Started on	Wednesday, 1 May 2024, 6:37 PM
State	Finished
Completed on	Thursday, 2 May 2024, 2:23 PM
Time taken	19 hours 46 mins
Grade	Not yet graded

Question 1
Correct
Mark 1.00 out of 1.00

OpenGL shaders are written in GLSL, which is a C-like language that enables programmers to control the 3D rendering process. What kind of CPU is most suited for this task of executing the *same* GLSL instruction on *each* vertex of a 3D graphic to form a pixel onto a 2D image?

- a. MIMD
- b. SISD
- [✓ c. SIMD]
- d. MISD

Correct
Marks for this submission: 1.00/1.00.

Question 2
Correct
Mark 1.00 out of 1.00

There are two subprograms (*threads*) spawned by a program (a *process*). Subprogram A maintains a dispatch queue where a 64-bit integer is pushed. If the queue has something during idle time, it pops one off of it and writes it into an 8-byte memory cell only if there are no other subprograms currently looking or writing at it. Subprogram B, on the other hand, checks the same memory cell whether there is a value and A is not writing to it every 250 ms. Once B sees that A is not using the memory cell, it will interpret its contents as a 64-bit integer and will process the data accordingly.

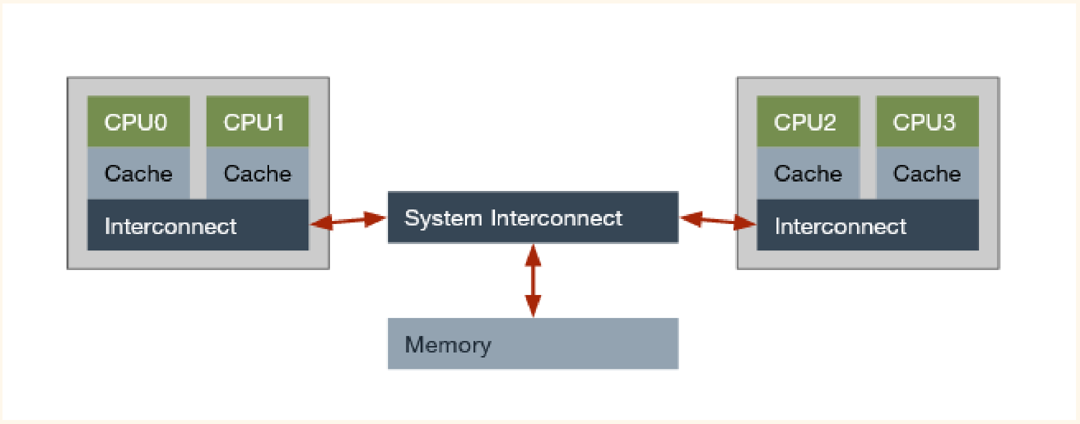
What kind of process interaction is this?

- a. Message Passing
- [✓ b. Shared Space]

Correct
Marks for this submission: 1.00/1.00.

Information

Consider the oversimplified system hardware architecture shown below for a quad-core CPU with two dies.



Question 3

Correct

Mark 1.00 out of 1.00

Consider a thread at CPU0 wanting to access a variable x, which apparently, does not exist in any of the cache lines of itself or in any of the other CPUs! With the request reaching the system interconnect, and the interconnect being made aware that x was not cached, where will it check for x next?

- a. Nowhere, because the operation will fail
- [✓ b. Memory]
- c. Die 1 (containing CPUs 2 and 3)
- d. Die 0 (containing CPUs 0 and 1)

Correct

Marks for this submission: 1.00/1.00.

Question 4

Correct

Mark 1.00 out of 1.00

Consider another thread at CPU0 wanting to change a variable \(y \). CPU0 contains a cache line to \(y \), so it can just update it. However, CPU2 *also* contains a *read-only* copy of y. Do we need to flush the cache line there?

- [✓ a. Yes]
- b. No

Correct

Marks for this submission: 1.00/1.00.

Question 5

Correct

Mark 1.00 out of 1.00

Consider two threads that share a single variable z. Will it be faster in general if the two threads are forced to run in a single CPU than having them on separate ones assuming that the threads do not have any more variables being shared across other CPUs?

- [✓ a. Yes]
- b. No

Correct

Marks for this submission: 1.00/1.00.

Information

Problem Statement

Consider implementing a breadth-first search algorithm on a graph represented as an adjacency list. A function `bfs()` accepts an adjacency list G and a source vertex s and will return a breadth-first search tree representing the traversal paths from s to any other node in G.

Question **6**

Correct

Mark 0.50 out of 0.50

Consider this "standard" implementation of the bfs() algorithm.

```
def bfs(G, s):
    bfs_q = [s]
    bfs_tree = [-1 for _ in range(len(G))]
    bfs_tree[s] = s

    while len(bfs_q) > 0:
        tp_stk = bfs_q.pop(0)

        for each_nbr in G[tp_stk]:
            if bfs_tree[each_nbr] >= 0:
                continue

            bfs_tree[each_nbr] = tp_stk
            bfs_q.append(each_nbr)

    return bfs_tree
```

What is the time complexity of this implementation for $|E|$ number of edges and $|V|$ number of nodes?

- a. $O(|V|^2)$
- [✓ b. $O(|V| + |E|)$]
- c. $O(|E|^2)$
- d. $O(|V||E|)$

Correct

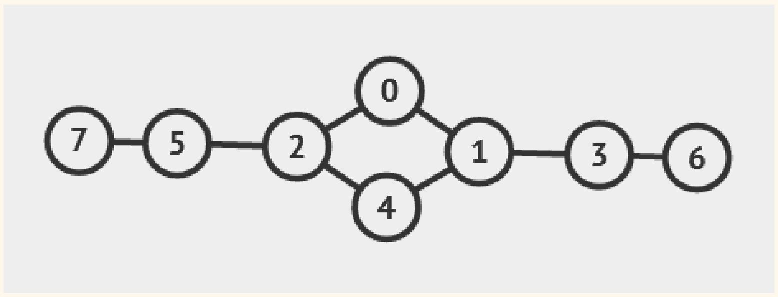
Marks for this submission: 0.50/0.50.

Question 7

Correct

Mark 0.50 out of 0.50

Consider the following graph below:



Its adjacency list representation is shown below for reference.

```
G = [
    [1, 2],
    [0, 3, 4],
    [0, 4, 5],
    [1, 6],
    [1, 2],
    [2, 7],
    [3],
    [5],
]
```

What is the length of the longest distance between node 0 and any other node? Note that the answer is a natural number, and node 0 is at distance 0 from itself.

Answer: 3

Correct

Marks for this submission: 0.50/0.50.

Question 8
Complete
Marked out of 2.00

One of the ways to be able to parallelize algorithms is to make them *iterable*. This means operating on them in terms of arrays or lists of data. Each element is then applied the exact same operation, which is embarrassingly parallel. In Python, this kind of "functional" programming is done using list comprehensions and their `map()`, `filter()`, and `reduce()` high-level functions.

This previous sequential implementation works for most purposes. However, we may be able to reform our `bfs()` such that it runs in a parallel machine. How can we reform the `bfs()` algorithm such that each operation is truly iterable? We are specifically looking for the following:

- Algorithm correctness
- Explanation thoroughness
- Handling of edge cases, if any

If possible, you can upload an optional pseudocode or program code in a language of your choice in .txt format to defend your case.

PS. Think of the places in the sequential `bfs()` algorithm above where you can substitute iterables. Additionally, a queue will *not* be needed but a *list* will still be.

To reform the **breadth-first search (BFS)** algorithm such that it can run efficiently on a parallel machine, we need to ensure that the algorithm utilizes a **shared-memory model** suitable for **multi-threaded environments**.

For BFS to run efficiently in a parallel environment we must consider the following:

1. **Initialization** – have a shared memory space where each thread can access and update the state of nodes
2. **Parallel Execution** – divide the nodes of the graph into batches that can be processed in parallel; each thread processes its batch of nodes
3. **Synchronization** – after processing a batch of nodes, threads need to synchronize to update the global state and prepare for the next batch of nodes to be processed

The Python code is an implementation of a parallel BFS algorithm using a *multiprocessing* library. We assume that the graph is large enough that it is suitable to be divided among several processors. Note that for very large graphs or when running on systems with many cores, the parallel version could provide significant performance improvements.

It can be observed that *Manager* was utilized from the multiprocessing module to manage shared data (*bfs_tree* and *next_layer*). This is important since in a multiprocessing environment, each process has its own memory space, and this method allows different processes to access and modify shared data.

Consequently, the code also handles a *pool of processes* to parallelize the BFS traversal. A *pool* of workers is created within each iteration and each segment of the BFS task (*process_layer*) is mapped across the pool. This allows each node in the current layer to find and list unvisited nodes. After mapping to the process pool, *close()* and *join()* methods ensure that all processes from the current BFS layer are finished before moving on to the next layer.

parallel_bfs.txt

Question 9

Complete

Marked out of 1.50

Given the algorithm that you have formulated above, enumerate the parts where there may be concurrent writes - that is, there may be parts where two or more "tasks" (elements in this case of an iterable algorithm) may have to write to the same variable or data. Explain whether synchronization or a thread safety mechanism is needed for the algorithm to be truly parallel.

In the reformed parallel BFS algorithm implemented using the multiprocessing module, the following elements or parts of the algorithm where concurrent writes can occur:

1. Writing to *bfs_tree* – since two processes may discover two different nodes simultaneously that share a common unvisited neighbor. If not properly synchronized, the two processes might try to write to the same index of the *bfs_tree* simultaneously.

   a. Synchronization – this method is inherently handled by Manger.list() which ensures that operations like assignments are managed **atomically**. Thus, when a process assigns a parent to a node (*bfs_tree[neighbor]=node*), this operation is thread-safe.

2. Appending to *next_layer* – processes add nodes to the *next_layer* managed list concurrently when they find new nodes to visit in the next level of BFS. This method of adding nodes could lead to data corruption if multiple processes try to append to the list at the exact same time without proper synchronization.

   a. Synchronization – since *next_layer* is a *Manager.list()*, the *extend* operation utilized to add new nodes is thread-safe. The manager's list handles synchronization internally, ensuring that concurrent *extend* operations do not interfere with each other.

Since overhead could lead to bottlenecks from the synchronization, we could consider pre-allocating space for the *bfs_tree*. Since each node will be visited exactly once, each process could work on a separate segment of the graph. Moreover, explicit locks, semaphores, or atomic operations could also be considered when managing shared data access manually in the case that *Manger.list()* is not available.

Question 10

Correct

Mark 0.50 out of 0.50

What is the runtime of your iterable `bfs()` algorithm for $|E|$ number of edges and $|V|$ number of nodes?

○ a. $O(\frac{|E|}{|V|})$

○ b. $O(|V||E|)$

[✓ c. $O(|V|+|E|)$]

○ d. $O(1)$

Correct

Marks for this submission: 0.50/0.50.

◄ [Slide] W10B: Parallel Programming and Hardware

Jump to...

SIMD Programming (Slides from UCSB) ►