for assignments. If this variable indicates that all threads are waiting, the total number of satisfying assignments may be computed as the sum of all local counters and all threads may terminate.

Please note that a local stack may hold at most $n - 1$ assignments (where n is the maximum number of variables in a formula) and that the global stack may only hold at most $2^t - 1$ assignments. Furthermore, since each assignment can be represented by n truth values, each local stack can be represented by an array of $(n - 1) \cdot n$ truth values and the global stack can be represented by an array of $(2^t - 1) \cdot n$ truth values. All these arrays can be allocated in advance; to improve data locality, let each thread allocate its local stack on its own.

Finally, please note that the (exponentially many) satisfying assignments must *not* be actually stored; it is only their number that is to be determined.