SHANGHAITECH UNIVERSITY

2022-2023 SEMESTER 1 MAKEUP FINAL EXAMINATION

CS121 PARALLEL COMPUTING

January 2023 Time Allowed: 100 minutes

Write your answer for each problem on a separate piece of paper, with your name and student ID at the top.

In all problems in which you are asked to design algorithms, you should clearly describe how your algorithm works and argue why your algorithm is correct, in addition to providing code / pseudocode as instructed.

All answers must be written neatly and legibly in English.

1. (a) Suppose you write a parallel program consisting of parallelizable and inherently sequential parts, and find that it achieves a factor 4 speedup using 8 processors. What speedup can you get if you run the same program using 16 processors?

(5 points)

(b) Consider the task graph shown in Figure Q1. Each node represents a subtask, and the time needed to perform the subtask is shown inside the node. An arrow between two subtasks indicates the former subtask must be finished before starting the latter.

(i) Ignoring communication time, what is the maximum speedup achievable when performing this task on a parallel computer with a sufficiently large number of processors?

(5 points)

(ii) Ignoring communication time, what is the minimum number of processors you need to achieve the maximum speedup? Briefly explain your answer.

(5 points)

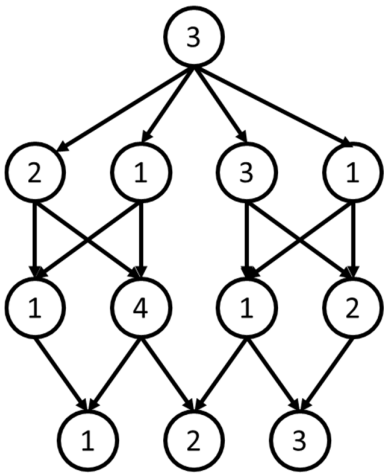

Figure Q1

(c) Suppose you have p processors arranged in a $\sqrt{p} \times \sqrt{p}$ mesh network, and each processor initially holds one value. Describe an algorithm for computing the maximum value in the network. Make your algorithm as efficient as possible. Give the asymptotic time complexity of your algorithm expressed as a function of p, assuming communicating a value or computing the max of two values takes constant time.

(10 points)

2. Consider the problem of sorting n integers which have values in the range 0 to S-1, for some $S \geq 1$. The *bucket sort* algorithm, using $B > 0$ buckets, works as follows. Assume for simplicity that B divides S. Then the i'th bucket, for $0 \leq i \leq B$-1, holds numbers in the range $[i \cdot \frac{S}{B}, (i+1) \cdot \frac{S}{B} - 1]$. Bucket sort first iterates through the n numbers and puts each one in the proper bucket. After this, it goes through each bucket and uses another sorting algorithm, say QuickSort, to sort the numbers in the bucket. Following this, the input values are sorted.

As an example, suppose we use bucket sort with 3 buckets to sort input values [3,1,5,0,1,2,4]. In this case, n=7 and S=6. We first form 3 buckets with values [1,0,1], [3,2] and [5,4]. Then we sort each bucket with Quicksort, to produce the output [0,1,1,2,3,4,5].

(a) Design a *parallel* implementation of bucket sort for a message passing parallel computer using MPI. Assume you have p processors and want to sort n numbers using p buckets, and that p divides n. Also assume all processors know S, the maximum possible input value. Initially each processor holds n/p numbers. At the end of the execution, each processor should have an array of numbers, such that if we read the arrays at processors 0, 1, ..., p-1 in order, the n input values are listed in sorted order.

You may use MPI pseudocode as we have done in class, as long as it is clear how your code works. You can also assume each processor can call a sequential QuickSort function. Different implementations are possible; make yours as efficient as possible.

(15 points)

(b) Analyze the speedup your algorithm achieves. Does the speedup depend on the values in the input? If so, describe a worst case input (achieving the minimum speedup), and a best case input (achieving the maximum speedup).

Note that if your algorithm achieves the same speedup on all inputs, you do not need to describe a best and worst case input. Also note that you can assume Quicksort sorts m numbers in $O(m \log m)$ time.

(10 points)

3. Recall that in *breadth first search (BFS)*, we visit the nodes of a graph in order of their distance from a root node, in terms of the number of edges; nodes at the same distance can be visited in any order. After BFS, each node should be labeled with its distance from the root.

Pseudocode for a sequential implementation of BFS is given in Figure Q3. It uses a queue `Q`, initialized to the root node, to keep track of the order in which nodes are visited. The graph `G` is given in adjacency list format, so that for each node `v`, `Adj(v)` is a list of the vertices adjacent to `v`. `distance` is an array giving the distance of each node from the root. As long as `Q` is not empty, the algorithm removes the first node `v` in `Q`, and for each unvisited neighbor `u` of `v`, sets its distance to be one more than `v`'s distance, and adds `u` to `Q`.

```
BFS(G,root)
for all nodes v in G
     distance[v] = -1
enqueue(Q,root)
distance[root] = 0
while Q ≠ ∅ {
     v = dequeue(Q)
     for each u∈Adj(v)
          if (distance[u] == -1) {
               distance[u] = distance[v] + 1
               enqueue(Q,u) }
}
```

Figure Q3

(a) Design a *parallel* implementation of BFS for a shared memory parallel computer using OpenMP. Make your algorithm as efficient as possible. Write your code in OpenMP or OpenMP-style pseudocode; in the latter case, clearly indicate your uses of parallelism, synchronization, etc. Also clearly describe how your algorithm works.

Note that your algorithm does not have to follow the code structure in Figure Q3, which is provided only for reference. If necessary, you can assume the availability of a shared queue data structure, accessible by all the threads. However, the queue is not guaranteed to be thread-safe.

(15 points)

(b) Discuss how much performance improvement your parallel algorithm can achieve, including any performance bottlenecks. On what types of graphs can the algorithm achieve a large improvement? On what types of graphs will it achieve a small improvement?

(10 points)

4. In this problem, suppose you are given an $n \times m$ matrix M (with n rows and m columns), stored in *row-major* format.

(a) Write a CUDA program to compute the *column sums* of M. That is, your program should output a size m array, where the i'th value of the array is the sum of the values in the i'th column of M. You only need to give the GPU kernel code, not the CPU host code. Assume that m is sufficiently large, say $m > 50000$.

(8 points)

(b) Next, we want to compute the *row sums* of M. That is, we want to output a size n array, where the i'th array value is the sum of the values in the i'th row of M. This can be done using the kernel function `rowSums` shown in Figure Q4. `sums` is the output array.

Unfortunately, `rowSums` is likely to be slow on a GPU. Explain the reason(s) for this by describing how `rowSums` works, and any inefficiencies the function has. Assume that $1000 \leq m, n \leq 10000$.

(7 points)

```
__global__ void rowSums(float *M, float *sums, int n, int m) {
    int row = threadIdx.x + blockIdx.x * blockDim.x;
    if (row < n) {
        float s = 0;
        for (int i = 0; i < m; i++)
            s += M[row * m + i];
        sums[row] = s;
    }
}
```

Figure Q4

(c) Write an improved row sums kernel that runs faster than the one in Figure Q4. Briefly explain how your kernel works and why it is faster. As before, you only need to give the GPU kernel code.

(10 points)

END OF PAPER