Exercise for Course

Parallel High-Performance Computing

Dr. S. Lang

Return: 10. December 2015 at the beginning of the exercise or earlier

---

Task 15 Resource Management (7 points)

In a system there are two identical occupiable resources and N candidates for using one of both resources. Implement with the ThreadTools a program, that ensures, that at most two candidates at a time can occupy the resources, but further, that each candidate can finally enter at some time. Each candidate shall occupy its resource with distinct (randomized) length. For this use some randomly choosen time:

```
1 sleep(rand() % 3);
```

For the working threads you as usual derive a class `Consumer` from `BasicThread`, that overloads the `run` method.

Use the prepared kernel in the file `resources.cc`. This can already be compiled with the instruction `make`. Test your program with $N = 22$: Here for example have 22 football players to share 2 showers after the game ☺. Test using the output, whether actually each of the players has taken a shower.

Task 16 ThreadTools: Traveling Salesman Problem (8 points)

A traveling salesman has to visit companies in n cities these are connected underneath each other. Thereby he shall choose a path as short as possible. The simplest solution is, starting at an arbitrary city and walk systematically along all possible pathes and calculate the „costs“. The (visited) pathes can be stored in the nodes of a tree, the best path can be found with depth search. For large n this solution is impracticable, since there are $(n-1)!$ possible pathes. An improvement uses *branch-and-bound*: Hereby with the traversal of the possible pathes it will be checked, whether the current path length superceeds that of the up-to-now optimal path, and is eventually terminated. A search tree with now regularily cutted pathes is shown in figure 0.4. The different path length is one of the main difficulties of the parallelization.

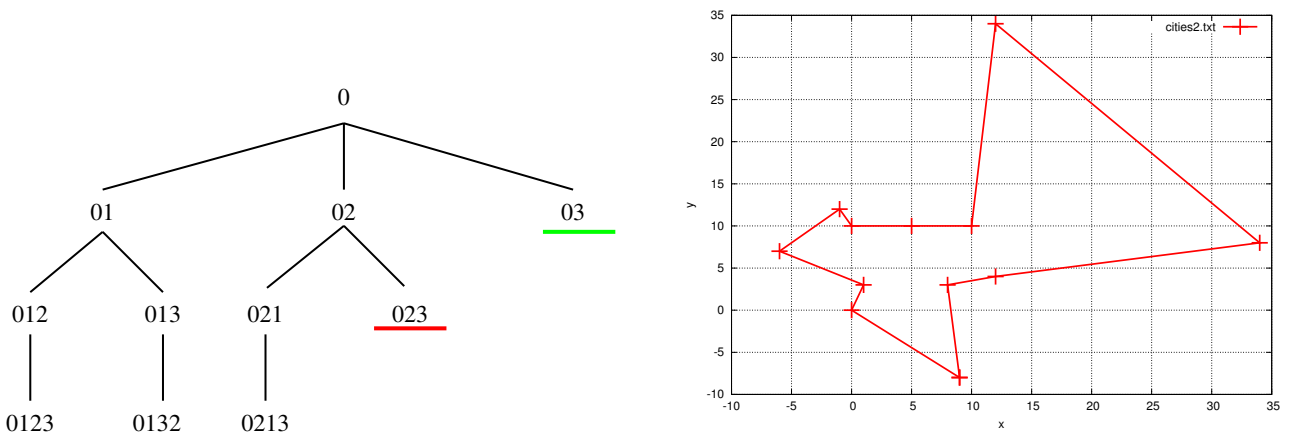

Abbildung 0.4: Left: Branch-and-bound search with irregular search depth. Right: Exemplar optimal path for 12 cities.

The method could for example be implemented using recursion. More favourable is however to use a variant with stacks:

```
1  // travelling salesman with stack: branch-and-bound-search
2  path = {0}
3  stack.push(path)
4
5  while (stack.pop(path)) {
6    if (path has maximal length of n)
7      if (path is new optimal path) best = path;
8    else
9      for (all still not visited aims i)
10       if (Length (path ∪ i) < best known path)
11         stack.push(path ∪ i);
12 }
```

The stack is initialized with the path, that only contains the start city. In each iteration the path on top is considered. Has the considered path the length n (count of city to be visited), then it will be evaluated, otherwise for each city, that has not been visited, a new path is put on the stack, if this path is not longer than the up-to-now best found path.

Parallelisation strategy with Threads:

The nodes of the search tree can be processed by a single thread, since each node characterises its successors in a unique way. Each thread contains a local stack with nodes to be processed. To share work between the threads, additionally a global stack is used. The global stack is inititalized at program start with the root of the tree and the worker threads are started with initially empty local stacks. Workers with empty stack try, to take a node from the global stack and to process with branch-and-bound. If the global stack is empty, waiting processes, that need new work (nodes), wait until working threads share again a part of their search tree. The global stack therefore contains a counter, how many processes are waiting, as well as a semaphore, where idle threads can wait by a P() operation.

Each working thread tests during its branch-and-bound phase after a certain number `MAXI` of nodes, that it has processed, whether it can push work to idle workers. This means to push a node from the local back to the global stack. The pushed node should contain as much work as possible. Therefore the pushed nodes should be taken from the lower end of the local stack, since nodes, stored in the search tree lower, with few remaining work, reside on the local stack on top. After the push onto the global stack the pushing thread wakes a idleing thread.

Hints for the skeletal structure

In the `threadtools.zip` with the kernel `tsp.cc`, `tsp_cities` with city data. In the file `tsp.cc` you find a program structure, that you can (but need not to) use to realize this task!

In the skeleton you already can find classes for node, line 125, the best up to now found node, line 151, the local (line 283) and the global stack (line 190) as well as methods to read (l. 74) and write (l. 25) a city file. The distances are stored in data structure with `std::vector<float>` named `dist`, the variable `nCities` stores the total city count and `nThreads` the number P of threads. A node of the tree stores its path (number of cities within a `char` array), the length of the path (count of cities) and the current necessary costs (distance of points in euclidean norm). The class `BestNode`, l. 151, for the best already found node is derived from the node class and contains additionally a function to update the best node, the access is locked by a mutex.

The global stack contains a Mutex variable `mutex`, l. 270, to ensure exclusive access, as well as a semaphore `waitSem`, l. 272, that is used for waiting and for the waking of idle threads. By the call of `wait()` (global stack, l. 221) a thread increases the number of idle workers with the semaphore. Furthermore this method returns the number of idle threads (or -1, if no one is waiting anymore, in this case also the flag to start the termination phase is set). The method `sharedSignal` (l. 210) is the counterpart, it decrements the count of waiting processes and wakes another thread. The class for the local stack contains two functions `popRear` and `peekRear`, (l. 309 resp. 318), that enable access to the bottom of the stack, to push the lower elements of the local stacks onto the global.

The worker class `Worker`, l. 344, realises a working thread. It is derived from `TT::BasicThread` and overloads therefore the method `run`. The method `branchAndBound` shall realize the above described search. An important help method is `expandNode` (l. 437), that determines for a given node its successors in the tree and puts them on the local stack.

Task

Implement a `run` method of worker threads and the method for `branchAndBound` of the `Worker`, as described above in the parallelisation strategy:

- The `run` method tries to get work from the global stack in an infinite loop. If it can fetch a node from the global stack, the node is stored on the local stack and the branch-and-bound is started. If it does not get a node from the global stack a check using the `wait` function of the global stack should test whether already $P-1$ threads are waiting (return value -1). In this case the last thread may not block, but has to wake the others (method `sharedSignal`). Is a thread waked, thus he returns from the `wait`, it has to check again with `terminate()`, whether the termination phase is introduced: Then it may terminate, otherwise the thread further tries to get work.
- In the method `branchAndBound` work is done until the local stack is empty. Every `MAXC` iterations are checked, whether work can be passed away and be pushed onto the global stack. A node shall only be passed away, if the remaining tree size is larger than `MIND`. The node to push shall be taken from the bottom of the stack. Here you can use the methods `popRear` and `peekRear` (reading access) in the class of the local stack. Has a node of the global stack been pushed, idle workers have to be waked.

Your program can be compiled by the call of `make`. Compile your code preferably with optimisation `-O3`. The first city file `cities1.txt` contains 5 cities on a line is meant for program validation purposes. Measure now the program runtimes (*user-time* in s) for $1, 2, 4$ and 8 threads and for the four remaining files `cities[2-5].txt` and calculate the gained speedup. If you have the possibility to measure the times also on other computers, that enable larger thread counts than in the pool, then this is favourable. For time measurement you can choose the command `time` in the console:

```
1 /usr/bin/time -f %e ./tsp ./tsp_cities/cities4.txt
```

measures the elapsed user time in seconds (switch off the output by setting the variable to `output =false`). Discuss the achieved speedups and plot for presentation purposes your results as in one of the previous tasks. If you pass the program as further parameter a filename, the output file with the optimal path is written. This can be used by Gnuplot with the command `plot '<datei.dat>'w lp` for a plot as in Figure 0.4 right. Please add to your solution a plot with the optimal path for file `cities5.txt`, that shows the optimum that your implementation has found!