CS Games 2016

Parallel Programming Competition

Participants	**2**
Workstations	**1**
Value	**5%**
Duration	**3 hours**

Simulations

Ever since the catastrophe that almost eradicated every form of life on Earth, scientists of the Dome have always dreamt of evaluating the survival chances of primitive life forms outside of the Dome.

To do so, a life simulation program is being developed internally within the Dome.
It simulates the survival chances of primitive cells in a hostile environment, with scarce resource availability.

Each cell can reproduce when in an adequate environment, but can fall prey to overpopulation.
The major issue this program faces is its large resource consumption.
When the simulation is run in a realist environment with a large dataset, the workload becomes too intense to be executed in an acceptable timeframe.

We must find a way to harness the multi-core capabilities of the machines of the Dome to improve its runtime efficiency. By doing so, we might observe the results of such a simulation and determine whether or not, life is possible outside of the Dome...

Problem Description

In this challenge, you will need to optimise a simple game of life program and make it as efficient as possible.
To do so, you will need to harness the parallel capabilities of your machines and think of clever tricks to optimise the simulation as much as possible.

The input for the program will be matrix containing either 0 or 1.
0 means the cell is dead, and 1, the cell is alive.

Each cell's state will be determined by its 8 immediate neighbours. For each cell at the ends of the grid, we shall consider the grid circular.
As such, when cell (0, 0) will want to check on its upper neighbours, we will check the state of cells at y-coordinates (matrix_height - 1).

The state of a cell at the next iteration will depend on the following rules:

- ❏ A cell with less than 2 neighbours dies;
- ❏ A cell with exactly 2 neighbours keeps its state;
- ❏ A cell with exactly 3 neighbours lives;
- ❏ A cell with more than 3 neighbours dies.

Example

At an instant t, let us consider the following grid:

1	0	1	0	0
1	0	1	0	0
1	0	1	0	0
1	0	1	0	0
1	0	1	0	0

After one iteration, we will obtain the following result:

1	0	1	1	1
1	0	1	1	1
1	0	1	1	1
1	0	1	1	1
1	0	1	1	1

Since the program needs to be as efficient as possible, we mandate the use of either C or C++ for this challenge.
You may use the standard libraries of the authorised languages, along with the parallel libraries **pthreads** and **Threading Building Blocks**.

Input Files Specification

A description file is based on the following specification:

- The first line is the size of the matrix. To simplify the problem. matrices are necessarily squared.
- Each following line will be the content of a line of the matrix, the state of a cell will be incidated through 0 for dead and 1 for alive.

Your program must work by taking two arguments: the path to the description file, followed by a number of iterations which must be performed.

Correction

We will note your program according to its performance.
The runtime will be measured using GNU Time, the first run will be discarded and the average time of the following 5 runs will be kept as reference.

A leaderboard will then be constituted to determine the winners.
In case of near-equivalence, an examination of the source code will be the last criteria.
Spatial use will also be a concern you must take care of since some of the inputs we will try your programs on will be huge and will need careful use of resources on your part.

Due to the sheer size of these inputs, we will only give you a part of these for testing and performance evaluation from your development machines.

To ensure proper testing, you must turn a solution compilant with the templates we give you as reference with this description.

The Makefile should be modified as necessary to ensure proper building of your solution while not changing the output binary file's name.
Any failure to comply will result in a 5% penalty.

Appendix

Parallel libraries usage manual:

Thread C

```c
int pthread_create(pthread_t restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(start_routine)(void*),
                   void *restrict arg);
```

Mutex C

```c
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                       const pthread_mutexattr_t *restrict attr);

int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Thread C++

```cpp
std::thread leThread (foo);
```

Mutex C++

```cpp
lock();
Unlock();
```

Suggested optimizations

Since this challenge is mostly an optimization problem, participants are encouraged to optimize their parallelization strategy and their sequential algorithm.

Such optimizations can be done through space partitioning, by removing dead areas for an iteration, therefore reducing the quantity of data to compute.

In addition to this optimization, trying different parallelization strategies can be beneficial depending on your algorithm.
It can either be done through the use of a thread pool and task-bag combination for dynamic distribution of chunks from the matrix.

Micro-optimizations can also have an impact on your program's behaviour at runtime, since they have an impact on the use of cache for instance.
Caching your resources locally can also be beneficial to avoid fetching data from a random place in memory.