Parallel Computing
Exercise 1 (April 22, 2025)

Wolfgang Schreiner
Wolfgang.Schreiner@risc.jku.at

The result is to be submitted by the deadline stated above via the Moodle interface.
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, matriculation number, and email address,
  - 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 OpenMP C/C++ or 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.

Hint In Assignment 3, you will be asked for an MPI solution of this problem in C/C++, you may consider this when choosing the problem and/or the implementation language for this assignment.

Tip If you program in C/C++, the tool `valgrind`[1] is useful to debug invalid memory accesses; this package is included in many Linux distributions (Debian: `apt-get install valgrind`).

Presentation Please be prepared to give a short (10 min) presentation of your results on April 29; you will be notified by April 24 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;
}
```

Please note that for computing multiple modular quotients a_i/b, one may compute once the modular inverse b^{-1} and then compute multiple modular products $a_i \cdot b^{-1}$.

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 and report your experience (in particular whether $B = 1$ is

already optimal). 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/threads (and potentially more).

Parallel Program (Advanced Version)

Most likely the basic version of the 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, which leads 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 (generally rows are distributed to threads in a "round-robin" fashion to ensure that the work load of a thread is not influenced by a particular distribution of data in the matrix).

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 indeed scales 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

Alternative B: Bin Packing

The "bin packing" problem is as follows[5]: we are given a supply of bins of capacity $bsize > 0$. Furthermore, we have $k > 0$ kinds of items which are determined by vectors $size\colon \mathbb{N}_k \to \mathbb{N}$ and $count\colon \mathbb{N}_k \to \mathbb{N}$ where, for every $i \in \mathbb{N}_k$, $size[i] > 0$ is the size of every item of kind i and $count[i] > 0$ is the number of items of that kind. We want to pack items into bins such that the number of bins is minimized, i.e., we want to find a vector $result : \mathbb{N}_{rlen} \to (\mathbb{N}_k \to \mathbb{N})$ with minimal length $rlen$ such that, for every kind $i \in \mathbb{N}_k$, we have $count[i] = \sum_{j \in \mathbb{N}_{rlen}} result[j][i]$ and, for every $j \in \mathbb{N}_{rlen}$, we have $bsize \geq \sum_{i \in \mathbb{N}_k} result[j][i]$.

For instance, given $bsize = 6$, $k = 3$, $size = [1, 2, 3]$, $count = [3, 6, 1]$, the packing $result = [[1, 1, 1], [2, 2, 0], [0, 3, 0]]$ has size $rlen = 3$ (which is minimal).

The problem of deciding whether a collection of items can be packed into a certain number of bins is NP-complete, thus it is unlikely that an algorithm exists that can compute exact solutions of every instance of the bin packing problem with better than exponential complexity (however, there exist fast approximation algorithms also exact algorithms that are often fast). In the following, we will consider a "brute-force" algorithm that solves the problem.

We start by the following algorithm that only computes the size *rlen* of an optimal solution:

```
binpack(count):
  if size(count) = 0 then return 0
  rlen := infinity
  for every bin that is admissible and maximal do
    rlen0 := binpack(count-bin)
    if rlen0 < rlen then rlen := rlen0
  return 1+rlen
```

Here $size(count)$ denotes the total size of all items listed in *count*. A *bin* is *admissible* if $bin \leq count$ and $size(bin) \leq bsize$ ($bin \leq count$ holds if we have, for every kind i, $bin[i] \leq count[i]$; $size(bin)$ is analogous to $size(count)$). Furthermore, *bin* is *maximal* if there is no admissible bin' with $bin < bin'$ (i.e., $bin \leq bin'$ and $bin \neq bin'$). Finally, $count - bin$ denotes the component-wise subtraction of vectors *count* and *bin*.

In a nutshell, this algorithm packs into the next bin as many items as possible in all possible ways and then solves the problem recursively for the remaining items; among all possibilities the one with the minimum number of bins is selected. The process stops when there are no more items left.

We can also formulate this algorithm in a tail-recursive way as follows:

```
binpack(count, rlen0): // global variable rlen
  if size(count) = 0 then
    if rlen0 < rlen then rlen := rlen0
    return
  if rlen0+1 >= rlen then return;
  for every bin that is admissible and maximal do
    binpack(count-bin, rlen0+1)
```

This algorithm uses the global variable *rlen*, which represents the smallest number of bins determined so far; this variable is initialized as "infinity". The algorithm receives an additional parameter *rlen0* which represents the number of bins required for the current packing. Initially called as

[5] In the following, we denote by $\mathbb{N}_k$ the set $\{0, \ldots, k-1\}$ of the k natural numbers less than k.

`binpack(count,0)`, the algorithm recursively calls itself with *rlen0* incremented by one. In the base case, the algorithm tests whether *rlen0* is smaller than *rlen* and, if this is the case, updates *rlen* correspondingly. The recursion terminates prematurely if the value of *rlen0* indicates that the value of *rlen* cannot be improved by the current recursive branch.

This algorithm can be easily augmented to compute the actual packing:

```
binpack(count, rlen0, result0): // global variables rlen, result
  if size(count) = 0 then
    if rlen0 < rlen then { rlen := rlen0; result := result0 }
    return
  if rlen0+1 >= rlen then return;
  for every bin that is admissible and maximal do
    binpack(count-bin, rlen0+1, add(result0,bin))
```

This algorithm now also uses a global variable *result* that represents the best packing computed so far and has an additional parameter *result0* that represents the current partial packing which requires *rlen0* bins. Initially called as `binpack(count,0,[])` with the empty list [ ] of bins, the algorithm adds in every recursive call the chosen bin to the list. In the base case, if a new optimal packing is found, both *rlen* and *result* are updated.

A problem that remains to be solved is how to determine in the loop every possible admissible and maximal bin. In fact, all such bins are printed by the following algorithm:

```
binprint(count, bin, bfree, i):
  if i = k then
    if maximal(bin) then print bin
    return
  max := min(count[i],bfree/size[i])
  if i < k-1 then
    for c from 0 to max do
      binprint(count[i<-count[i]-c], bin[i<-c], bfree-c*size[i], i+1)
  else // i = k-1
    binprint(count[i<-count[i]-max], bin[i<-max], bfree-max*size[i], i+1)
```

Called as `binprint(count,empty,bsize,0)` (where *empty* represents a new vector of length k where all values are 0), the algorithm tail-recursively fills vector *bin* with all possible possible admissible and maximal contents and prints them (in this code, $a[i \leftarrow e]$ denotes a *copy* of a that is identical to a except that at index i it holds element e).

Since this algorithm is tail-recursive, it can be written in this more efficient iterative form:

```
binprint(count, bin, bfree, i):
  loop
    if i = k then
      if maximal(bin) then print bin
      return
    max := min(count[i],bfree/size[i])
    if i < k-1 then
      for c from 1 to max do
        binprint(count[i<-count[i]-c], bin[i<-c], bfree-c*size[i], i+1)
      bin[i] := 0; // case c = 0 handled iteratively
    else // i = k-1
      count[i] := count[i]-max; bin[i] := max; bfree := bfree-max*size[i];
    i := i+1
```

In every branch of this version of the algorithm, one recursive procedure call is replaced by an update to the procedure parameters, which is followed another iteration of a loop; only in the case $i < k - 1$ we have $max - 1$ recursive procedure calls left.

However, we are not interested in printing the bins, we want to use them for the recursive calls of the `binpack` procedure described above. Therefore we use the structure of `binprint` as a blue print for the final version of `binpack` that combines the construction of a new bin with the packing of the remaining items:

```
binpack(count, rlen0, result0, bin, bfree, i): // global variables rlen, result
  loop
    if size(count) = 0 then
      if bfree < bsize then { rlen0 := rlen0+1; result0 := add(result0, bin) }
      if rlen0 < rlen then { rlen := rlen0; result := result0 }
      return
    if rlen0+(if bfree < bsize then 1 else 0)) >= rlen then return
    if i = k then
      if not maximal(bin) then return
      rlen0 := rlen0+1; result0 := add(result0,bin);
      bin := empty; bfree := bsize, i := 0
      continue
    max = min(count[i],bfree/size[i])
    if i < k-1 then
      for c from 1 to max do
        binpack(count[i<-count[i]-c], rlen0+1, copy(result0),
                bin[i<-c], bfree-c*size[i], i+1)
      bin[i] := 0 // case c = 0 handled iteratively
    else // i = k-1
      count[i] := count[i]-max; bin[i] := max; bfree := bfree-max*size[i];
    i := i+1
```

This procedure is called as `binprint(count,0,[],empty,bsize,0)`. Its core structure is a loop, but one branch of the loop body may give rise to a number of recursive calls of the procedure (admittedly the code is a bit complex, but the derivation given above should make it understandable). This algorithm shall serve as the basis of a sequential and parallel implementation.

As an optimization, this algorithm is to be called only after already some non-optimal packing has been constructed by some simple but fast algorithm. For instance, we may "greedily" scan the list of items and either place the next item in the list into the current bin or, if the item does not fit into the bin, close this bin and start a new one. In this way, we may initialize *result* and *rlen* before calling the optimal algorithm which will then only consider packings of length less than *rlen*.

Sequential Program

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 bins of size *bsize* and n items distributed among k kinds.

For randomly assigining item numbers to kinds, you may split the interval $[0, n]$ into k subintervals by generating $k - 1$ different random numbers in $[1, n - 1]$ and use the sizes of the subintervals as the item numbers. For distributing the sizes of the items, you may choose k different random numbers in

the interval $[1, 1 + bsize/2]$ such that no too many bins are forced to hold just single items. However, you may also apply other strategies for the distribution of item numbers and sizes.

It is a good idea to parameterize the program with the seed of the random number generator (such that runs can be repeated for the same input, -1 may indicate to use the current time as the seed) and the number of threads to be used for later parallel computation (0 may indicate sequential execution). Please let the program check that indeed a correct packing has been constructed (all items are placed in some bin and the content of a bin does not exceed the size of the bin). A possible program output for the sequential algorithm might thus be:

```
BinPack 1737068196857 0 30 6 12
============================
seed:1737068196857, threads:0, items:30, kinds:6,
number*size:[4*1,4*2,7*3,3*5,3*6,9*7], bin size:12
Greedy bin packing (0 ms): 15 bins.
[4,4,0,0,0,0],[0,0,4,0,0,0],[0,0,3,0,0,0],[0,0,0,2,0,0],[0,0,0,1,1,0],[0,0,0,0,2,0],
[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1],
[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1]
The packing is correct.
Optimal bin packing (15210 ms, sequential computation): 11 bins.
[1,1,1,0,1,0],[2,0,1,0,0,1],[1,0,1,0,0,1],[0,1,1,0,0,1],[0,1,1,0,0,1],[0,1,1,0,0,1],
[0,0,1,0,0,1],[0,0,0,1,0,1],[0,0,0,1,0,1],[0,0,0,1,0,1],[0,0,0,0,2,0]
The packing is correct.
```

In contrast, the program output for the parallel algorithm (see below) might look as follows:

```
BinPack 1737068196857 4 30 6 12
============================
seed:1737068196857, threads:4, items:30, kinds:6,
number*size:[4*1,4*2,7*3,3*5,3*6,9*7], bin size:12
Greedy bin packing (0 ms): 15 bins.
[4,4,0,0,0,0],[0,0,4,0,0,0],[0,0,3,0,0,0],[0,0,0,2,0,0],[0,0,0,1,1,0],[0,0,0,0,2,0],
[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1],
[0,0,0,0,0,1],[0,0,0,0,0,1],[0,0,0,0,0,1]
The packing is correct.
Optimal bin packing (5707 ms, parallel computation with 4 threads): 11 bins.
[1,2,0,0,0,1],[0,0,2,0,1,0],[0,0,0,1,0,1],[0,0,0,1,0,1],[0,0,0,1,0,1],[0,0,0,0,2,0],
[2,0,1,0,0,1],[1,0,1,0,0,1],[0,1,1,0,0,1],[0,1,1,0,0,1],[0,0,1,0,0,1]
The packing is correct.
```

The concrete input may significantly influence the runtime of the algorithm; thus benchmark the program for (at least) two inputs of different sizes that let the sequential program run at least 1 min and at least 3 min, respectively.

Parallel Program

Implement a shared memory version of the parallel algorithm in Java or in OpenMP C/C++. For this, consider the following points:

- The main source of parallelism are the $max - 1$ recursive calls of `binpack` which can be executed asynchronously in parallel (with each other and with the main procedure) such that the search tree for an optimal packing is investigated in parallel. Please note, however, that

parallel executions of `binpack` have to be performed on *copies* of those array arguments that are updated by these calls.

- The parallel program may (or may not) perform faster if the recursion tree is "cut off" at a certain depth, i.e., recursive calls beyond that depth are executed sequentially. Please experiment with some cut-off strategies and report your experience.
- The only point of synchronization among parallel tasks is the access to the global (shared) variables `rlen` and `result` which must be appropriately protected by the application of a suitable synchronization construct. However, it is possible that a frequent comparison with variable `rlen` (in order to prematurely terminate the execution, see the algorithm) becomes a bottleneck. Therefore it may be a good idea to perform such a comparison without synchronization, either on a potentially outdated value of the variable (which may lead to superfluous executions), or using the keyword `volatile` in the declaration of `rlen` (which ensures without synchronization that the latest value of the variable is read); please experiment with some variable access strategies and report your experience.

  However, for actually updating `rlen` and `result`, the comparison has to be indeed performed (by the use of a suitable synchronization construct) atomically together with the updates.

Below some suggestions are given how to implement the parallel solution in various parallel programming frameworks.

Java Solution The program may be based on the Java concurrency framework, using the classes `ForkJoinTask` and `ForkJoinPool`[6]. In order to recognize the termination of the parallel solution, the body of the recursive function should keep track of the subtasks it has generated and, before returning, wait for their termination. Rather than the Java concurrency framework, you may also use virtual threads (introduced in Java 21) for the tasks.

Alternatively, one may explicitly create a set of worker threads that operate on a shared pool (queue or stack) of task descriptions (access to this stack has to be propertly synchronized); each task description consists of the arguments of a call to the recursive function. Rather than executing a recursive call directly, a corresponding task description may be generated and placed into the pool. The main program may recognize the termination of the algorithm by checking whether the pool is empty and all threads are in a state where they wait for another task from the pool (which may be indicated by a corresponding variable set by each thread).

It may be that one alternative is substantially faster than the other one, so implement *both* and compare them in your benchmarks.

OMP C/C++ Solution The program may annotate each recursive function call that shall be executed in parallel by the annotation `omp task` to create a parallel task (since the functions do not return a result, it is not necessary to use `omp taskwait`; a function automatically waits for the termination of all tasks it has generated before returning). The main program has to embed the initial function call into a section marked as `omp parallel` to create the parallel threads that process these tasks; the function call itself must be annotated as `omp single` to let one thread start the execution of the function.

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

Alternatively, one may use the annotation `omp parallel` to create a set of worker threads that process a shared pool of task descriptions (see above explanation of the Java solution).

It may be that one alternative is substantially faster than the other one, so implement *both* and compare them in your benchmarks.

Please note that (different from the Java solution) the C/C++ solution has to explicitly manage heap-allocated memory, i.e., every array allocated on the heap must be eventually freed (since many temporary arrays are created, the program may run out of memory otherwise). For instance, if a recursive function receives a freshly allocated array as an argument for local use, this function should free the array before return. To speedup execution, you also may implement your own memory management scheme (but then make sure that it works correctly in a parallel execution context).

Benchmarks Benchmark the program for $P = 1, 2, 4, 8, 16, 32$ cores/threads (potentially more).