Parallel Computing
Exercise 1 (April 14, 2020)

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 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 21; you will be notified by April 16 whether such a presentation is requested from you.

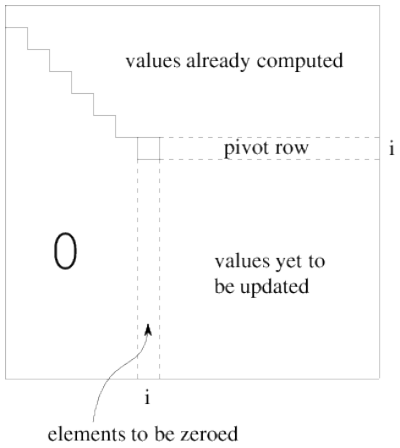
[1] http://valgrind.org/

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 $\hat{A}x = e$ (i.e. all coefficients below the main diagonal of $\hat{A}$ are zero) which has the same solution(s) as the original system.
2. The new system $\hat{A}x = e$ is solved by backward substitution (we determine the solution $x_{N-1} = e_{N-1}$, and substitute the solution in $\hat{A}$ 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 5 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.

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 iteration of the triangulation step, 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; also two threads processing

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

different rows may compete for the same cache lines (cache lines may span across rows leading to "false sharing"). Therefore write another version of the program that adresses these problems:

- To prevent any false sharing, every row is extended by 8 unused `long values` ($8 \cdot 8 = 64$ bytes, the size of a cache line); also it is recommended to store the vector b in the same matrix to avoid false sharing among b (every row of the matrix has thus $n + 1 + 8$ entries).
- To prevent repeated cache line transfers, 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

Problem B: The Number of Satisfying Assignments of a Formula

The goal 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 1 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 1 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. Also please note that different formulas of the same size may lead to different speedups; you may thus want to experiment with different seed values of the random number generator.