Parallel Programming - Report 2

Lucas Paes

November 2022

1

The following naive implementation can deadlock since `lock(a)` always happens first, and a and b might be a different stacks in different threads:

```
function move_top(a: Stack, b: Stack) {
    lock(a);
    lock(b);

    elem = pop(a);
    push(b, elem);

    unlock(a);
    unlock(b);
}
```

Listing 1: Naive implementation with deadlock.

In particular, the following sequence of events leads to a deadlock with this implementation:

Thread 1	Thread 2
`move_top(a, b)`	`move_top(b, a)`
`lock(a)`	`lock(b)`
`lock(b)`	`lock(a)`

Table 1: Thread calls that cause a deadlock for the naive implementation.

To fix this, we can introduce a total ordering of locks and make sure locks of lower order are never acquired if we have contetion of a higher order lock. In this code sample, I introduced the function `is_smaller(a, b)`, which returns whether $a < b$ in this total ordering. In practice, one could use lock IDs or memory addresses to implement this ordering.

The following code sample will never deadlock since it guarantees the same acquire and release order independently of what is a or b:

```
function move_top(a: Stack, b: Stack) {
    small, big: Stack;

    if (is_smaller(a, b)) {
        small = a;
        big = b;
    }
    else {
        small = b;
        big = a;
    }

    lock(small);
    lock(big);

    elem = pop(a);
    push(b, elem);

    unlock(big);
    unlock(small);
}
```

Listing 2: Implementation that avoids deadlock by assigning a total order to locks.

2

This approach will work, since it guarantees only one writer can update the list at a time while no one is reading from it. If no one has acquired the exclusive write-lock, the list can be considered immutable, and so multiple readers might read from it without interference.

3

From reading the code, we can deduce that array elements will be initialized according to the following recurrence:

$$\begin{cases} a_0 = 0 \\ a_i = a_{i-1} + i, \quad i \geq 1 \text{ and } i < n \end{cases} \tag{1}$$

Assume hypothesis $H(i)$ for the value of a_i as $H(i) = \frac{i(i+1)}{2}$. If we plug $H(i-1)$ back into the original recursion for a_i, we get

$$\begin{aligned} a_i &= a_{i-1} + i \\ &= H(i-1) + i \\ &= \frac{(i-1)((i-1)+1)}{2} + i \\ &= \frac{(i-1)i}{2} + i \\ &= \frac{i^2 - i}{2} + \frac{2i}{2} \\ &= \frac{i^2 + i}{2} \\ &= \frac{i(i+1)}{2} = H(i) \end{aligned} \tag{2}$$

And thus $a_i = H(i)$ and our hypothesis holds true. We now have a formula for a_i that only depends on i, and no longer depends on a_{i-1}. Writing a parallelizable loop is trivial, and the code for it follows:

```
for (int i = 0; i < n; i++) {
    a[i] = i * (i + 1) / 2;
}
```

Listing 3: Parallelizable loop for array initialization.

4

Setup

First, I implemented a `main` function that reads an n value from the first argument of program invocation to determine the problem size. To avoid overflows for large n, all `int` types were replaced by the `uint64_t` type available in `<inttypes.h>`.

The main function then populates an array of size n with decreasing values starting from `UINT64_MAX` and ending at 0. This maximizes the difference m between the maximal (first) and minimal (last) elements of the array, creating the worst case scenario for the counting sort algorithm of $O(n + m)$.

The `count_sort()` function is parallelized with the OpenMP pragmas, depending on whether compilation macros `PARALLELIZE_1` and `PARALLELIZE_2` have been defined. It is called and timed using `omp_get_wtime()`, and the program outputs the time it took to run the function as a string on the last line of its output. The serial program is compiled as follows:

```
   clang -pedantic-errors -Wall -Wextra -fopenmp -O1
-o bin/5.out 5.c
```

An extra `-D PARALLELIZE_1` or `-D PARALLELIZE_2` is added to the compilation flags to produce parallelized versions of the program.

Finally, I wrote a Bash script that compiles all binaries and runs each of them three times while varying k from 1 through 17, thus varying n from $2^1 = 2$ to $2^{17} = 131072$. The script saves the measured time results to a CSV file, and a Python script plots the execution time for each $(k, repetition, implementation)$ combination.

The programs were run with `OMP_NUM_THREADS=4` on an Intel i7 processor with 4 cores.

Discussion

We can see from the results in Figure 1 that the serial implementation is the fastest for problem sizes smaller then approximately $n = 500$. However, it is clear that the outer loop parallelization (`parallel1`) overtakes it at around $n = 1000$, while the inner loop parallelization (`parallel2`) overtakes it at around $n = 5000$.

It is also visible that `parallel1` outpaces `parallel2` significantly for large n. This is probably due to the constant swapping between a master thread and a team of threads at every iteration of the outer loop in `parallel2`, which introduces more barrier points and, thus, contention.

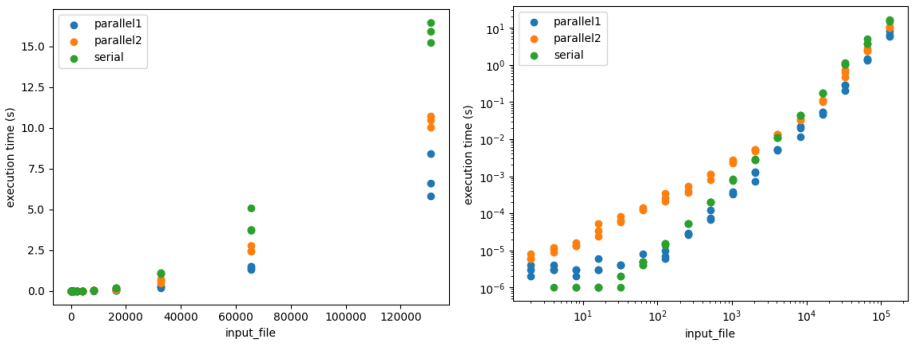

Figure 1: Linear (left) and log (right) scale plots of execution times over number of elements in the array (3 repetitions each).

Reproducibility

For reproducibility, source files, execution scripts, input files and generated CSVs can be found in the Github repository[1].

[1] `https://github.com/serramatutu/5dv152-report3`

5

Acknowledgement

It is important to acknowledge I used LodePNG[2] for generating images of the Mandelbrot set, and adapted a function for converting HSV color schemes to RGB from `helenjw` on Github[3]. Since generating an image with a pretty color palette is not a central problem nor required for this report, I did not spend too much time on this. All of the parallelized code and the experimental setup is mine, though.

Also, I was not able to run this question in Kebnekaise on time. I am submitting what I have, and, if necessary, I will upload Kebnekaise results on the next deadline.

Setup

Much of the test setup is the same as the one in question 4. In fact, the scripts used for compiling, running and plotting results were the same, and the convetion that the last output of the program is its execution time was followed. This made it easier to compare results from different input files for the Mandelbrot problem.

Four versions of the Mandelbrot function were implemented:

- `serial`: Serial implementation that processes one pixel at a time.
- `for`: Parallel for implementation that parallelizes the outer for loop. The outer loop was chosen to avoid the contention introduced by inner loop parallelization, as seen in Question 4.
- `pixeltask`: Task-based implementation that assigns a task for each pixel in the image. This results in millions of fine-grained tasks.
- `rowtask`: Task-based implementation that assigns a task for each row. This results in thousands of coarse-grained tasks.

They were compiled with:

```
clang++ -stdlib=libc++ -fopenmp -Wall -Wextra -pedantic -O3
-o bin/5.out 5.cpp lodepng.cpp
```

Results

As visible in Figure 2, all parallel implementations outperform the serial version. This is probably due to the fact that, even for a small image, the Mandelbrot

[2] `https://github.com/lvandeve/lodepng/`
[3] `https://github.com/Helenjw/CSC317/blob/master/lab1/src/hsv_to_rgb.cpp`

problem requires some thousands of pixels with thousands of iterations each, which is already a lot of work to be performed.

We can also see that as the dimensions grow, the `rowtask` implementation, with coarser grain tasks, is the one which runs quicker. This is due to the reduced overhead of task switching and scheduling, which was too high for `pixeltask`.

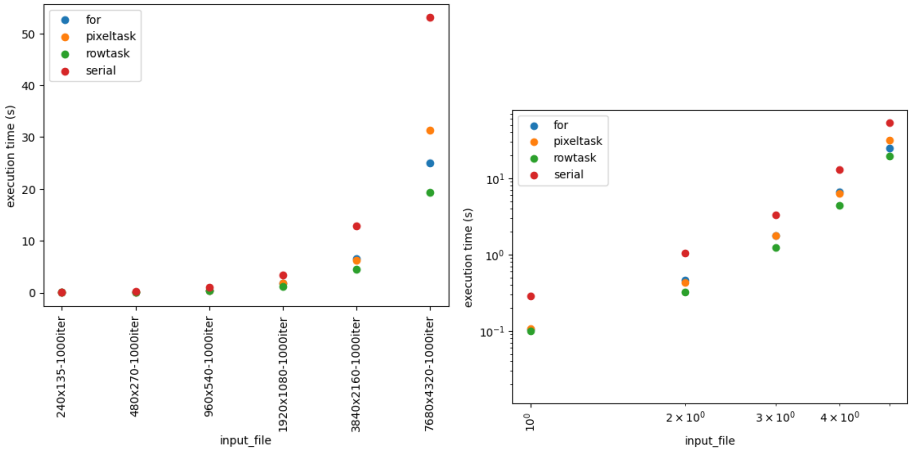

Figure 2: Linear (left) and log (right) scale plots of execution times over image size.

Finally, I include one of the generated images (Figure 3).

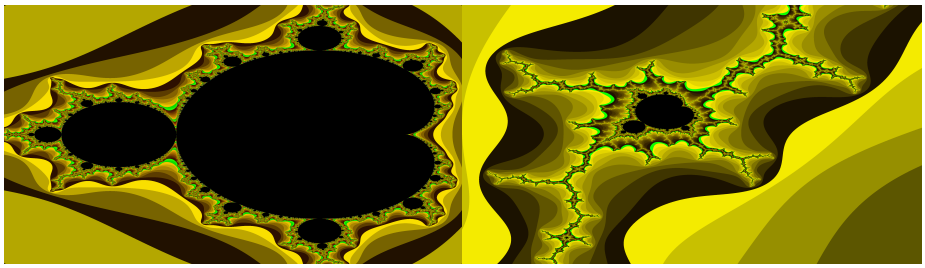

Figure 3: Generated Mandelbrot set images. Default coordinates are on the left, some interesting coordinates I found are on the right.