Module Code: COMP322101

Module Title: Parallel Computation

© UNIVERSITY OF LEEDS

School of Computing

Semester 2 2019/2020

Examination Information

- There are **10** pages to this exam.
- Answer **all 2** questions.
- The total number of marks for this examination paper is **100**.
- This exam is worth approximately **58.82%** of the final module grade.
- The number in brackets **[ ]** indicates the marks available for each question or part question.
- It is expected that it will take approximately **4 hours** to complete this exam.
- Your submission should be less than **5000 words** in total.
- It is possible to receive full marks using less than **3000 words** in total.
- You may lose marks for including extensive discussions that are not relevant to the question.
- You will not receive any marks, and may lose marks, for simply copying material from your notes verbatim.
- You are reminded of the need for clear presentation in your answers.
- You are required to submit a single **PDF file** *via* the Minerva submission point.
- Support for the examination will be available *via* the module Yammer group.

Question 1

This question concerns a parallel implementation of a cellular automoton called "Conway's game of life". The rules for this game are very simple. There is an $N \times N$ array of integers (*i.e.* a two-dimensional array) for which each entry in the array has a value that is either 1 (we say the entry "has life") or 0 ("no life"). The game consists of a (possibly infinite) sequence of iterations from some initial configuration. At each iteration the following rules are applied to determine the value of each array entry after the iteration:

- If the entry has a value of 1 and either two or three (not more and not less) of the surrounding eight entries have a value of 1, then the entry has a value of 1 at the end of the iteration;
- If the entry has a value of 0 and exactly three of the eight surrounding entries have a value of 1, then the entry has a value of 1 at the end of the iteration;
- Otherwise the entry has a value of 0 at the end of the iteration.

The diagram below shows three consecutive iterations for $N = 6$:

$$\begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \qquad \begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array} \qquad \begin{array}{cccccc} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Some sequential code, for which we fix the values of the arrays `u` and `unew` in their first and last rows and columns, is shown below.

```
1     // The code below is repeated for each iteration of Game of Life
2     for (i=1; i<N-1; i++) {
3         for (j=1; j<N-1; j++ ) {
4             neighbours = u[i-1][j-i] + u[i-1][j] + u[i-1][j+1]
5                        + u[i][j-i]               + u[i][j+1]
6                        + u[i+1][j-i] + u[i+1][j] + u[i+1][j+1];
7
8             if (neighbours==3 || neighbours+u[i][j]==3)
9                        unew[i][j] = 1;
10            else
11                       unew[i][j] = 0;
12        }
13    }
14    // Overwrite u with unew before end of iteration
```

(a) Suppose an MPI implementation across p processes seeks to partition the problem into strips, so that each process is responsible for $M = (N-2)/p$ rows of `u` and `unew`. You may assume that $N-2$ is divisible by p, *i.e.* `(N-2)%p==0`.

(i) Explain what is meant by the term "ghost cells" and describe how they are used to facilitate message passing when the problem is partitioned in this way. Your answer should include an explanation of why each process needs to store local arrays of size $(M + 2) \times N$ in order to implement this message passing strategy (also known as "halo exchanges").

[6 marks]

(ii) Assuming that blocking sends and receives are used to complete the ghost cell exchanges at each iteration, explain the order in which the functions `MPI_Send` and `MPI_Recv` would need to be called. You may provide your answer in the form of pseudo-code with the following structure:

```
if (rank==0) {
   // Give pseudo-code here.
}
else if(rank==p-1) {
   // Give pseudo-code here.
}
else {
   // Give pseudo-code here.
}
```

[6 marks]

(iii) At each iteration, what is the total amount of data that must be passed between processes, *i.e.* how many integers in total? State your answer in terms of p and N.

[2 marks]

(b) Now suppose that the code is modified so that 'periodic' boundary conditions are applied. This means that we also update the values in the first and last rows and columns. To do this we assume that the neighbour row above the first row is the last row, the neighbour below the last row is the first row, the neighbour left of the first column is the last column and the neighbour right of the last column is the first column. The following sequential code illustrates this.

```
1  // The code below is repeated for each iteration of Game of Life
2  for (i=0; i<N; i++) {
3      iplus = (i==N-1?0:i+1);
4      iminus = (i==0?N-1:i-1);
5
6      for (j=0; j<N; j++ ) {
7          jplus = (j==N-1?0:j+1);
8          jminus = (j==0?N-1:j-1);
9
10         neighbours = u[iminus][jminus] + u[iminus][j] + u[iminus][jplus]
11                    + u[i][jminus]                     + u[i][jplus]
12                    + u[iplus][jminus]  + u[iplus][j]  + u[iplus][jplus];
13
```

```
        if (neighbours==3 || neighbours+u[i][j]==3)
                unew[i][j] = 1;
        else
                unew[i][j] = 0;
    }
}
// Overwrite u with unew before end of iteration
```

Consider an MPI implementation across p processes so that each process is responsible for $M = N/p$ rows of `u` and `unew`. You may assume that N is now divisible by p, *i.e.* `N%p==0`.

(i) Explain how the structure of the ghost cell exchange differs from part (a), and why the total amount of data transferred is different. Your answer should include a statement of the total amount of data that is sent in this case (in terms of p and N).

[2 marks]

(ii) Explain how deadlock can occur if blocking sends and receives are used without taking sufficient care.

[4 marks]

(c) Now consider the case where N is not an exact multiple of p in part (b).

(i) Discuss how you would modify the partitioning of the rows of `u` and `unew` accross the p processes in this case.

[4 marks]

(ii) What are the implications for load balancing in this situation?

[4 marks]

(d) Now consider an MPI implementation of the periodic boundary problem using a block partition of the grid (as opposed to the strip partition considered above).

- Let the number of processes $p = pi \times pj$ for positive integers pi and pj;
- Assume `N%pi==0` and let $Mi = N/pi$;
- Assume `N%pj==0` and let $Mj = N/pj$.

The block partition is such that the grid is divided into p sub-grids of dimension $Mi \times Mj$ laid out in pi rows and pj columns. This is illustrated below for $pi = pj = 4$.

x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

(i) What is the total amount of data that needs to be transferred between processes at each iteration? Give your answer in terms of pi, pj and N.

[4 marks]

(ii) In the case where $pi = pj = \sqrt{p}$, how does this compare with the data transfer requirements of the previous partitioning strategy, *i.e.* the strip partition (for which $pi = p$ and $pj = 1$)? Illustrate your answer by considering the case $p = 1024$.

[2 marks]

(iii) Comment on the relative merits of the two partitioning strategies taking into account the number of messages that need to be sent at each iteration as well as the total amount of data.

[2 marks]

(e) Suppose next that non-blocking sends and receives are used to overlap the communication with the computation for the two partitioning strategies discussed in this question. Explain why the partition in just one dimension (*i.e.* $pi = p$ and $pj = 1$) is likely to be better the block partition (with $pi = pj = \sqrt{p}$) in an MPI implementation for sufficiently large values of N. Hint: consider the data layout in memory.

[8 marks]

(f) A more efficient sequential implementation of the first version of "game of life" described in part (a) of this question (*i.e.* for which we do not update the values of `u` in its first or last rows or columns) could keep a track of the first and last non-zero entry in each row of the array `u`. In this case can the original code snippet could be replaced by the following.

```
1     // The code below is repeated for each iteration of Game of Life
2     for (i=1; i<N-1; i++) {
3         for (j=low[i]; j<high[i]; j++ ) {
4             neighbours = u[i-1][j-i] + u[i-1][j] + u[i-1][j+1]
5                        + u[i][j-i]                + u[i][j+1]
6                        + u[i+1][j-i] + u[i+1][j] + u[i+1][j+1];
7
8             if (neighbours==3 || neighbours+u[i][j]==3)
9                        unew[i][j] = 1;
10            else
11                       unew[i][j] = 0;
12        }
13    }
14    // Overwrite u with unew before end of iteration
```

In the above, `low[i]` stores the lowest column number in row i that could possibly become "alive" after the iteration, and `high[i]` stores the highest column number in row i that could possibly become "alive" after the iteration. For this question you need not be concerned with how the program keeps track of the values stored in the arrays `low` and `high`.

(i) Comment on the weaknesses of the load balancing strategies considered so far in this question when considering a parallel implementation of the code above.

[2 marks]

(ii) Suggest a better approach to load balancing in this case and briefly describe how it should be implemented.

[2 marks]

(iii) Suggest a way of further improving the efficiency of the serial implementation and an appropriate parallelization (and load balancing) strategy in this case.

[2 marks]

[Question 1 Total: 50 marks]

Question 2

This question concerns a GPGPU implementation of the bubblesort algorithm that is to be implemented in OpenCL. For a first attempt, it is assumed that the size R of the floating point array to be sorted is no larger than T, where T is the maximum work group size that the device supports. In the host code, the array is copied to GPU memory, then a kernel is enqueued, before the sorted array is copied back to the host. The kernel that actually performs the sort is given below.

```
1     __kernel
2     void bubblesort( __global float *array, __local float *scratch )
3     {
4       int gid = get_global_id(0);
5       int workGroupSize = get_local_size(0);
6
7       // Copy to local memory.  Ensure completed before sorting.
8       scratch[gid] = array[gid];
9       barrier(CLK_LOCAL_MEM_FENCE);
10
11      int i, j;
12      for( i=0; i<workGroupSize; i++ ) {
13
14         // Compare-and-swap.
15         if( gid < workGroupSize-1 )
16             if( scratch[gid] > scratch[gid+1] ){
17
18               float temp = scratch[gid];
19               scratch[gid] = scratch[gid+1];
20               scratch[gid+1] = temp;
21             }
22      }
23
24      // Copy back to global memory.  Ensure sorting completed first.
25      barrier(CLK_LOCAL_MEM_FENCE);
26      array[gid] = scratch[gid];
27    }
```

(a) Inspect the code for this kernel, and then answer the following questions.

(i) What does the descriptor `__local` before the argument `scratch` in line 2 denote, and why is this a suitable choice for this kernel? Where is the memory for `scratch` allocated, and how large should the memory pointed to by `scratch` be?

[5 marks]

(ii) Somebody new to GPU programming looks at the kernel code and is confused by line 8. They understand it is copying the array to the memory pointed to by `scratch`, but do not understand why it is not in a loop. Explain why line 8 does indeed copy the full array despite not being in a loop.

[5 marks]

(iii) What function does the `barrier` command on line 9 perform in the context of this kernel?

[2 marks]

(iv) On testing it is found that the sort is sometimes performed correctly, but sometimes fails. Identify the line(s) in the code where the problem lies, and explain why it produces this non-deterministic behaviour. How would you solve this problem? You do not need to provide code or pseudo-code as long as your description is clear.

[4 marks]

(b) The bubblesort described in part (a) is limited to arrays no larger than the maximum work group size T supported by the hardware. Now suppose that you want to extend the algorithm to work for array sizes R greater than this, *i.e.* $R > T$. Someone suggests that you adapt your host code to make multiple calls to a modified kernel.

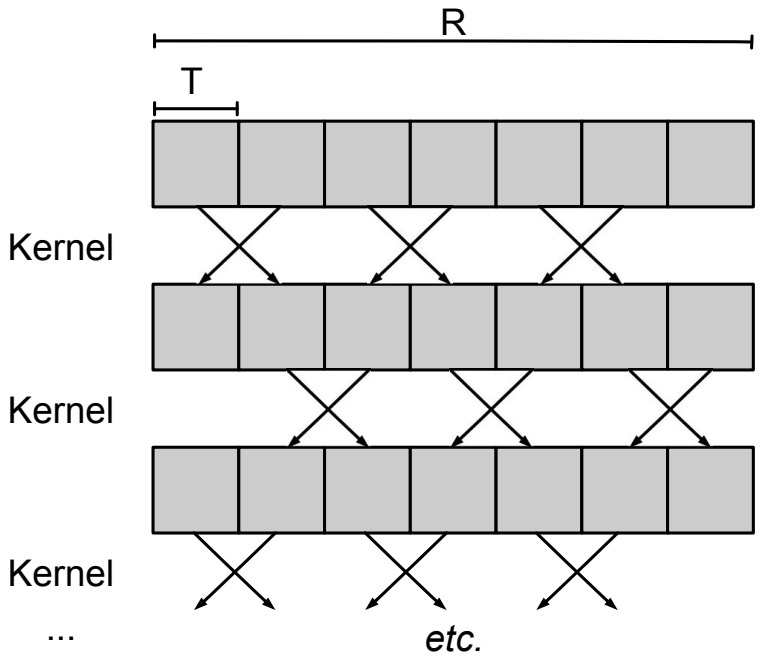
(i) Why do you think it was recommended to make multiple kernel calls?

[3 marks]

(ii) Do you agree it is impossible to perform bubblesort for large problems $R > T$ using a single kernel, and even if it could be achieved, would you recommend doing so? Explain your answer.

[3 marks]

(iii) The host code is now altered to call a modified kernel multiple times to perform the full sort for $R > T$. During each kernel call, sections of the array of size $2T$ are sorted using an algorithm which is an extension of that given in part (a). The index ranges for these small array sections alternate between kernel calls, and by calling the kernel a sufficient number of times, the full list becomes sorted. The first few kernel calls and the alternating index ranges are shown in the diagram below for $R = 7T$, where the grey squares denote array sections of size T, and the crossed arrows denote sorting over $2T$ array elements. How many times must the host code enqueue the kernel to ensure the data is fully sorted? Explain your answer. You may assume R is divisible by T, *i.e.* `R%T==0`. Is your answer different for odd and even values of R/T?



[6 marks]

(iv) With this new implementation, is there a maximum array size R that can be sorted? Explain you answer.

[2 marks]

(c) Returning to the original kernel of part (a), identify three sources of parallel overhead in this code, giving line number(s). For each, explain why the introduction of the overhead was necessary for this specific problem, and whether or not you expect it to be a significant overhead.

[6 marks]

(d) Suppose that you now implement an OpenMP version of bubblesort, and you have verified it correctly works on a multi-core CPU. The same implementation also works in serial, *i.e.* for $p = 1$ core. To determine the parallel scaling of this OpenMP bubblesort, you perform some timing runs.

(i) You measure the time to sort a list of size $R = 1000$ in serial, and find that your implementation takes $t_\mathrm{s} = 1$ ms. You then sort a list of size $R = 4000$ using $p = 4$ cores, and find the execution time has dropped to $t_\mathrm{p} = 0.3$ ms. Finally, you measure the time to sort a list of size $R = 8000$ on $p = 8$ cores, and now find $t_\mathrm{p} = 0.2$ ms. What is the speedup S and efficiency E for both $p = 4$ and $p = 8$?

[4 marks]

(ii) Do the timing runs in part (i) correspond to weak or strong scaling? Explain your answer.

[2 marks]

(iii) It is suggested that a fraction $f = 0.2$ of the serial bubblesort has not been parallelised in the OpenMP implementation. What is the maximum speedup S for this f for both of the cases given in (i) above, according to Amdhal's law, and again for the Gustafson-Barsis law? By comparing these predictions to the actual measurements from (i), what can you say about the hypothesis that $f = 0.2$?

[6 marks]

(iv) Suppose that the time for the GPU version to execute for a problem size $R = 128$ is $t_{\mathrm{p}} = 0.1$ ms using 128 work items or threads. Can you calculate the speedup S for this case?

[2 marks]

[Question 2 Total: 50 marks]

[Grand Total: 100 marks]