# SP programming homework4 Report

## Multithread Implementation

- **Communication between threads:**

  First, I use a variable `cur` to record how many iteration has been calculated, and use `done` to record how many spawned threads have completed their work for the current state. The value of `cur` is set to be -1 initaially. For each state, the spawned thread will enter a `while` loop to wait for the condition `cur == e` satisfied, which means all the spawned threads are allowed to compute $e^{th}$ state currently. So, after all the threads are spawned, the main thread will set `cur` to be 0 and signal all the spawned threads to start running for the current state. And after a spawned thread completes the calculation of the current state, it will set `done` to be `done+1` and signal the main thread to check if `done` is equal to the quantity of the spawned threads. The main thread will always wait for the condtion `done == q` satisfied then it set `cur` to be `cur+1` and signal all the spawned thread to keep running for the next state. In this part, I use `pthread_cond_wait` to wait for the condition satisfied, and use `pthread_cond_signal` to notify all the thread that are waiting for the condtion changed.

- **Computation of each state of the board:**

  I use two board: `board[2]`, `board[e%2]` is two $e^{th}$ the current state, and `board[(e+1)%2]` is to record the $(e+1)^{th}$ state.

- **Access to the shared resource:**

  Any shared resource is together with a mutex, so if any thread want to access or modify any resource it will need to lock the resource using `pthread_mutex_lock`.

- **Work assignment:**

  If $row > column$, every spawned thread will be assign a $((row/n) \cdot column)$-sized region except for the last spawned thread. If $row < column$, every spawned thread will be assign a $(row \cdot (column/n))$-sized region except for the last spawned thread. The remaining part of the work will be assigned to the last one. The reason I use this way to distribute the work is to make sure that every thread can get the similar work load and no thread is idling when input size is imbalanced.

# Multiprocess Implementation

The implementation is similar to the multithread part. The only significantly different part is the way of communication between processes and how I shared resource in multiprocess.

- **Communication between processes:**

  I use `SIGUSR2` to notify child processes that all the child processes has completed their work so they can keep running for the next state. After a child process completes its work, it will send a `SIGUSR1` or `SIGUSR2` to notify the parent process to add 1 to the variable `done`. Once `done == 2`, the parent process will send `SIGUSR2` to the children. The first forked child always sends `SIGUSR1` to the parent and the second one always sends `SIGUSR2` if they need to notify the parent process. The way I let the child processes send different signals is to make sure the signal parent process received will not be overwritten. I also used `sigprocmask` and `sigsuspend` properly to avoid signal lost.

- **Share resource:**

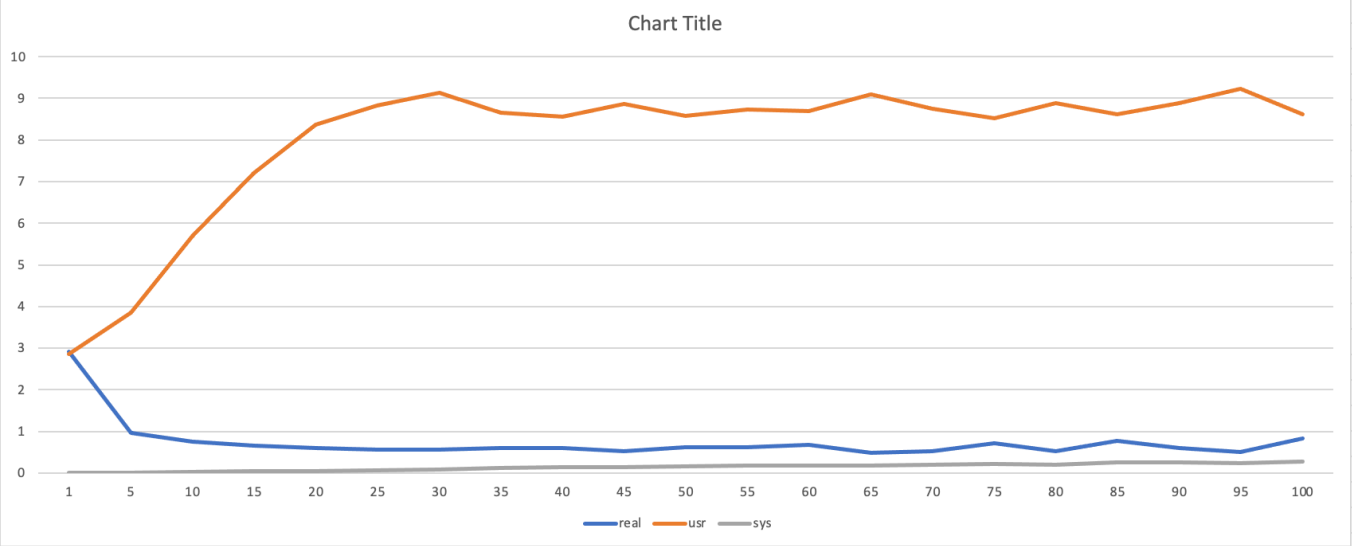
  Here I use `mmap` to share the resource so they can calculate the result together.

# Comparison and comment

```
b08902149@linux15 [~/SP_prog4] time ./main -t 2 ./sample_input/in1.txt out.txt

real    0m0.021s
user    0m0.003s
sys     0m0.000s
b08902149@linux15 [~/SP_prog4] time ./main -t 2 ./sample_input/largeCase_in.txt out.txt

real    0m2.259s
user    0m4.402s
sys     0m0.001s
b08902149@linux15 [~/SP_prog4] time ./main -t 20 ./sample_input/in1.txt out.txt

real    0m0.010s
user    0m0.005s
sys     0m0.000s
b08902149@linux15 [~/SP_prog4] time ./main -t 20 ./sample_input/largeCase_in.txt out.txt

real    0m0.797s
user    0m6.737s
sys     0m0.021s
```

- **Using 2 thread versus Using 20 threads:**

  Because there are more threads that can work at the same time, the real CPU time when using 20 threads is much less than that when using 2 threads. The user CPU time is higher when using 20 threads, the reason is specified in the next part.




- **Comparison with different number of threads:**

  The user CPU time increases when the number of threads increases because the number of calling `pthread_create` is more. Besides, when using more threads, we need more time for the user-level thread to context switch and the overhead will be larger. The marginal benefit of using multithread will decrease when the number of thread increases. Although the execution time decreased in the beginning, the benefit from an additional thread start being eliminated by the overhead when using more thread.

```
b08902149@linux15 [~/SP_prog4] time ./main -t 2 ./sample_input/in1.txt out.txt

real    0m0.030s
user    0m0.000s
sys     0m0.003s
b08902149@linux15 [~/SP_prog4] time ./main -p 2 ./sample_input/in1.txt out.txt

real    0m0.012s
user    0m0.003s
sys     0m0.000s
b08902149@linux15 [~/SP_prog4] time ./main -t 2 ./sample_input/largeCase_in.txt out.txt

real    0m2.345s
user    0m4.442s
sys     0m0.020s
b08902149@linux15 [~/SP_prog4] time ./main -p 2 ./sample_input/largeCase_in.txt out.txt

real    0m2.125s
user    0m4.093s
sys     0m0.010s
```

- **Using 2 processes versus Using 2 threads:**

  I expected that the system CPU time when using multiprocess should be higher than that when using multithread because system calls and context switch should spend more time, but the result didn't match my expectation.