CS3210 Assignment 1

Joshua Tan Yin Feng (A0199502Y), Kishen Ashok Kumar (A0204777X)

Introduction

In this report, we will discuss our use of the Pthread and OpenMP library to improve the execution time of the Game of Invasion program. We first provide an analysis of the original Game of Invasion program, followed by our experiment methodology. Next, we outline our use of the Pthread and OpenMP libraries. In each section, we specify the details of our implementations for both libraries, our design considerations, implementation specific results. We then wrap up with a discussion across all 3 implementations.

Analysis of Game of Invasion

In our preliminary analysis, we focused on the "goi.c" file as the majority of the simulation logic is implemented here. Reviewing the program, we made 4 key observations:

1. the computational logic to update is the same for each cell
2. the value of each new cell is only dependent on the old world state and not on the value of the new value of its neighbours
3. world states are immutable and a new world state is created for each generation
4. we need to update the variable `deathToll` throughout the entire program

From (1), we intuitively thought of potentially using a Single Program Multiple Data (SPMD) parallel programming pattern to optimise the program. Furthermore, given the immutability of world states and with each new cell updating independently from each other, we can update each cell in the new world state in any arbitrary order. Implicitly, we do not need to worry about potential race conditions or synchronisation issues when updating the new world state. The only thing we need to ensure is that every cell for the new world state is fully updated before we move on to the next generation. Lastly, if we were to parallelise our program, we will need to ensure that `deathToll` is updated correctly. If `deathToll` is a shared variable, then we will require synchronisation constructs such as a mutex to ensure that we do not run into race conditions.

In "goi.c", the original program utilises a nested *for* loop to iterate through each cell in the current world state to get its next state and updates the new world state accordingly. This nested *for* loop is where the majority of the computational work is done and it is where we focused our parallelization efforts on.

```c
// get new states for each cell
for (int row = 0; row < nRows; row++)
{
    for (int col = 0; col < nCols; col++)
    {
        bool diedDueToFighting;
        int nextState = getNextState(world, inv, nRows, nCols, row, col, &diedDueToFighting);
        setValueAt(wholeNewWorld, nRows, nCols, row, col, nextState);
        if (diedDueToFighting)
        {
            deathToll++;
        }
    }
}
```

Figure 1. `getNextState` *for* loop in original sequential implementation

Design considerations

We used Foster's Design Methodology to identify how we could decompose the overall problem into smaller tasks. Each of these tasks could then be done in parallel before combining the results together. These tasks would be done using threads since we are using Pthreads and OpenMP.

Identifying the small tasks

Since the problem is characterised by its grid nature, it was intuitive to think of decomposing each task at the cellular level. We initially define each task as follows:

> *Each task is responsible for updating a single cell state and the shared `deathToll` variable.*

Every task would need to spawn a new thread in order to complete its work. Once the work is completed, the thread would exit and a new one has to be spawned for another cell.

Communication between tasks

Each thread will require access to these data in order to work:

1. the current world state
2. the new world state
3. the invasion plans (if any)
4. the row and column being worked on
5. the `deathToll` variable

Thankfully, the current world state, new world state and invasion plans need not be copied into each thread. Instead, a pointer to their memory locations would suffice. Similarly, since the `deathToll` variable is shared between threads, a pointer to its memory location would do. However as mentioned before, a synchronisation construct such as a mutex would be required to prevent race conditions. The row and column variables would have to be copied into each thread since it is unique for every task.

Reducing the costs of parallelisation

Based on our above definition of the work done by each task, it is easy to observe that we would incur huge total costs from updating each cell. The problem is made worse the lesser the maximum number of threads available. In order to amortise the costs of creating a thread, we decided that each task should do at least one row of computation. To further eliminate reusing threads, we decided to split the total number of rows amongst each task evenly. This way, we would only need to create no more than the maximum number of threads in each generation.

Since `deathToll` is a shared variable, there would be costs involved in the creation, locking and unlocking of mutexes. We decided that it might be more efficient to remove this shared memory dependency by forcing each thread to update a local variable tracking the total number of deaths as it updates every cell in its assigned row. Once all the threads are completed, the master thread would then accumulate the total number of deaths from each thread and update the overall `deathToll` variable.

With these 2 changes, the revised definition of a task is as follows:

Each task is responsible for updating every cell state in its assigned row(s). In addition, it will also update a local `deathToll` variable which will then be communicated back to the master program once the task is completed.

Mapping to processors

Since we are using Pthreads and OpenMP, the libraries will handle the mapping for us and we do not need to worry about such a problem.

Experiment Methodology

Data Collection

Before we go into the details of our implementations, we will briefly elaborate on our experiment methodology. To collect data, we used the `perf` command. The metrics we were interested in measuring were:

- real time elapsed
- number of page faults
- number of L1-dcache-loads
- number of L1-dcache-load-misses
- number of branches
- number of branch misses
- number of instructions
- number of cycles.

Knowing the real time elapsed would allow us to objectively compare the execution speeds across our implementations while the other metrics would potentially help us explain our results and any anomalies. Note that we use the terms “real time elapsed” and “execution time” interchangeably throughout this document, but they effectively refer to the same thing.

Since we are measuring the execution times against the number of threads, we run each implementation 5 times for every thread count. The `perf` command helps us to find the averages automatically so we don’t have to. It even reports to us the standard deviation across the repeats so that we are able to identify any anomalies and rerun the tests if needed.

In general, the overall command, along with its options, used was something like this:

```
perf stat -e
page-faults,L1-dcache-loads,L1-dcache-load-misses,branches,branch-misses,instructions,cycles -r 5
-- ./goi.out sample_inputs/sample6.in death_toll.out 8
```

Machines used

We used the machines provided by the School of Computing at the National University of Singapore. The details of the machines used can be found in Annex A.

Original Sequential Program Execution Time

In order to benchmark our implementations, we recorded the execution times for the original sequential program. This can also be found in Annex A.

Pthread: Master-Worker

Implementation details

Having chartered our roadmap, we now had a clear idea of how to begin parallelising our program. The first parallel programming pattern we used was a Master-Worker pattern.

Since we are using threads, we needed to store all the arguments needed in a struct.

```c
// struct to contain the args for tasks
typedef struct TaskArgs {
    const int *world;
    const int *inv;
    int *wholeNewWorld;
    int nRows;
    int nCols;
    int startRow;
    int endRow;
    int retVal;
} TaskArgs;
```

Figure 2. struct data structure to store `TaskArgs`

The `world`, `wholeNewWorld` and `inv` are all pointers to the current world state, new world state and invasion plans for that generation respectively. Variables `startRow` and `endRow` are start and end range of rows that are assigned to each task. `nRows` and `nCols` are just auxiliary constants to be used to denote the world size. `retVal` is the local death toll counter that is to be updated by the task. Once the worker thread has finished its task, the master thread would then acquire this value and update the global `deathToll` counter.

To ensure fair workload across the threads, we split the total number of rows evenly across the different tasks. If the number of rows is not divisible by the maximum number of threads, we make sure that each task does at most one extra row of computation. The programming logic for achieving this is shown in figure 3.

Our implementation removes the need to free the memory allocated for each `TaskArgs` variable after each generation since the `TaskArgs` can be reused. The only thing that needs to be changed are the pointers to the different worlds and invasion plans respectively. This would potentially save us some precious time.

```c
// Args for each thread
TaskArgs *tArgs[nThreads];

/*** Initialise the thread args ***/
// Number of rows each thread works on
int rowsPerThread = nRows / nThreads;

// When nRows % nThreads != 0, we want to
// split the remainder rows equally amoung
// each row (i.e. every thread should take on at
// most 1 more row than rowsPerThread)
int leftoverRows = nRows % nThreads;

int startRow = 0, endRow;
for (int threadIdx = 0; threadIdx < nThreads; threadIdx++)
{
  tArgs[threadIdx] = (TaskArgs*) malloc(sizeof(TaskArgs));
  tArgs[threadIdx]->startRow = startRow;
  endRow = startRow + rowsPerThread;
  if (leftoverRows > 0)
  {
      leftoverRows--;
      endRow++;
  }
  tArgs[threadIdx]->endRow = endRow;
  tArgs[threadIdx]->nRows = nRows;
  tArgs[threadIdx]->nCols = nCols;
  tArgs[threadIdx]->retVal = 0;
  startRow = endRow;
}
```

Figure 3. Initialising each `TaskArgs` with information

In each generation, we will spawn a thread for each task. Before the thread starts, we ensure that the pointers to the different worlds and invasion plans are set correctly. The master program then waits for all threads to finish their tasks before accumulating the total death count from each thread. Once completed, the next generation may proceed. As mentioned, we did not have to free and reallocate memory after each generation. This is because we can reuse the `TaskArgs` and only need to change the pointers to the different worlds and invasion plans.

```c
for (int threadIdx = 0; threadIdx < nThreads; threadIdx++) {
    // Set args for each thread
    tArgs[threadIdx]->world = world;
    tArgs[threadIdx]->wholeNewWorld = wholeNewWorld;
    tArgs[threadIdx]->inv = inv;

    int rc = pthread_create(&threads[threadIdx], NULL, threadWork, tArgs[threadIdx]);
    if (rc) {
        printf("Error creating thread\n");
        if (inv != NULL)
        {
            free(inv);
        }
        free(world);
        free(wholeNewWorld);
        exit(1);
    }
}

// Join threads
for (int threadIdx = 0; threadIdx < nThreads; threadIdx++) {
    pthread_join(threads[threadIdx], NULL);
    deathToll += tArgs[threadIdx]->retVal;
    tArgs[threadIdx]->retVal = 0;
}
```

Figure 4. Master thread creating worker threads to execute their tasks before joining them at the end

The code for this Master Worker implementation can be found in “**goi_master_worker.c**”.

Results

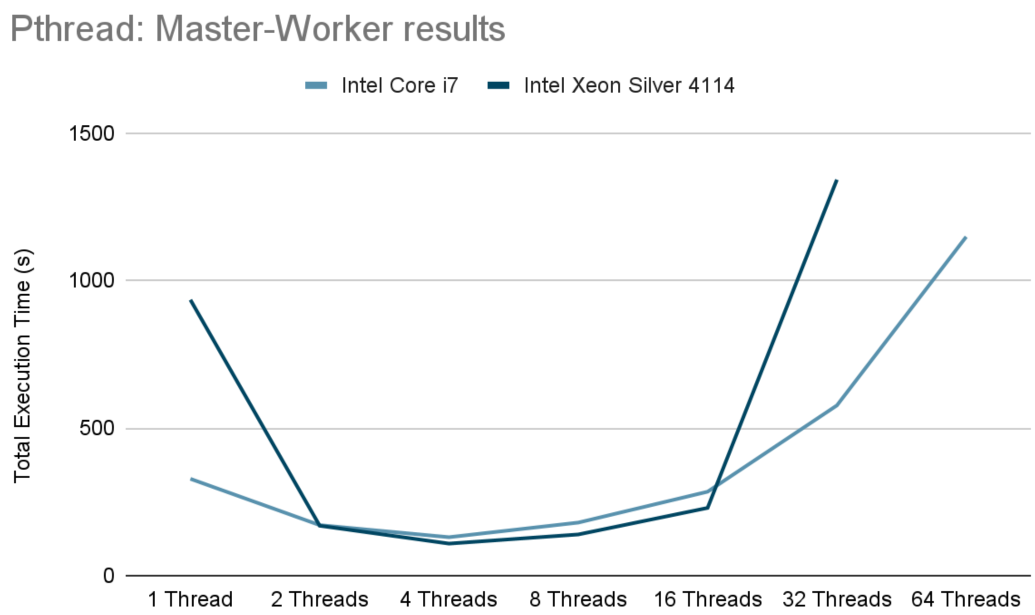

Figure 5. Results for Pthread Master-Worker implementation

Compared with the results from the original sequential program, our pthread implementation outperformed it on both machines. Both machines peaked at around 4 threads and were approximately twice as fast as running compared to running only 1 thread.

The full results can be found in Annex B.

Pthread: Thread Pool

Despite achieving a good speedup, we felt that the program was still incurring unnecessary costs with the repeated creation and deletion of threads after every generation. We believe that this recurring process is very costly especially with a large number of generations. Thus, we wanted to find a way to remove this inefficiency. To do this, we turned to using a thread pool parallel programming pattern. Using a thread pool implementation would allow us to reuse the threads instead of recreating them every generation. Instead of creating our own thread pool from scratch, we borrowed a pthread pool implementation from here: https://github.com/jonhoo/pthread_pool.

In our Master-Worker implementation, we only wanted to create at most the maximum number of threads for every generation to prevent excess threads. This was done by assigning each task a fixed number of rows it must work on. Additionally, if a thread finishes its task faster than others, it will just exit. However, with a thread pool implementation, we are no longer bound by such constraints. This is another potential benefit brought about with thread pools. With this, we made some revisions to our definition of a task:

> *Each task is required to work on a predetermined number of rows at a time. In addition, it will also need to update the shared variable `deathToll` upon completion.*

With this new definition, whenever a thread finishes its task faster than the others, it does not need to exit. Instead, it can take on another task and continue to work on it. However, because of the way our thread pool is implemented, we are unable to let our master program accumulate the individual death tolls in each thread. Hence, we need to ensure that each task updates the shared variable `deathToll` directly once it has completed.

Implementation Details

The thread pool consists mainly of 2 things: the task arguments queue and the threads. Since every thread is going to do the same task, the task need not be redefined, only the arguments to the tasks. Therefore, it is a task arguments queue instead of a task queue.
Without going too much into the implementation specifics, task arguments are added to the queue and the threads would acquire these arguments to work on them. When there are no arguments in the queue, the threads would wait on a conditional. When new arguments are added, the threads would be woken up and start their tasks.

As seen in figure 6, we are creating tasks arguments. Note the following changes to `TaskArgs`:

- there is no `startRow` and `endRow` variable, only a single `row` variable
- there is a pointer to the share variable `deathToll`
- there is a pointer to the mutex `lock`

```
typedef struct taskArgs {
    const int* world;
    const int* inv;
    int *wholeNewWorld;
    int nRows;
    int nCols;
    int row;
    int *deathToll;
    pthread_mutex_t *lock;
} TaskArgs;
```

Figure 6. New structure for `TaskArgs` for Thread Pool implementation

The `row` variable here simply refers to the starting row for the task. We create these task arguments and add them to the thread pool queue. The task would then work on `TASK_SIZE` number of rows, starting from the variable row. `TASK_SIZE` is a predefined macro in the program and can be modified accordingly. We also add a pointer to the shared variable `deathToll` and the mutex `lock`. Each thread would need to gain access to the mutex before it updates `deathToll`.

As before, we need to wait for the entire world state to be updated before we can continue. Hence the program needs to wait for the task arguments queue to be empty before it can proceed to the next generation.

```c
// get new states for each cell
for (int row = 0; row < nRows; row += TASK_SIZE)
{
// each task operates on 3 rows
    TaskArgs *tArgs = (TaskArgs *)malloc(sizeof(TaskArgs));
    if (tArgs == NULL) {
        printf("Failed to mem alloc for task args\n");
        pool_end(p);
        free(world);
        free(wholeNewWorld);
        if (inv != NULL) {
            free(inv);
        }
                exit(1);
    }

    tArgs->world = world;
    tArgs->wholeNewWorld = wholeNewWorld;
    tArgs->inv = inv;
    tArgs->nRows = nRows;
    tArgs->nCols = nCols;
    tArgs->row = row;
    tArgs->deathToll = &deathToll;
    tArgs->lock = &lock;
    pool_enqueue(p, tArgs, 1);
}
pool_wait(p);
```

Figure 7. Creating `TaskArgs` and adding them to the task arguments queue. The pool waits for all tasks to complete before the program continues.

```c
pthread_mutex_lock(tArgs->lock);
(*(tArgs->deathToll))++;
pthread_mutex_unlock(tArgs->lock);
```

Figure 8. Locking and unlocking mutex to update shared variable `deathToll`

The code for this Thread Pool implementation can be found in “goi_pthread.c”. The actual thread pool implementation can be found in “pthread_pool.c”.

Results

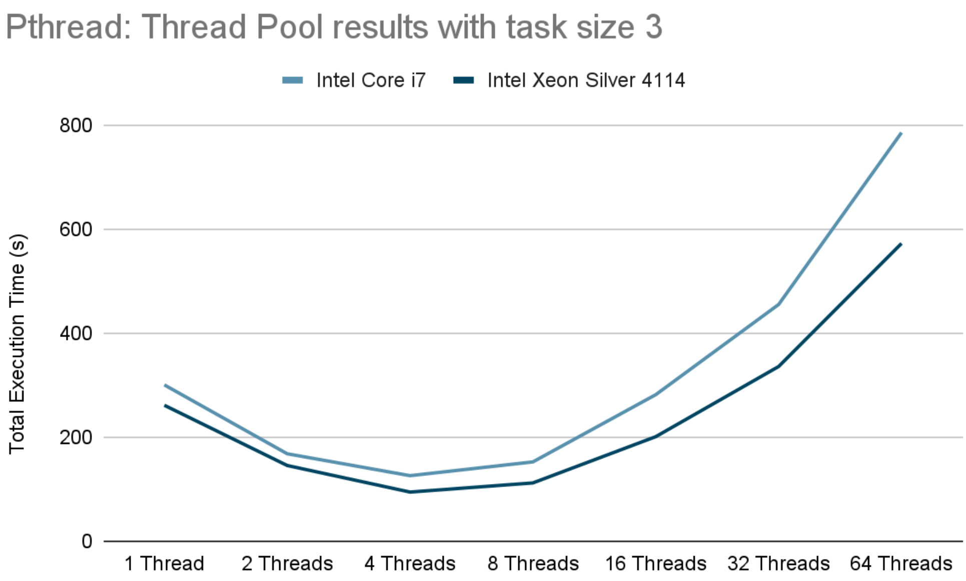

Figure 9. Results for Thread Pool implementation with a task size of 3

We tested our program using different values of `TASK_SIZE`. Initially with a `TASK_SIZE` of 1, our Thread Pool implementation fared better than our Master-Worker implementation. With a `TASK_SIZE` of 3, we saw the best improvements.

Thread Pool Discussion

Our results confirmed our beliefs that a Thread Pool would run faster than a Master-Worker implementation. We have included a figure further below (figure 13 and 14) illustrating the differences between the 2 as well as our OpenMP implementation.

Increasing the value of `TASK_SIZE` also produced diminishing returns. When going from a `TASK_SIZE` of 2 to 3, the improvements in execution time were minimal and we believed that going any further would only produce similar or worse results. We posit that when each task is given a larger number of rows to work with, there would be less context switches. A task size of 2-3 is optimal and going any further would only make each task.

The full results can be found in Annex C.

OpenMP

Having identified the main area to parallelise, implementing an OpenMP solution was relatively straightforward.

Implementation Details

Since OpenMP relies on compiler directives, there wasn't much to be modified from the original sequential program. We simply added the `#pragma` directives to parallelise the *for* loop and added a critical section to synchronise the updating of `deathToll`.

```c
#pragma omp parallel for shared(deathToll, wholeNewWorld) private(row, col, diedDueToFighting)
for (row = 0; row < nRows; row++)
{
    for (col = 0; col < nCols; col++)
    {
        int nextState = getNextState(world, inv, nRows, nCols, row, col, &diedDueToFighting);
        setValueAt(wholeNewWorld, nRows, nCols, row, col, nextState);
        if (diedDueToFighting)
        {
            #pragma omp critical
            {
                deathToll++;
            }
        }
    }
}
```

Figure 10. Adding `#pragma` directives to `getNextState` *for* loop

To set the number of threads created by the OpenMP library, we simply used the `omp_set_num_threads` function.

```c
// set OMP thread count
omp_set_num_threads(nThreads);
```

Figure 11. Setting the number of threads for OpenMP

The code for this OpenMP implementation can be found in “goi_omp.c”.

Results

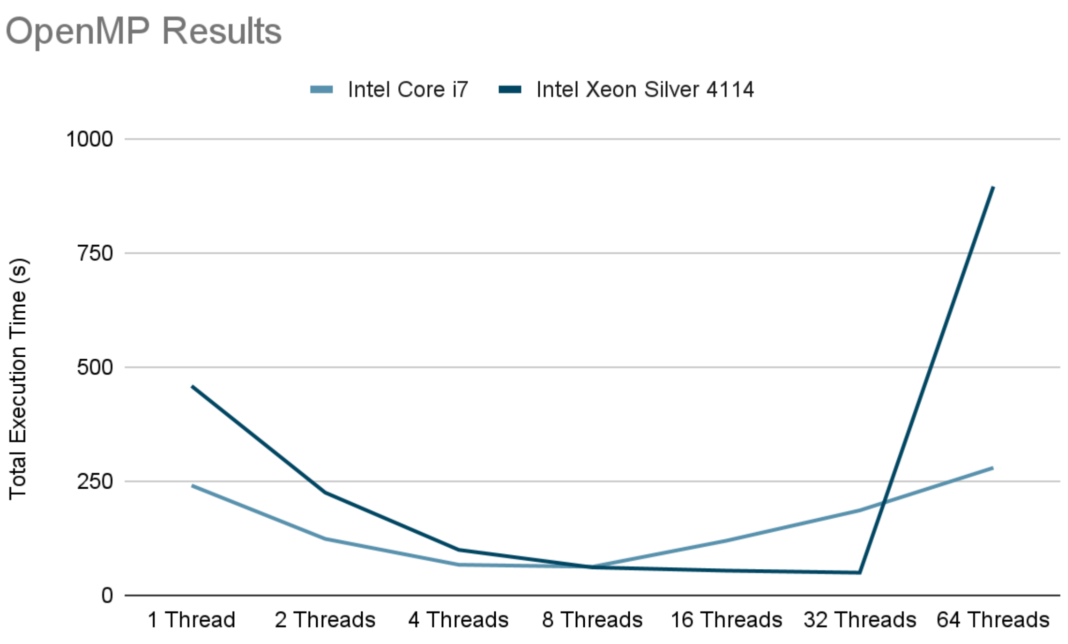

Figure 12. Results for OpenMP implementation

Figure 12 displays similar results to the Pthread implementations. For both machines, they achieved peak performance as the thread count approaches the maximum number of threads allowed to run on the machines respectively.

The full results can be found in Annex D.

Combined discussion across the 3 implementations

Before we proceed any further, we state that any mention of Pthread implementation in this section refers to both the Master-Worker and Thread Pool implementations unless specified otherwise.

Comparison between Pthread and OpenMP

On both machines, our OpenMP implementation performs better than our Pthread implementation.

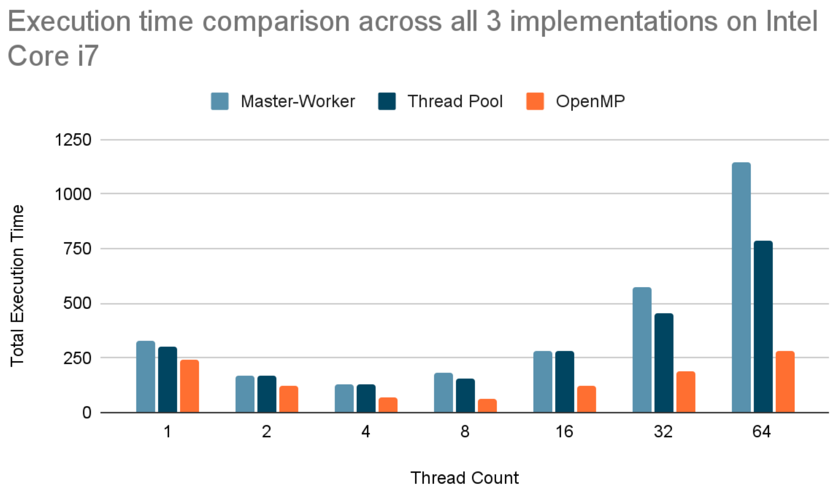

Figure 13. Execution time comparison across all 3 implementations on Intel Core i7

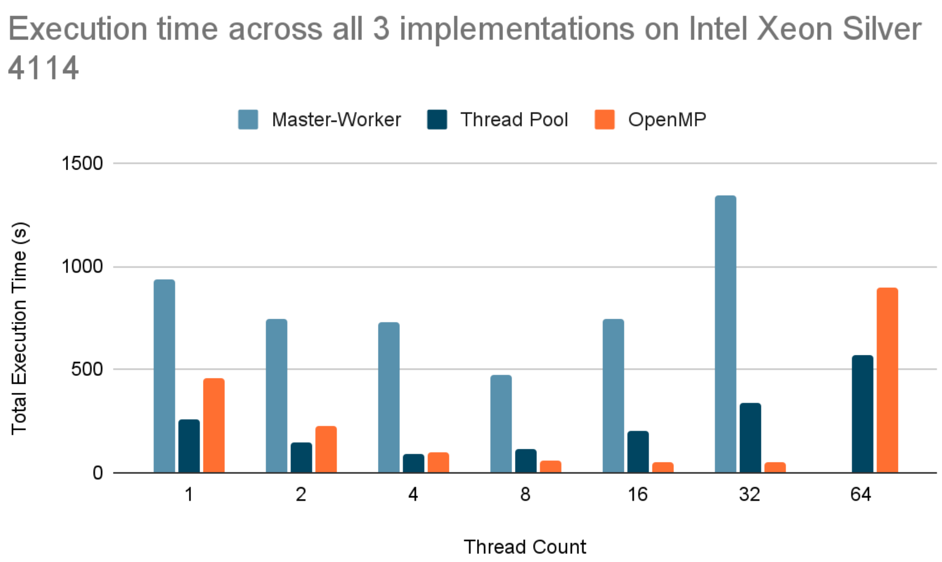

Figure 14. Execution time comparison across all 3 implementations on Intel Xeon Silver 4114

With regards to the comparison between OpenMP and the Pthread implementations, we are limited by our lack of knowledge about the implementation of OpenMP and the behaviour of compilers. However, we still attempt to make educated guesses to account for the difference between the two implementations. We believe that it could be due to the following reasons:

1. the compiler managed to optimise the thread implementations better than we did in the Pthread implementations

2. memory management was much better in the OpenMP implementation than the Pthread implementation

Elaborating further on (2), the OpenMP implementation in general had much fewer L1-dcache-loads and load misses than the Pthread implementations on both machines. With fewer loads and a smaller percentage of load misses, it probably reduced the time taken for memory accesses. A similar trend was also observed for percentage of branch misses. We believe that this accounts for why OpenMP performs slightly better than our Pthread implementations.

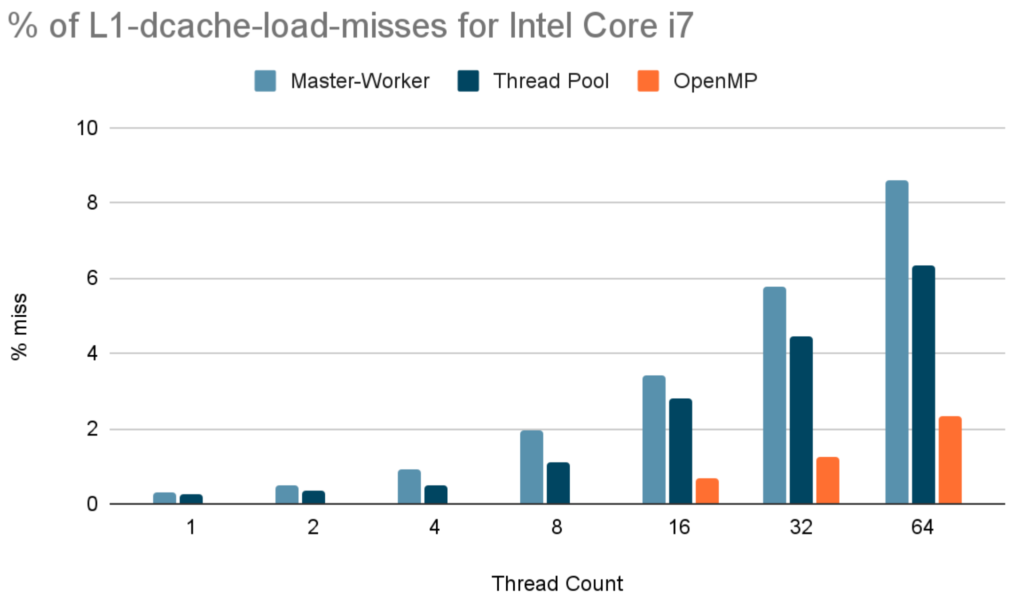

Figure 15. Percentage of L1-dcache-load-misses for Intel Core i7 across all 3 implementations

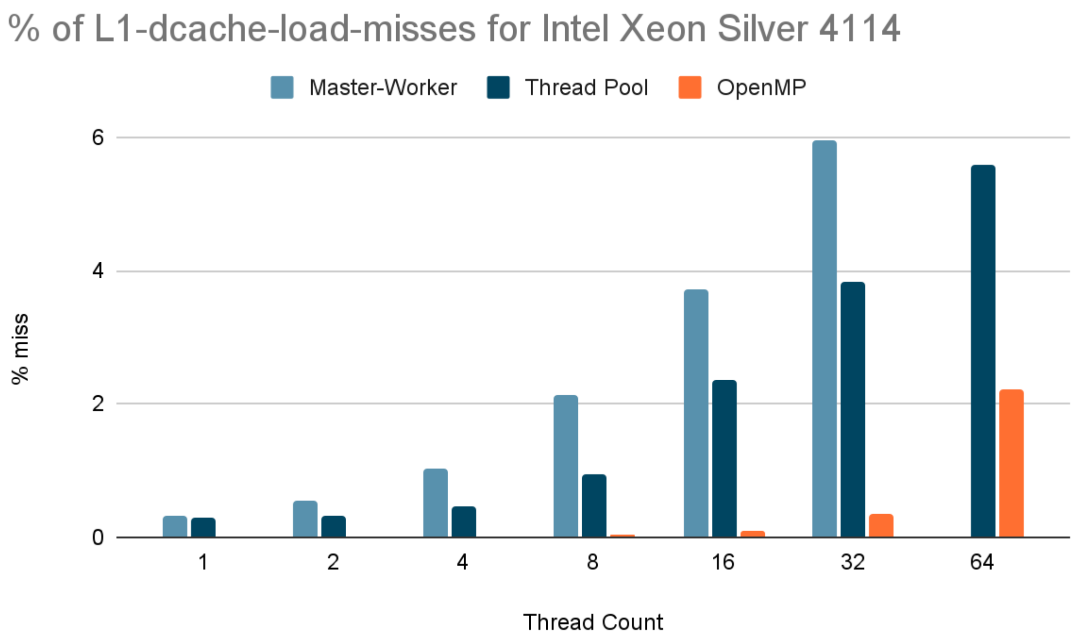

Figure 16. Percentage of L1-dcache-load-misses for Intel Xeon Silver 4114 across all 3 implementations

Peak performance

Previously, we skipped any discussion on why execution times were the lowest as the thread count approaches the maximum concurrent thread limit on each respective machine. We will revisit those discussions now.

It should be trivial to say that any thread count above the maximum concurrent thread limit would not benefit the execution of the program. In fact, the excess threads only consume more resources – more threads only means more expensive context switches. Furthermore, with every context switch, the next thread might need to access a different memory location than the current thread, resulting in caching issues.
When thread count is lower than the maximum concurrent thread limit, the execution speeds are also slow because the program could still benefit from having another thread sharing the workload. Optimal execution times are achieved when the thread count is equal to the maximum concurrent thread limit.

Interestingly, our Pthread implementations for Intel Xeon Silver 4114 started to deteriorate in performance past 4 threads whereas the OpenMP implementation kept improving marginally till 32 threads before deteriorating. We are unable to fully explain such a behaviour and can only posit that the OpenMP implementation optimised the use of threads much better than our Pthread implementations.

Other non-implementation specific details

Copying start world and invasion plans

The original sequential implementation used *`for`* loops to copy `startWorld` and `invasionPlans`. Without compiler optimisations, this process is definitely slower than using `memcpy`. However, we do not know if the compiler would automatically optimise this process for us. Hence we replaced the *`for`* loops with an explicit call to `memcpy` instead. Despite this, we were unable to experience any noticeable speedups. Perhaps if the world was smaller and the work done memory copying operations was comparable to the work done to update the world states, we might be able to see a significant improvement.

Conclusion

Our parallelisation efforts using both Pthreads and OpenMP on the Game of Invasion program proved to be successful. However, we were unable to fully account for the differences between Pthread and OpenMP implementations. In terms of parallel programming patterns, we chose patterns which we felt best suited the circumstances. For example, if this was a distributed architecture, we might have opted for another pattern instead.
In conclusion, this assignment has greatly aided our learning and application of concepts taught in the class.

Compilation Instructions

In order to compile our program, navigate to the directory with all our code and execute the command "**make build**". This will compile our programs and produce three executables : **goi_threads**, **goi_omp** and **goi_master_worker**.
goi_threads and **goi_omp** are part of our submission requirements and **goi_master_worker** is included for completeness of our report.
The following details how the executables map to the programs we referenced in our report:

goi_threads - Our pthread solution that makes use of a task pool

goi_omp - Our OMP solution
goi_master_worker - Our pthread solution without a task pool

Annex A

Here are the details of the sample used and the machines used to test our implementations. As a baseline for comparison, we also include the results from running the original sequential program on the respective machines here.

Sample Details

File name: sample6.in
Number of generations: 1000000
World Size: 50x60

Machine Details

Machine 1: Intel Core i7-7700
Hostnames: soctf-pdc-012, soctf-pdc-013, soctf-pdc-014
Specifications:

- 3.60GHz
- 4 cores (8 threads)
- 32GB DDR4
- 500GB 7200RPM SATA HDD

Machine 2: Dual-socket Intel Xeon Silver 4114
Hostnames: soctf-pdc-019, soctf-pdc-020
Specifications:

- 2.20GHz
- 2*10 cores (40 threads)
- 64GB DDR4
- 1TB Seagate 7200RPM HDD

Original Sequential Results

Machine	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
Intel Core i7-7700 (soctf-pdc-013)	239.4815	66	1130337221234	67276034	440725812907	211423392	2567555016260	999979829458
Dual-socket Intel Xeon Silver 4114 (soctf-pdc-020)	216.5662	66	1130325328622	35494312	440696301099	207801210	2567557044306	992934338474

Annex B

Results for Pthread **master-worker** implementations using `perf` command taken over an average of 5 repeats. A NULL value indicates that we did not complete that experiment. This was due to the fact that it took too long to run and the results wouldn't have contributed meaningfully to the discussion.

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
1	328.018	91	1172217894108	3581385888	453163838647	571239209	2648595140253	1099676477659
2	171.108	94	1186614249184	6111539827	465310883323	518521368	2706491838726	1158393046074
4	130.2263	101	1217411530721	11282690512	489026909858	714162244	2816915707038	1330937367858
8	179.7300	8000098	1300606119084	25671822495	553518612915	1436628012	3133696336586	2071586979408
16	284.39	24000101	1459383575070	50045572708	674899087794	2627170096	3716404689999	2592307720792
32	576.760	56000101	1786201759105	103296521553	925270081611	5262028200	4912555919336	3920125179442
64	1148.45	120000106	2444484804489	210666629473	1429251005501	9929978277	7322806857778	6818206227924

Results on Intel Core i7-7700 (soctf-pdc-012)

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
1	934.74	90	1171885469952	3691649683	454736700870	655843925	2658379085432	1387841589579
2	169.05	94	1185587207668	5840227145	464723355323	585486358	2707195839530	1136880728825
4	108.503	97	1214755174806	10541039261	487353009514	608183033	2815722265378	1242843145487
8	139.505	8000100	1295089244186	23058060124	549805828168	1147516290	3121682653236	1616911963006
16	229.455	24000099	1450258447074	45960595109	669479851656	2327157686	3698102187335	2258047565488
32	1342.02	56000100	1831256675997	109137981781	953583230361	5803759473	5066723123038	5011550272719
64	-	-	-	-	-	-	-	-

Results on Dual-socket Intel Xeon Silver 4114 (soctf-pdc-019)

Annex C

Results for Pthread **thread pool** implementations using `perf` command taken over an average of 5 repeats. A NULL value indicates that we did not complete that experiment. This was due to the fact that it took too long to run and the results wouldn't have contributed meaningfully to the discussion.

Task Size: 1 row

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
1	331.59	78	120730006239	3936368121	475156839642	690649402	2765426279211	1443928744049
2	167.984	85	1210985982228	4193301780	476710170358	703562013	2770224351614	1365676836143
4	123.66	93	1220507415235	5708372832	483370803507	890845705	2797888249586	1589859889940
8	152.95	109	1277329395330	13956225157	525814288334	1686728875	3000151357896	2515394826273
16	282.27	144	1488376031637	41673796249	681437531115	4622833741	3746062575621	3944137123401
32	458.03	191	1781592148045	79676047741	900428724576	8464151581	4806060622637	5975720579140
64	788.232	295	2326478616259	148816617711	1307798302358	15213864770	6778824526375	9649085711728

Results on Intel Core i7-7700 (soctf-pdc-012)

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
1	636.17	80	1328661280904	8594240903	566115494556	1525079548	3208266581452	2029166686068
2	379.78	87	1293055753584	11133113053	540491100217	1393177630	3103643174159	1981589103968
4	255.24	101	1303592672017	12695140093	543017064413	1517520012	3080888694315	2031436969658
8	279.78	108	1390317935552	21118068121	606476912553	2525076265	3381107494360	2623255873606
16	1588.033214242	151	1747681077898	66623096564	87127909403	6404653345	4640140011125	4948968487213
32	-	-	-	-	-	-	-	-
64	-	-	-	-	-	-	-	-

Results on Dual-socket Intel Xeon Silver 4114 (soctf-pdc-019)

Task Size: 2 rows

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
1	309.784	80	1227925517403	4686366881	490191276245	815728615	2845773515092	1429392001131
2	174.730	86	1230042863764	5586372289	492072815037	875514706	2850414973008	1496083536682
4	128.5316	95	1227725165888	6447680665	490488747064	974706274	2836868245078	1676975070358
8	180.76	113	1293658122847	15913578758	539976387450	2031818371	3072779391114	2676828596757
16	368.20	140	1518964987226	44855716324	709077789657	5281829218	3894376930601	4272530171371
32	601.01	196	1965884600599	102891284944	1044385552030	10928466093	5524669650858	7309474297327
64	1098.21	301	2777384386687	207103798561	1654891892675	21052606958	8494869330514	12825019521528

Results on Intel Core i7-7700 (soctf-pdc-014)

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
1	267.869	80	1226479427576	4675742074	489726478376	751263746	2842181225374	1322419776959
2	148.8777	84	1227276802579	5654786907	490342653926	808763766	2845313330919	1355862989292
4	99.2393	94	1235906569754	7163164520	496995206814	999699080	2877474529290	1435340806960
8	118.551	110	1291803608948	13721654503	539213816369	2092186485	3084785991277	1715708858983
16	238.03	143	1520397730452	41613587283	711187435299	5388014233	3936191156962	2863549572236
32	422.14	195	1884126018056	84740100854	985205329195	10278036968	5290298176016	4617803467619
64	880.14	301	2593003554879	167364723513	1519644804378	19587892197	7928198721077	7986470159451

Results on Dual-socket Intel Xeon Silver 4114 (soctf-pdc-020)

Task Size: 3 rows

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles

1	300.575	78	1207190231084	3345059400	475205781507	665888106	2767149029723	1329672812956
2	168.192	84	1209249751683	4149377757	476669974815	709769815	2774690013686	1372340186460
4	126.081	95	1219443808831	5808268060	484296191283	897812350	2804876305237	1633271383356
8	152.546	112	1275992479427	13906098889	526815417834	1686924989	3009750130318	2528256355015
16	281.802	140	1485280868137	41349245588	683494369711	4657055159	3768054677481	3965296017359
32	455.093	197	1775260648827	78697773627	901464820834	8476413078	4828453572039	5979457181895
64	785.460	299	2319119374515	147417251857	1310888361495	15271517209	6821779546816	9683054646484

Results on Intel Core i7-7700 (soctf-pdc-014)

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
1	261.391	80	1206172960969	3455576039	474634133645	629501089	2766933927435	1255945769161
2	145.6567	85	1205822670140	3980286080	474379935807	660245313	2765678024205	1278250881297
4	94.574	94	1219529935592	5806241359	484744586379	854936455	2816518575415	1358433705329
8	112.146	113	1275333206164	12172036676	526966262155	1871979676	3023848766653	1630608110055
16	201.07	142	1458035115365	34404801207	664387268957	4262585141	3704582506675	2539688454971
32	335.87	195	1724630280605	65959341422	865437006423	8011757043	4697672368589	3833490889925
64	572.38	302	2234485117350	124850107955	1249743345801	14446491286	6596020156378	6224948879743

Results on Dual-socket Intel Xeon Silver 4114 (soctf-pdc-020)

Annex D

Results for OpenMP implementations using `perf` command taken over an average of 5 repeats.

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
1	240.518	90	1130681150050	148408319	443906236040	213572151	2571919570982	1004963235031
2	123.6972	90	1130841012793	113471082	444007370546	220508838	2572539950971	1010396989325
4	66.9812	96	1131443148843	241878171	444823978845	231952934	2575612695156	1068323216902
8	62.6002	104	1132720495486	458105351	446831218874	267000019	2583162334684	1995330871952
16	119.540	122	1196075869892	8067403796	497396136821	933626059	2805937178818	2456882061087
32	186.0859	155	1266990932559	15946058377	556060192122	1759741563	3090770443743	3230018817831
64	279.2780	224	1411739396857	32896432606	674743786553	3373185519	3648146974182	5024204345956

Results on Intel Core i7-7700 (soctf-pdc-012)

Thread Count	Total Execution Time	Page-Faults	L1-dcache-loads	L1-dcache-load-misses	Branches	Branch-misses	Instructions	Cycles
1	458.42	204	1130787156348	257684931	443950048158	218537408	2572302112176	1151540123073
2	225.01	262	1131258731240	135944567	444701327365	221903882	2574980922300	1121296575905
4	99.56	453	1131467131415	200748119	444951444468	226669927	2575977821419	1086502468957
8	61.27	601	1132754684474	480534095	447029137022	241645222	2583726994320	1270307270221
16	54.14	705	1135457573197	1059207986	451409926901	291775780	2599966391693	2156430755219
32	49.712	1268	1141734708746	4130483159	461559829012	336574983	2637625400035	3950142554703
64	895.22	17169	1383280475961	30820823396	643734837448	4021444068	3505388693198	4861561984356

Results on Dual-socket Intel Xeon Silver 4114 (soctf-pdc-019)