**Anirudha Dudhasagare\_HW1\_002697516**

**Part 2: Essay Questions:    50 points**

Answer the following questions in a Word document docx:

1. Explain the differences between serial and parallel code.      [4 points]

In serial computing, A problem is divided in discrete set of instructions and these instructions are executed in order one after another on a single processor. Only one instruction is executed at a time

While in parallel computing a problem is divided in discrete parts which can be solved concurrently. These parts are then broken down into series of instructions which can be executed simultaneously on different processors. The main purpose of parallel code is to compute a solution in less time compared to serial code

2. What is the FLOPS and its usage in parallel computing?        [2 points]

FLOPS is an abbreviation for Floating-Point Operations per Second. It is used in ratio to peak performance to find overall efficiency of parallel code.

It is a measure of computer performance, which relies on floating point calculations. FLOPS is a accurate measure of performance than simply measuring instructions per second.

3. Simply explain the computer architecture of von Neumann.  [4 points]

Von Neumann Architecture is fundamental model for mist modern computers. It’s also known as stored-program computer where both program instruction and data are kept in the electronic memory.

It’s made up of following components:

1. CPU – mainly for fetching, executing instructions and performing calculations, which houses Control unit and Arithmetic Logic Unit (ALU).
   1. Control Unit: CU decodes instructions received from CPU and sends it to ALU.
   2. Arithmetic Logic Unit: Also known as ALU, its primary tasks are to perform calculations such as add, subtract and compare as per the decoded instructions received from CU.
2. Memory – The memory is where the data and instruction sets are stored and from where the CPU fetches, performs operations and stores result back on or sent back to I/O devices.
3. I/O devices – I/O port/device allows for new input and output to be delivered. devices such as keyboard, mouse and monitor transmit data which allows users to use computer

4. Simply explain: what is shared memory architecture and what is distributed memory architecture?    [4 points]

**In Shared Memory,** all the processors share a common global memory space, hence allowing all processors to directly access and perform read/write operations on same memory via an interconnected bus network.

Shared memories are classified into two types based on their memory access times.

1. UMA (Uniform Memory Access): it is represented by symmetric Multiprocessor Machines (SMPs) where every processor is identical. UMAs have equal access time to memory since all processors share same memory
2. NUMA (Non-Uniform Memory Access): It is usually made by physically connecting multiple SMPs. where these SMPs can directly access each other’s memory. Not all processors have equal access time to all memories. It can access its own local memory uniformly but accessing memory block of another processor group would take longer.

Key advantage is a global memory space which provided user friendly programming and uniform data sharing between tasks. Main disadvantage of this architecture is lack of scalability between memory and CPUs and responsible implementation for synchronization constructs that ensure correct access to global memory.

**In Distributed Memory Architecture,** there is a requirement of efficient communication network. Each processor has its own local memory. These Processors + Memory combo communicates by exchanging messages over network, typically using Message Passing Interface (MPI).

Key advantages of distributed architecture: Memory is scalable with the number of processors and access its own memory without the overhead resulting in scalable, performant and cost-effective system.

Disadvantages of this architecture are requirement of highly skilled programmer. Its hard for serial code to comply with this architecture and memory access times are non-uniform.

5. Please give the parallel programming models in common use and simply explain.     [4 points]

Shared Memory Model (without threads): In this model, processes or tasks share a common address space to read and write asynchronously. Various mechanism are used such as locks and semaphores to control access to shared memory, resolve conflicts, prevent race conditions and deadlocks.

Thread Model: It’s a type of shared programming model where a single heavy weight process can have multiple light weight execution path concurrently. There two different implementation of thread model -> POSIX Thread (also known as PThreads) and OpenMP. POSIX is a library of subroutines whereas OpenMP is a set of compiler directives embedded in the serial/parallel code. It utilizes a Fork-Join Execution Model.

Distributed Memory Model/ Message Passing Model: in this model, a set of tasks use their own local memory for computation. These tasks exchange data through communication over a network by sending and receiving messages. The Data Transfer requires cooperative operations to be performed by each process. Its programmer’s responsibility for define parallelism and maintain synchronization across all nodes over the network. The Message passing implementation usually compromises of a library of subroutines (MPI).

Hybrid Model: This model is a combination of other shared memory architecture such as Thread model or Distributed Memory Model where OpenMP is usually used for parallelism within its own node and MPI to communicate between process on other nodes.

6. According to the lectures, how to evaluate your parallel performance practically? List the calculating formulas and explain.       [5 points]

**SpeedUp**: it is a common way to assess the efficiency of a parallel code. Speed up is defined as the ration of wall-clock time for a serial code to the wall clock time for a parallel code.

Formula ->

S = wall clock time spent in serial execution/wall clock time spent in parallel execution

**Parallel Efficiency**: measures how efficiently a parallel code utilizes multiple processors. Usually the speedup cant be greater than the number of parallel resources on which the program is running.

Formula ->

Sp = Speedup/ Number of Cores

**Amdahl’s Law**: It is used to predict theoretical speedup in a latency of the execution of a task at a fixed workload when using multiple processors. It shows that code’s serial part limits the potential speedup from parallelizing code no matter how many processors you add up.

Formula->

Where, T1 = the execution time on single core

TN = runtime of same program on n cores

Nis the number of cores

S = Serial Runtime/Parallel Runtime

= T1 / TN

=

= (for simplification )

**Gustafson’s Law**: this law completely flips Amdahl’s script. Instead of focusing on a fixed workload, Gustafson’s states that parallel portion of program increases as the number of core increases. Instead of reducing the execution time, we increase the amount of computation to be done within given time interval

Formula->

Scaled speedup = serial runtime/parallel runtime

S =

= (

= N + (1 - N).

= 1 + (N – 1).

7. Briefly explain the hybrid parallel programming model on current supercomputers (HPC clusters).      [4 points]

The Hybrid Parallel Programming model is usually used on current supercomputer clusters. These clusters are standalone computer with its own CPU, memory and OS called as Nodes. These nodes utilize combination of two types of parallelism model.

1. Shared Memory Model (Thread Model) -> within a single node, multiple CPUs share a common local memory. OpenMP is often implemented to allow processes/tasks to run on multiple CPUs. These Nodes can also be connected to accelerators such as GPU. NVIDIA’s CUDA framework is usually used to access GPU’s cores and achieve parallelism.
2. Distributed Memory Model: This model utilizes Message Passing Interface (MPI) for nodes to communicate with each other and exchange data over a network. The MPI is a library of subroutines which are called withing the parallel source code to achieve parallelism.

8. Explain the elements of Flynn's taxonomy.   [4 points]

Parallelization can be achieved by using multiple instructions or multiple data streams. Flynn’s Taxonomy classifies different type of parallel computing architecture based on their method of achieving parallelization. It is a two by two table where rows represent the type of instruction and columns represent the type of data, which results in four main types of classification.

Single Instruction Single Data (SISD): it is a single CPU thread executing one operation at a time.

Single Instruction Multiple Data (SIMD): it executes single stream of instructions concurrently across multiple data by a vector processor. It is used by most modern computers usually with GPUs.

Multiple Instructions Single Data (MISD): It is rare type of architecture. Used to filter data to detect errors.

Multiple Instructions Multiple Data (MIMD): usually a cluster of nodes which can operate separately but are connected by high speed network which enables to effectively communicate with each other.

9. Explain the terms of nodes and cluster.      [4 points]

**Node**: A node is a standalone physical computer with its own Operating System and network connection. Multiple nodes are connected together over a network to form a cluster or a supercomputer cluster. A node consists of its CPU/cores, GPU, RAM, Power management, local storage and controllers in a socket which can be installed to in a Rack Cabinet to form this cluster. It’s a standalone von Neumann Computer usually referred as blade.

**Cluster**: Cluster is a collection of computers called Node/Blade that are interconnected and function as a single unit. They help provide uniform environment for tasks running on cluster and work together to provide fault tolerant access to a file storage. These clusters are based on MIMD architecture as mentioned in the Flynn’s Taxonomy.

10. Please list the three primary API components in OpenMP and give simple examples.      [4 points]

Three primary APIs of OpenMP are:

1. Compiler Directives: These are annotation we include in the source code that helps compiler to result in parallel code.

Eg./syntax (In one of our assignments we used) ->

#pragma omp parallel private(nthreads, tid)

{ code block }

1. Run-time Library Routines: These routines are used for setting and querying number of threads to be used, fetching Thread IDs, team size.

Syntax: #include<omp.h>

int omp\_get\_num\_threads(void)

1. Environment Variables: These are can be used to control runtime behavior like setting the number of threads, deciding on how loop iterations are divided and binding threads to processors.

Syntax: export OMP\_NUM\_THREADS=8(Discovery)

11. Please write the steps to use gcc compiler to compile a C file “hello\_omp.c” with OpenMP flag and get the executable file “hello\_omp”. On Linux bash shell system, set 4 threads by using the environment variable; then run this executable. [3 points]

       Note: don’t run it, only give the commands.

1. COMPILE -> gcc -o hello\_omp -fopenmp hello\_omp.c
2. SET ENVIRONMENT VARIABLE -> export OMP\_NUM\_THREADS=4
3. RUN THE EXECUTABLE -> ./hello\_omp

12. In HPC, there are two main ways of scaling a program. Please explain them.     [4 points]

Two Main ways of scaling are Strong Scaling and Weak Scaling.

Strong Scaling: The problem size is constant and number of tasks to solve the problem increases. Basically, you split the size of workload over additional number of cores to execute it faster and reduce the execution time. It is associated with Amdahl’s Law where the workload/problem size remains constant, you just add more compute power by adding more cores/tasks. Its performance is affected by the ratio of time spent communicating to time spent communicating. The larger the ratio, the worse the strong scaling behavior will be. The reason adding more task does not result in significant reduction in execution time is because of parallel overhead, the more you add tasks, the more time it has to spend in establishing communication between tasks and worse the ratio gets.

Weak Scaling: The Problem size increases proportionally with the number of tasks, keeping the work done per task constant. We focus on solving bigger workload trying to keep the time-to-result constant. So here the execution time basically remains same for smaller as well larger problem size, given that we increase the number of tasks. Usually Associated with Gustafson’s Law, A program’s serial part limits the potential of speedup from parallelizing the code. As we add more processors, the parallel part gets bigger and serial part stays same and becomes less of deal enabling us to do more in the same time. Well written parallel programs have good weak scaling performance. The major factor affecting its performance is the synchronization overhead.

13. When you run your serial code on multiple cores (CPUs), you can employ two different approaches. Please explain these two approaches [2 points]. Two theoretical results (laws) describe the conditions for the success of each approach. Please explain the two corresponding laws [2 points].

The two approaches when running serial code on multiple cores is strong scaling and weak scaling.

Strong Scaling employs the idea of splitting original workload across multiple available cores or adding more cores to it in order to achieve faster execution time.

Weak Scalling focuses on using extra cores/resources and proportionally increasing the workload/computation while keeping the time to result constant. Here the focus is to execute the scaled-up version of the original problem in constant time.

Strong scaling is associated with Amdahl’s Law is used to predict potential speed up when using multiple processors until we reach a point of diminishing results.

In reality, the theoretical speedup is not achieved due to Parallel overhead. The more cores you add up, the more time it spends in establishing with each other. The Strong scaling performance is affected by the ratio of time spent communicating to ratio of time spent computing. The larger the ratio the worse the performance.

Weak Scaling is associated with Gustafson’s Law which states that the serial part of the code is what limits the true potential of speedup, the time to execution in serial part will always be same no matter how many cores you add, but the parallel part time can reduce the time, so you focus on solving larger problems in same amount of time it takes for smaller problems as you proportionally add cores. It is used when you have plans to run larger problem. The performance is affected by the synchronization overhead – time spent synchronization as you add more cores.