ETH zürich

High-Performance Computing Lab for CSE **2021**

Student: Anna Hutter Discussed with: Alain Hügli, Sina Klampt

Solution for Project 2

Due date: 26.03.2021 (midnight)

HPC Lab for CSE 2021 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to Moodle (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

This project will introduce you to parallel programming using OpenMP.

1. Parallel reduction operations using OpenMP [10 points]

1.1. Reduction clause

We define a parallel region around the loop which calculates `alpha_parallel`. In this region we parallelize the for loop with the reduction clause using the operator + and the reduction variable `alpah_parallel`.

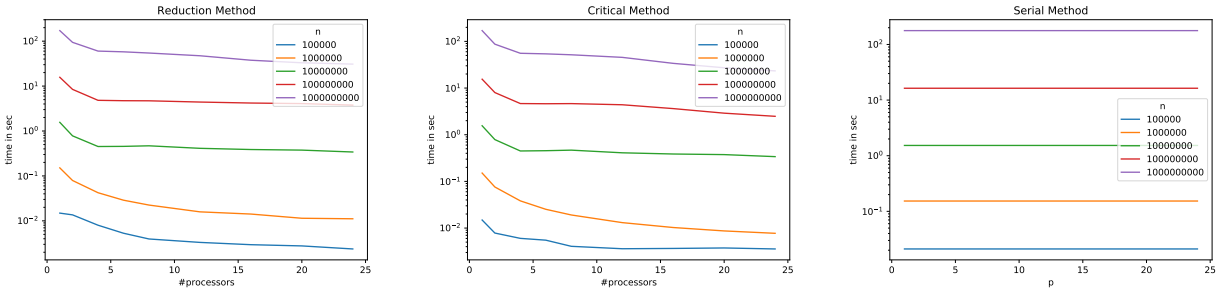
1.2. Parallel and critical clauses

When using the parallel and critical clauses we also start with a parallel region around the loop calculating `alpha_parallel`. To avoid extreme time loss we do not make a critical region inside the loop but give each thread a `local_alpha` to which they add `a[i]*b[i]`. After the loop we definie a critical region where each thread adds their `local_alpha` to `alpha_parallel`. This saves a lot of time since each thread only needs to enter one critical region.

1.3. Results

As we can see in the plots below all the solutions scale we. The reduction and the critical parallelization are roughly the same performacewise. When looking at the exact numbers we can observe that the critical version is a little bit better but not significant. It can't be seen in the plots.

The parallelization of the loops is benificial for all the N seen in this plot since the parallel version is always sigificant better than the sequential version.



For the parallel plot we run the code for 1,2,4,6,8,12,16,20 and 24 threads. All three plots are the mean from 10 measurements.

2. The Mandelbrot set using OpenMP [30 points]

2.1. Serial implementation

The serial version of the code is implemented like the pseudocode on the projectdescription.

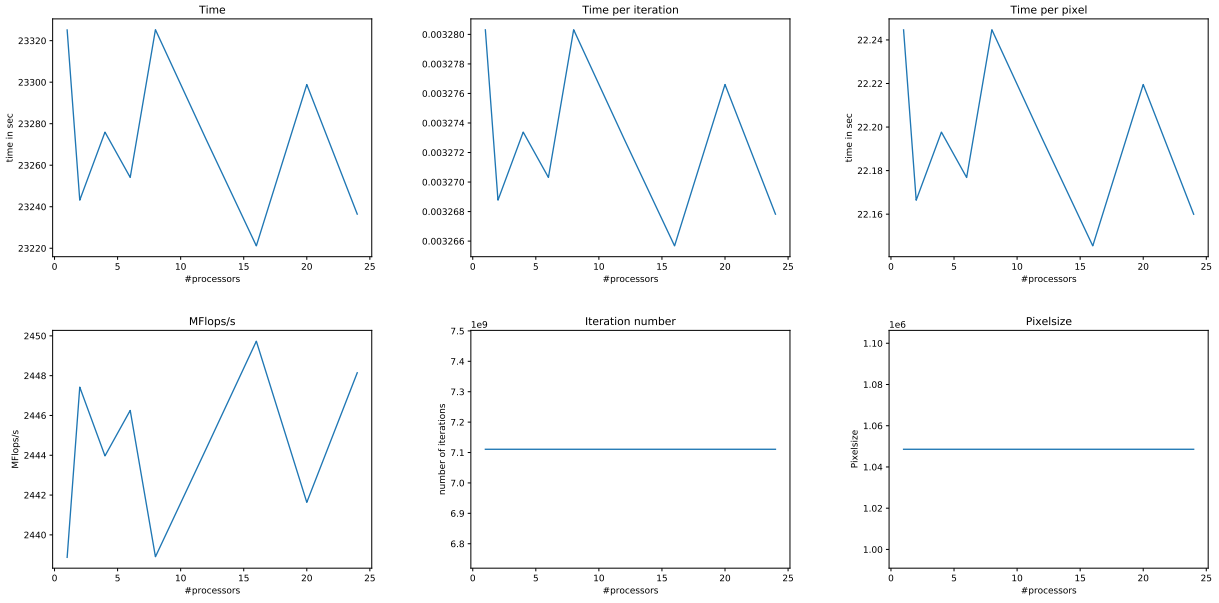
2.2. Parallel implementation

If we want to be sure about the values of `cx` and `cy` when parallelizing the loope we can not calculate theme the way they are in the serial version. To avoid provlems we make `cx` and `cy` private variables and calculate them in the inner loop. This lets us still use `cx` and `cy` for the calculations in the while loop. Other variable which need to be private are `x`,`y`,`x2`,`y2` as well as the new variable `n` which calculate the number of iterations for one complex number. `n` is added to the public variable `nTotalIterationsCount` and to avoid race conditions we used `#pragma omp atomic` for the calculation.

We only parallelized the outer loop with `#omp pragma for` since we already have between 512 and 4096 iterations in this loop. After trying different scheduels and blocksizes `schedule(dynamic)` and `schedule(static)` seem to run the best. In this implementation we therefore us the dynamic schedule.

2.3. Results

As we can see in the plots our version does not scale at all. We were not able to figure out what the problem was and did not try it many times since the plots took multiple hours to run.



For the parallel plot we run the code for 1,2,4,6,8,12,16,20 and 24 threads. The plots are the mean from 5 measurements. We made plots for all the N values mentioned in the projectdescription but chose to present the ones from N = 1024 as a representative case. All the other plots look roughly the same.

3. Bug hunt [15 points]

3.1. Bug 1

`#pragma omp parallel` and `#pragma omp for` can only be combined, if the loop starts at the begin of the parallel region. If there is any other code between the start of the parallel region and the for loop that needs to be parallelized the `#pragma omp for` needs to be in the parallel region directly before the for loop with no other code inbetween. In this case we have `#pragma omp parallel shared(a, b, c, chunk) private(i, tid)` as the initialization of the parallel region and `#pragma omp for schedule(static, chunk)` right in front of the for loop.

3.2. Bug 2

The variable `tid` needs to be private since each thread needs to know its own id. If it is public like in this case we dont know if we ever visit the if statement since at any time any thread can write his number to `tid`. I also assumed, that we wanted to have the total for each thread and not the over all total. Therefore the variable `total` also needs to be private. The solution would be to use `#pragma omp parallel private(tid, total)`. If we would want to know the total over all the thread the variable `total` would not be private. Insted there would be a private variable `loc_total` and we would calculate the local total over each thread. At the end of the privat section we would add the `loc_total` to `total` and make a critical section to avoit race contitions.

3.3. Bug 3

When using section in omp only one thread enters each section. In this program only threads entering a section call the print_results function. Since not all threads enter a section not all threads will call the print_results function. Because of this not all threads will reach the barrier in the print_results function and the program stops. This can be resolved by deleting the barrier in the print_results function. This is okey since it is the last part of each section and we have a barrier before we exit the sections nowait part.

3.4. Bug 4

The problem is, that the array a does not fit on the stack of each thread and we therefore get a segmentation fault. Since a is private each thread needs enough memory to allocate a. To avoid this we can give each thread more memory by calling "export OMP_STACKSIZE="9M" " before executing the program. For this size of a 9MB is enough since we have 1048*1048 doules and each double needs 8B of memory. The stacksize we need to allocate therefore needs to be calculated new if the size of a changes.

3.5. Bug 5

In this code a deadlock occures because the thread in section 1 sets locka first and then wants to set lockb. At the same time the thread in section 2 stes lockb and after initializing b tries to set locka. This leads to a deadlock since both threads are waiting for the other thread to unset a lock. To avoid this both threads need to set the first lock initialize their array and unset the lock. Then both step set locka first and lockb secound add their array to the other array and then unlock both locks. This prevents a deadlock because both threads unset their locks after initializing their array and then try to set the locks again in the same order. For this example I assumed that the order in which the for loops get executed does not matter.

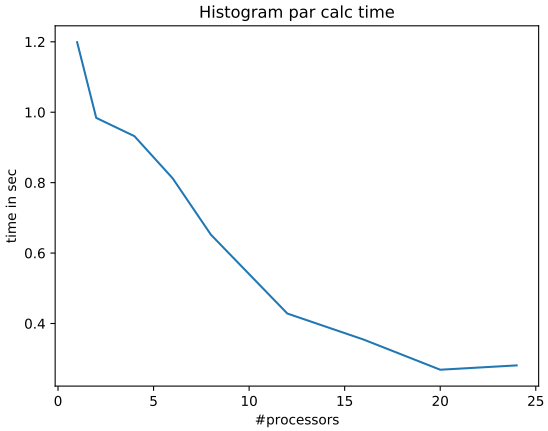
4. Parallel histogram calculation using OpenMP [15 points]

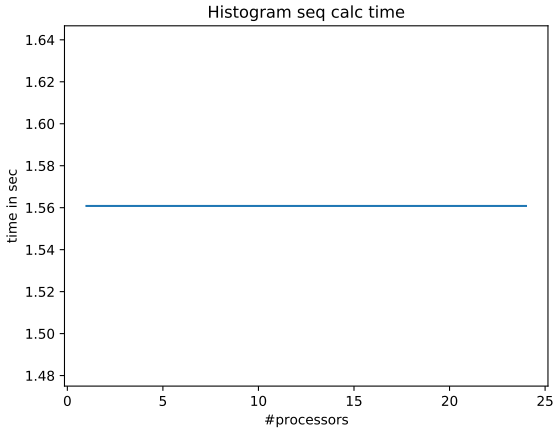
4.1. Parallel implementation

When looking at the for loop we need to parallelize we see, that each thread accesses a randomn location in the `dist` array. Since having a critical region in each iteration of the for loop would make the code extremely slow we give each thread a local array `locdist`. After the initialization of `locdist` we parallelize the for loop with `#pragma omp for`. Each thread adds the numbers up in to the local dist array. Before exiting the parallel region we add the numbers of the local dist arrays to the dist array. To avoid race conditions we use the command `#pragma omp atomic` when adding `locdist` to `dist`.

4.2. Results

As seen in the parallel runtime plot our solution scales well. The sequential code is even slower then the paralle version with 1 thread. We are not sure why this is the case but we suspect that this is a measurement error.





For the parallel plot we run the code for 1,2,4,6,8,12,16,20 and 24 threads. The plots are the mean from 10 measurements. The sequential plot is also the mean from 10 measurements.

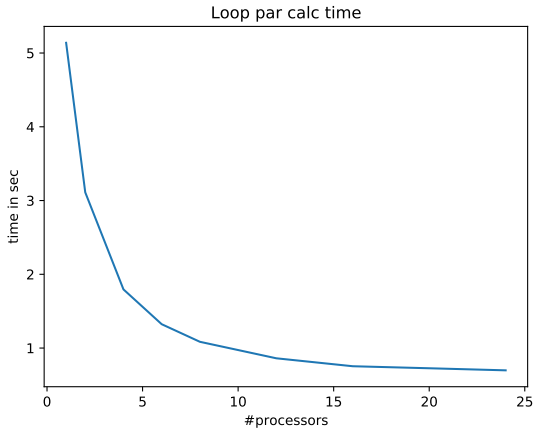
5. Parallel loop dependencies with OpenMP [15 points]

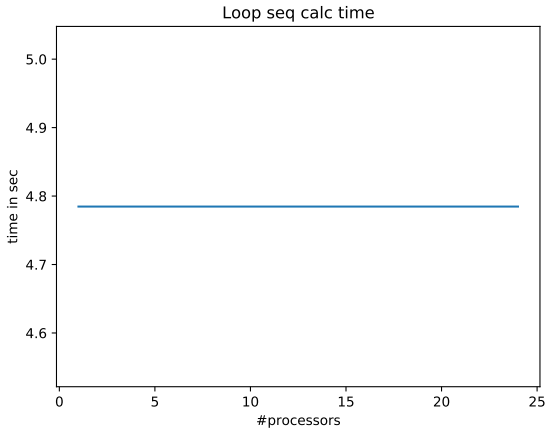
5.1. Parallel implementation

The way we parallelize this for loop is by making a parallel region and giving each thread a block it has to calculate. This ensures that each thread gets one connected block. For the first element each thread calculates the start value with `pow(up,tid*N/nthreads) * Sn`. Then the threads calculates the loop for its assigned block and saves the elements in the `opt` array. For the parallelization of the loop we use reduction on the multiplication of `Sn`.

5.2. Results

As seen in the parallel runtime plot the solution scales very well. The parallel solution also only needs 5 seconds for one thread which is nearly as good as the serial solution.





For the parallel plot we run the code for 1,2,4,6,8,12,16,20 and 24 threads. The plots are the mean from 10 measurements. The sequential plot is also the mean from 10 measurements.

6. Task: Quality of the Report [15 Points]

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to Moodle.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your OpenMP solutions.
  - your write-up with your name project_number_lastname_firstname.pdf,
- Submit your .tgz through Moodle.