Université de Liège
Faculté des Sciences appliquées

A Mandelbrot set explorer

INFO9012-1 : Parallel Programming

Auteurs :
Manon Gerard s201354
Tom Léonard s200974

Professeur :
P. Fontaine

3e année de Bachelier Ingénieur Civil

Année académique 2022 - 2023

1 Tests

We tested our programs with a machine with 6 cores and chose 6 threads and on a window of size 300 on 300. For each phase, we tested to zoom in at the maximum to count the fps.

Phase	fps counts
sequential version	3
Phase 1	15
Phase 2	15 - 29,4
Phase 3	15 - 29,4
Phase 4	5 - 15

TABLE 1. Caption

For phase 1, no limitation in FPS were imposed, so we noticed a few 100 values, but these immediately dropped.

2 Phase 2

Firstly, we implemented a unique circular buffer. When this buffer is created, the Pdata and associated images are created for each element of the buffer. Each element of this buffer has its associated lock created with `pthread_mutex_init`. At maximum there are 15 producers and initially the consumer sets the first images in number equal to the producers, so the buffer must be able to have at least these first images. Therefore, the capacity of the buffer is 16 as in reality it always needs at least an element unused. We also created a thread pool for the producers threads thanks to `pthread_create`.

The consumer, which is the main thread, sets the first images in the buffer to a "*to compute*" status thanks to `circularBufferEnqueueToCompute`. Then, the producer threads looks for the work to do with `circularBufferChooseNextToCompute`, which locks an element of the buffer. When a work is available, the thread changes the status to "*computing*" inside the function. If no work is available, the lock is released and the thread sleeps for 1/60 seconds. This time was chosen to be smaller than 1/30 seconds, where 30 is the maximum of fps that can be reached. Like this, the thread won't miss any image calculation. When a producer has found a work to do, it computes it thanks to `mandelbrot` which changes the value inside the Pdata of that element buffer. We changed the arguments of this function to take into account the parameters which are scale, cx, cy and angle. After this calculation, the status of the element is changed to "*computing*" and the lock is released. Once a task has been produced, the consumer gets it thanks to `circularBufferDequeue`. When no tasks are available, the consumer simply sleeps for 1/60 seconds. After dequeuing, the consumer displays it in the window and release the lock of this element which was lock in the dequeue function. The consumer is also responsible for requesting the other tasks once again with `circularBufferChooseNextToCompute`.

3 Phase 3

For this phase, we started from the previous version and tried to remove the busy waiting, which corresponds to the "sleep". For this, in addition to the locks, semaphores were introduced for each element of the buffer. Our previous function for the circular buffer now do not need to verify anymore if the status is the correct. We adapted the producer consumer approach this way :

```
Initially: E=15, N=0, $A_{1,....,15}$=0
Process Producer
1  while ⊤ do
2      N.wait()
3      element where there is the work.lock()
4      go get the work and do it
5      element where there is the work.unlock()
6      $A_{idx\ work\ done}$.signal()
```

```
Process Consumer
1  while ⊤ do
2      E.wait()
3      element enqueued.lock()
4      enqueue the task
5      element enqueued.unlock()
6      N.signal()
7
8      $A_{idx\ waiting}$.wait()
9      element dequeued.lock()
10     dequeue the image
11     element dequeued.unlock()
12     E.signal()
13
14     interpretting the inputs
```

We chose instead of setting the nthreads - 1 first images to set the 16 - 1 images so that the buffer is full initially of this to calculate. This facilitated the use of the buffer as once the consumer dequeues an image, it only needed to enqueue a new task.

4 Phase 4

For this part, we had to initiate the MPI. Once that was done, we attributed to rank 0 the job of the consumer. Firstly, it was responsible for sending the parameters of the first images to calculate to each of the producer threads (from 1 to size - 1). These producers received the parameters, calculate mandelbrot and send back the calculated Pdata to the consumer of rank 0. When the consumer received the Pdata, it displayed the image, calculate the fps and send back to the producer from which it received the image a new task, i.e new parameters. In order to receive the images in the right order a variable *i* kept track of the process number from which the consumer is waiting to receive the image.

Unfortunately, we did not have the time to connect to the ms800 machines. But we tested the code on our computer. The result seemed quite disappointing as it was really slow.