**VEDANSI PARSANA**

**TASK 3A)**

🡪**What are the basic steps (show all steps) in building a parallel program? Show at least one example.**

There are three basic steps in building a parallel program. First, identify which groups of tasks or sections of data are “independent” and hence can be implemented simultaneously, and divide such data into equivalent partitions. Second, delegate these partitions effectively to the available processors using load balancing techniques (either static or dynamic) to most efficiently process the workload. Third, manage the data access, scheduling, communication, and synchronization of tasks during the execution of the program to ensure it is executed correctly. The most obvious example of when and how to utilize parallel programming is when we want to perform a single process on a large array - an array which can be divided into subarrays to be processed identically and independently from one another. On the execution of a parallel program, a “master” thread would identify the number of available processors and divide the array equally in accordance with the number of available processors, and then distribute the subarrays to those processors. Each processor then can independently receive a subarray, perform the necessary computations on it, and return the results to the master thread which would be expecting it.

A picture containing drawing

Description automatically generated

**🡪 What is MapReduce?**

MapReduce is a programming model developed to process large datasets via parallel, distributed computing utilizing multiple machines. The model builds the map and reduce combining concepts found in some functional languages - where the map takes a function and a sequence of values as input before applying the function to each of the given values and the reduce combinators combines each of the resulting values into one final value by way of a binary operation.

**🡪 What is map and what is reduce?**

Map is a process of producing intermediate key/value pairs from the input key/value pair and then, via the MapReduce library, sending the set of values associated with a given key to the reduce function to determine the final output. The reduce function takes a single key and the set of intermediate values associated with that key and combines with the values to form a smaller set of values which is then returned as the final output.

**🡪 Why MapReduce?**

MapReduce is perfect for processing Big Data because it is highly accessible, as you can process more data by simply adding additional computing nodes. MapReduce also allows the user to focus on the problem at hand without having to be concerned with the harder technicalities of the system (like data distribution or fault tolerance).

**Show an example for MapReduce.**

An example for MapReduce would be determining a term’s frequency throughout many unique documents. The map returns an intermediate <word, frequency> pair for each of the provided documents, and then the reduce function which would add those intermediate results for the given word to return a final total frequency for that term.

Example:

Map (String key, String value):

// key: document name

// value: document contents

for each word w in value:

EmitIntermediate(w, "1");

Reduce (String key, Iterator values):

// key: a word

// values: a list of counts

int result = 0;

for each v in values: result += ParseInt(v);

Emit (AsString(result));

**🡪 Explain in your own words how MapReduce model is executed.**

When the MapReduce function is called, the MapReduce library splits the input dataset into equivocal “shards” to be processed concurrently over multiple machines. Each machine starts a copy of the same program. One copy is the “master” and others are “worker” programs. The master assigns map or reduce functions to each available worker for parallel processing. If a worker is assigned a map task, it analyses its allocated shard for the intermediate key/value pairs. When found, the intermediate pairs are, by way of a buffer, regularly written to the local disk. The master is notified of the completion of the task and in turn, informs an idle worker of the intermediate pairs’ location on the disk. The worker assigned to a reduce task sorts through the intermediate data before finally emitting the final output. After the entire dataset has been processed, the master exits the MapReduce.

**🡪 List and describe three examples that are expressed as MapReduce computations.**

**Distributed Grep:** The map function would take a string as input and return a list of lines which matched the pattern of that string in the form of intermediate pairs. The reduce function would render no change on the intermediate pairs though, simply returning the supplied input as the final output.

**Count of URL Access Frequency:** The map function would take logs of URL access requests as input and return intermediate pairs which reads “the given URL was accessed once.” The reduce function would then add those intermediate results for the given URL to return a final total frequency of access requests for that URL in the form of a <URL, total count> pair.

**Inverted Index:** The map function would take as input the text on a set of documents (for example, the individual pages of a whole book), and for each of the words on a page, it would return a list of <word, page number> pairs. Thus, page one read “once upon a time” and page two read “in a land far away,” then the pairs for page one would be: <once,1>, <upon,1>, <a,1>, <time,1> and page two would follow likewise. The reduce function will then identify which page numbers corresponded with a given word and sorts the page numbers to emit a final <word, list(page number)> pair. So, the pages mentioned earlier would emit “once - 1” and “a - 1, 2” to name two examples.

**🡪 When do we use OpenMP, MPI and, MapReduce (Hadoop), and why?**

**OpenMP:** OpenMP is used for parallel programming via shared-memory, as generated threads work from a shared memory and data to allow multiple processors to perform chunks of the program. OpenMP programs allow us to differentiate which sections of the program should be done in parallel and which should be done in sequential.

**MPI:** Message Passing Interface is a portable library of routines used for developing parallel code over multiple machines. MPI can be used in combining with OpenMP to introduce threading into the program. It is thorough, intensive computation such as in scientific computing.

**Hadoop MapReduce:** MapReduce is used for distributed parallel programing with the goal of performing relatively simple parallel operations on extremely large datasets. As the dataset is distributed in smaller blocks of data among multiple machines, MapReduce has a higher fault tolerance than similar systems such as MPI.

**🡪 In your own words, explain what a Drug Design and DNA problem is in no more than 150 words.**

DNA stores all the information that a cell needs to create proteins which, depends upon a protein’s unique shape and performs different functions within the body. Some of those functions are not beneficial and can actually lead to diseases. Drug Design is the process of finding and testing ligands, small molecules which are capable of binding to proteins, to determine if they (when familiarized to proteins related with a specific disease) will bind to the problematic proteins in a way which eases that disease. This process starts with representing a three-dimensional model of the protein in question and then testing ligands against that represents to see whether they will even bind to the protein or not. If a ligand does bind to the protein, then it is given a score rating according to how strongly it binds and whether the binding produces a positive change. The best scoring ligands are then produced (as the drugs) for further testing.

**Task 3B) Parallel Programming Basics:**

In the part I learned about the design of drugs using molecular model computation. The programs match proteins with ligands to find the best pair. The codes apply map-reduce, in which the ligands are mapped using their binding score and reduce pulls together the sequences to find the highest scoring ligand pair.

The first code uses sequential sorting giving us an idea on how much of a difference there is between it and parallel. It was set up easily as once the directory was created the files were just copied in and we ran Makefile to create an executable. Sequential by far to the longest to run with over 2 minutes wait period. By comparing the pairs one by one we can see that sequential doesn’t use most of the Raspberry’s capabilities.

A screenshot of a cell phone screen with text

Description automatically generated

The second code I used for searching to the top ligands was OpenMP. It took a lot longer to implement this one as it used TBB (Thread Building Blocks) which isn’t part of the basic libraries. It took a lot’s of time to figure out the error popped up related to TBB but than I found it and installed the TBB libraries after it was installed rest everything went smoothly.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Once the TBB was done OpenMP ran quickly taking .02 seconds to do just what sequential did.

The C++11 solution was as simple as sequential to implement because TBB was set up and after that there were no problems. It ran .02 which is little faster than OpenMP but it’s hard to relate with such close numbers.

A screenshot of a cell phone

Description automatically generated

Run Times:

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

|  |  |
| --- | --- |
| Implementation | Time(s) |
| dd\_serial | 146.45 |
| dd\_omp | 0.02 |
| dd\_threads | 0.02 |

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| Implementation | Time (s) 2 Threads | Time (s) 3 Threads | Time (s) 4 Threads |
| dd\_omp | .02 | .06 | .20 |
| dd\_threads | .02 | .04 | .17 |

Questions:

1. What approach is the Fastest?

• The fastest running code is C++11, with the time at 4 threads beating OpenMP and sequential which is not even close ever.

2. Determine the number of lines in each file. How does the C++11 implementation compare to the OpenMP implementation?

• The C++11 code has less lines of code, with 137 to the OpenMP’s 152 lines.

A screen shot of a smart phone

Description automatically generated

3. Increase the number of threads to 5 threads, what is the run time for each?

A screenshot of a cell phone

Description automatically generated

|  |  |
| --- | --- |
| Implemetation | Time(s) 5 threads |
| dd\_omp | 2.07 |
| dd\_threads | 0.79 |

The time of both increased by around 4 times approximately.

4. Increase the maximum ligand length to 7 and rerun each program. What is the run time for each?

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

|  |  |
| --- | --- |
| Implemetation | Time(s) |
| dd\_serial | 150.16 |
| dd\_omp | 0.02 |
| dd\_threads | 0.02 |