Parallel and Distrubuted Computer Systems

First Assignment

Konstantinos Chatziantoniou 8941

1/11/2018

Aim of the assignement

Three different versions of parallel programms have to be made, using:

- pthreads
- cilk
- openmp

that implement the quicksort algorithm.
The programms should check if the result is correct and work for 2^p threads where $p \in [1 : 8]$ and for 2^n elements to be sorted, where $n \in [12 : 24]$.
`n` , `p` are defined by the user.
Then, the speed of the sequential and the three parallel quicksort versions(and other implementations of them) will be compared, for different number of threads and elements.

Hardware Specifications

The code was ran on a HP laptop with the following specs:

- CPU: i7 7500u (2 cores/ 4 threads)
- RAM: 8gb ddr4
- OS: ubuntu 18.04
- compiler: gcc (with -Ofast flag) (icc for cilk)

Code Explanation

Each recursive call of quicksort, uses a different part of the array.This means, those two calls can be done in parallel. The parent thread will spawn a child thread to do the first call, then it will continue doing the second call and wait for the child.

In `cilk` and `openmp` it is straightforward how to parallelize those calls:

- `Cilk`: I used the `cilk_spawn` keyword in front the first call. I can set the number of threads(cilk workers) by using the cilk api for linux (`cilk_set_params("nworkers") , char* n);`)

- `Openmp`: I used `#pragma omp parallel` and `#pragma omp single` directives,to call the entry point for qsort, and `#pragma omp task` for the first recursive call. I can set the number of the `num_threads(n)` parameter in the parallel directive.

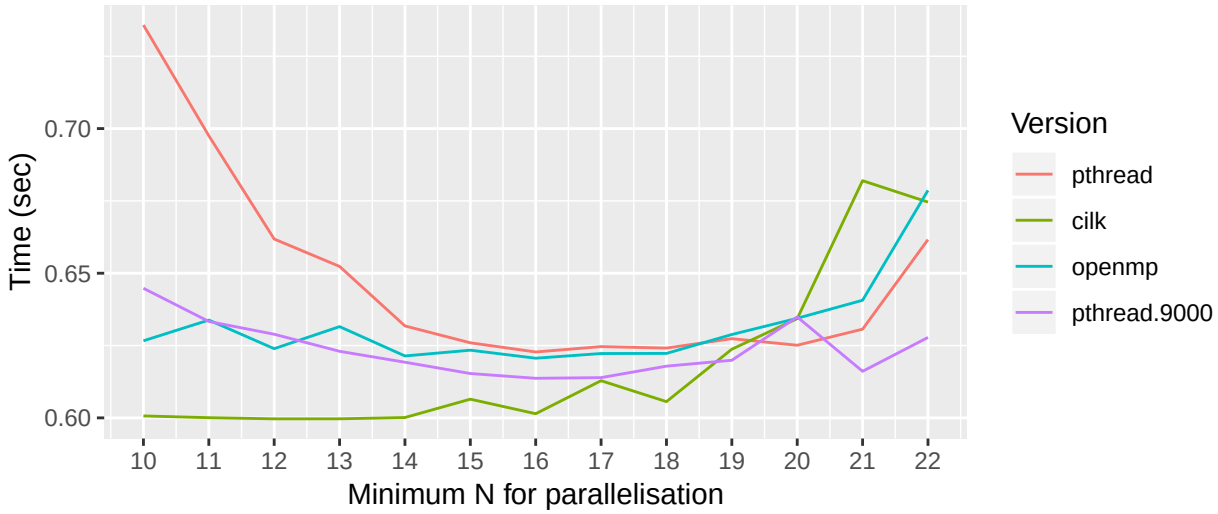
For `pthread`, to parallelise the recursive calls, I used the `pthread_create` function for the first call, normally call the second and wait for the spawned thread to join. In `pthread` there is no automatic way to control the maximum number of threads. Instead, I created a global array of `pthread_t` (tid) and one of `int`(availableThreads). When I want to create a thread, I call the `findAvailableThread` function, which checks for a `0` in availableThreads array(1 means pthread_t[index] is in use) and returns the `index` when one is found. If all are reserved it returns -1. When a thread joins, I call the `releaseThread()` function to make `availableThreads[index]` $= 0$, so it can be reused. Both those function lock a mutex before changing the array.

The `pthread.9000` version, also parallelises the `partion`. The parent thread does partition for the first half and the spawned thread for the other half of the array. They wait for the other to complete the partition,do the rest of the swaps and wake it to continue with the recursive calls.

Optimal number of elements for parallelisation(grain size)

Creating and terminating a thread introduces some overhead. When the problem size is small, this overhead may make the parallel programm slower than the sequential. So, an optimal number of elements `n` should be found[1], below which sequential approach will be used.

figure 1



From *figure 1* it is apparent that `cilk` and `openmp` are not affected much by small grain size, propably because of the automatic optimisation. Pthread ,though, is at least 10% slower for low grain size($n < 13$). All parallel programms are at least 8% slower for large grain size($n > 20$). As optimal minimum number for parallelisation we will choose:

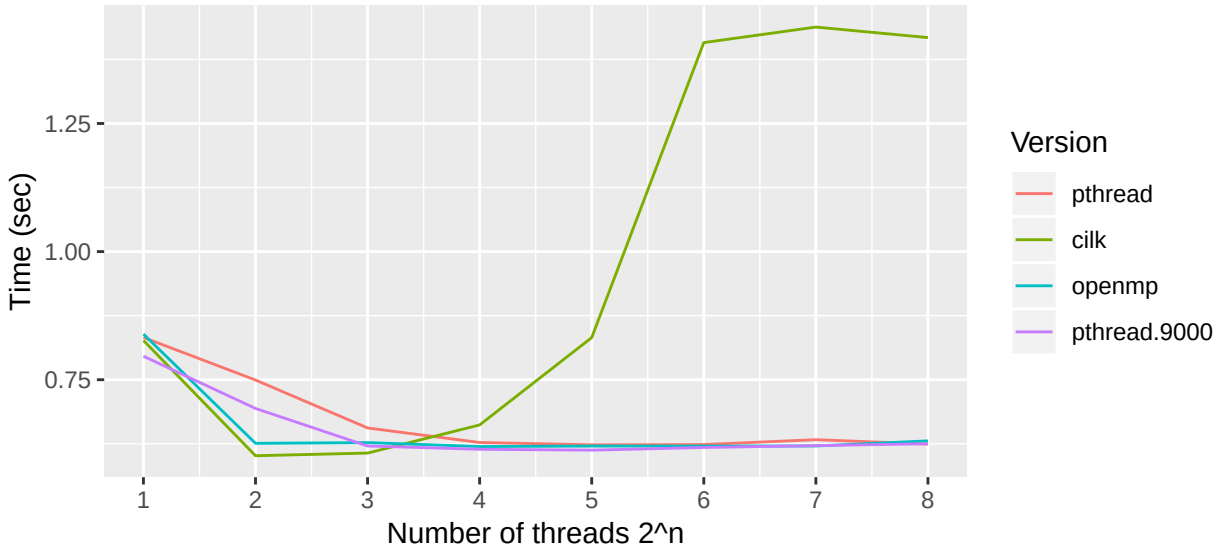
- Cilk $n = 13$
- Openmp $n = 14$
- Pthread $n = 18$
- Pthread.9000 $n = 17$

[1]Number of elements 'n' is different for each machine.

Optimal number of threads

The benchmarks were run on a 2-core CPU with hyperthreading(4 physical threads). The impact on speed of creating more threads than the physical threads is uncertain. It could be negative, because of the overhead of thread context switching, or positive(e.g. when a thread is 'sleeping' or waiting to lock a mutex). In order to determine the optimal number of threads[2], testing is required.[3]

figure 2



From *figure 2* , it seems that:

- cilk works better for, $n \in [2,3] \Rightarrow p \in [4,8]$
- openmp works better for $n > 2, \Rightarrow p > 4$
- pthread works better for $n > 5, \Rightarrow p > 32$
- pthread.9000 works better for, $n > 4, \Rightarrow p > 8$

Final Comparison

Since the optimal number of maximum threads and number of elements for parallelisation have been determined, it's time to compare the programms, sequential and parallel, for different problem sizes. *(The yellow line is* $nlogn$*)*

[2]optimal number of threads is different for each machine
[3]Cilk's workers cannot be more than 64.

figure 3.1

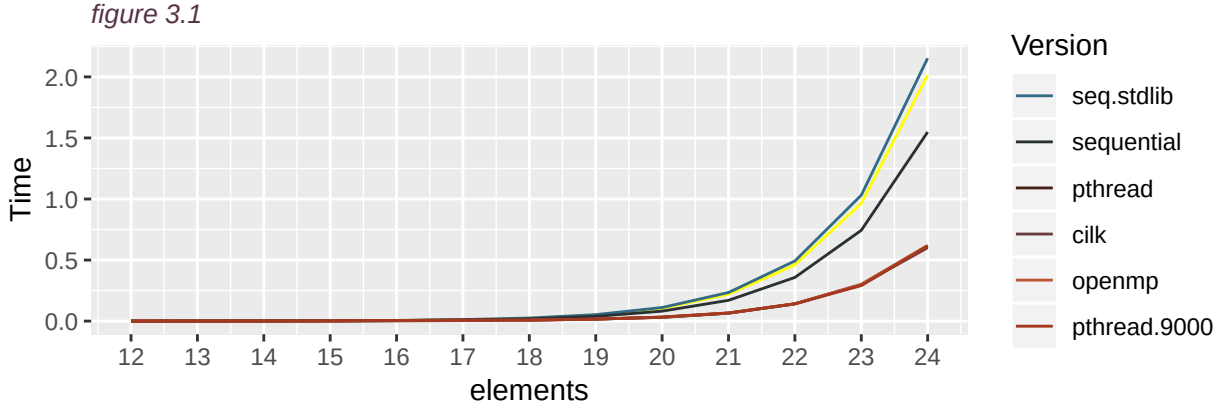


figure 3.2

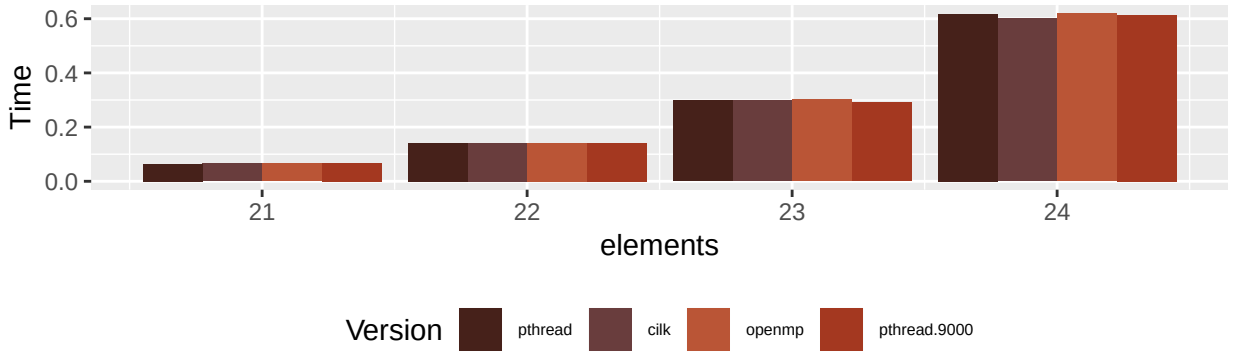


Figure 3.1 shows that for `n` larger than `19`, there is a noticable difference in speed, between the parallel and the sequencial programms. Also, the growth rate of the sequentials is `O(nlogn)` , while the parallel's rate of growth is obviously less.

From *Figure 3.2* it is apparent that all four programms have the same speed for various `n`, with Cilk being the winner by a very narrow margin.

Misc

1. Cilk had noticably better performance (25% faster) when compiled with icc instead of gcc.
2. In both pthread programms, I create two functions; one that returns `void*` and has `void*` arguments(the `void*` is casted as `pth_input`), and another that returns `void` and has `pth_input` arguments. Those two function do exactly the same. When I wrote the programm with just the function with the pointers, the program ran much slower (x1.75 slower).
3. Sometimes of the 'noise' of the OS affects a lot the time of the programms.Occasionally 10-20% increase in time has been noticed.

The following plots show the combination of max threads and min N for parallelisation for all parallel programs.