**What is the FLOPS? and usage in parallel computing?**

In computing, floating-point operations per second (FLOPS) is a measure of computer performance, useful in fields of scientific computations that require floating-point calculations. For such cases, it is a more accurate measure than measuring instructions per second.

Parallel computing is the use of multiple processors to work on a single problem simultaneously. This can significantly improve the performance of applications that can be broken down into smaller tasks that can be executed independently.

FLOPS is a useful metric for measuring the performance of parallel computing systems because it can be used to estimate how quickly a system can complete a given task. For example, if a parallel computing system has a FLOPS rating of 100 teraFLOPS (TFLOPS), it means that the system can perform 100 trillion floating-point operations per second.

Here are some examples of how FLOPS is used in parallel computing:

Climate modeling: Climate models are complex mathematical models that simulate the Earth's climate. These models require a large number of floating-point operations to be performed, so parallel computing is essential for running them efficiently.

Drug discovery: Drug discovery is a complex process that involves testing many different molecules to see if they have the desired therapeutic effect. This process can be accelerated by using parallel computing to test multiple molecules simultaneously.

Financial modeling: Financial models are used to predict the future performance of financial assets. These models can be very complex and require a large number of floating-point operations to be performed. Parallel computing can be used to speed up the execution of these models and make them more useful for investors.

Overall, FLOPS is a valuable metric for measuring the performance of parallel computing systems. By understanding how FLOPS is used, programmers can design more efficient parallel programs and achieve better performance.

**Simply explain the computer architecture of von Neumann.**

The von Neumann architecture is a type of computer architecture that stores both program instructions and data in electronic memory. This is in contrast to earlier computers, which were programmed by hardwiring the instructions into the computer circuitry.

The von Neumann architecture consists of four main components:

Central processing unit (CPU): The CPU is responsible for executing the program instructions.

Memory unit: The memory unit stores both program instructions and data.

Input/output (I/O) unit: The I/O unit allows the computer to communicate with the outside world.

Bus: The bus is a communication pathway that connects the CPU, memory unit, and I/O unit.

The von Neumann architecture works by fetching instructions from memory, executing them, and then storing the results back in memory. This process is repeated until the program terminates.

Here is a simplified explanation of how the von Neumann architecture works:

The CPU fetches the next instruction from memory.

The CPU decodes the instruction and determines what operation to perform.

The CPU performs the operation and stores the result in a register.

The CPU fetches the next instruction from memory and repeats steps 2-4 until the program terminates.

The von Neumann architecture is a simple and elegant design, and it has been the basis for nearly all modern computers.

The von Neumann architecture is a powerful and flexible design that has allowed computers to become increasingly sophisticated and powerful over the years.

A diagram of a computer component

Description automatically generated

**Simply explain: what is shared memory architecture and what is distributed memory architecture?**

Shared memory architecture:

In a shared memory architecture, all processors have direct access to the same physical memory. This means that processors can read and write to each other's memory without any overhead. This makes shared memory architectures very efficient for parallel computing, as processors can easily share data and communicate with each other.

Distributed memory architecture:

In a distributed memory architecture, each processor has its own local memory. This means that processors cannot directly access each other's memory. Instead, they must communicate with each other over a network to access data that is stored on other processors. Distributedmemory architectures are less efficient than shared memory architectures for parallel computing, but they can be scaled to much larger systems.

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Shared memory architecture** | **Distributed memory architecture** |
| Memory access | Direct access to physical memory | Access to local memory only |
| Communication | Not required | Required to access memory on other processors |
| Efficiency | More efficient | Less efficient |
| Scalability | Limited scalability | High scalability |

**List three factors that can cause parallel overhead according to the lecture and simply explain.**

Communication: Communication overhead occurs when processors need to communicate with each other to share data or coordinate their actions. This can be slow, especially if the processors are located on different machines in a distributed memory system.

Synchronization: Synchronization overhead occurs when processors need to wait for each other to finish their tasks before they can proceed. This can happen for a variety of reasons, such as when processors need to access shared data or when they need to coordinate their actions.

Task: Task overhead occurs when processors need to initialize, execute, and terminate tasks. This can be relatively small for simple tasks, but it can become significant for complex tasks.

**Give the formula of Amdahl’s Law including the number of processors.**

In computer architecture, amdahl’s law is a formula which gives theoretical speedup in latency of the execution of a task at a fixed workload that can be expected of a system whose resources are improved

Amdahl’s law is often used in parallel computing to predict the theoretical speedup when using multiple processors.

Speedup = 1/ (P/N) + S

Where P = parallel fraction, N is the number of processors and S serial function

**Please give the parallel programming models in common use and simply explain.**

There are several programming models in common use:

Shared memory model (without threads)

Threads

Distributed memory model

Data parallel model

Hybrid model

Single program multiple data (spmd)

Multiple program multiple data (mpmd)

Shared memory model (without threads) :

In this programming model, processes/ tasks share a common address space, which they read and write to asynchronously.

Various mechanisms such as locks/ semaphores are used to control access to shared nemory, resolve contentions and to prevent race conditions and deadlocks.

This is perhaps the simplest parallel programming model

A diagram of process and process

Description automatically generated

Thread model

This programming model is a type of shared memory programming

In the threads model of parallel programming, a single “heavy weight” process can have multiple “light weiaght” concurrent execution paths. There are two very different implementations of threads: POSIX Threads and OpenMP

POSIX threads – part of Unix/ linux operationg systems, library based, commonly referred as Pthreads, very explicit parallelism, requires significant programmer attention to detail

OpenMP (Open Multi Processing) – complier directive based, portable/multi-platform, including Unix and windows platforms, available in C/C++ and Fortan implementations, provides for “ incremental parallelism” can begin with serial code.

Distributed Memory/ Message Passing Model –

A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and /or across an arbitrary number of machines

Tasks exchange data through commumnications by sending and receiving messages

Data transfer usually requires cooperative operations to be performed by each process

Implementations:

Message passing implementations usually comprise a library of subroutines. Calls to these subroutines are imbedded in source code

The programmer is responsible for determining all parallelism

A diagram of a computer network

Description automatically generated

**Give 3 possibilities for poor parallel performance according to the lecture and simply explain.**

Poor load balancing: This occurs when threads are not evenly assigned tasks, resulting in some threads being idle while others are overloaded. This can lead to a slowdown in overall performance.

Data races: This occurs when two or more threads try to access and modify the same data at the same time. This can lead to unpredictable and incorrect results

Synchronization overhead: This occurs when threads need to wait for each other to finish their tasks before they can proceed. This can happen for a variety of reasons, such as when threads need to access shared data or when they need to coordinate their actions.

**Briefly explain two popular hybrid parallel programming models on current supercomputers and HPC clusters.**

**Single program multiple data (SPMD)** – SPMD is actually a high-level programming model that can be built upon any combination of the parallel programming models

Here single program – all tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data-parallel, or hybrid

Multiple data – All tasks may use different data

The SPMD model, using message passing or hybrid programming is probably the most commonly used parallel programming model for multi-node clusters

A close-up of a grey square

Description automatically generated

**Multiple program multiple data (MPMD) –**

Like SPMD, MPMD is actually high-level programming model that can be built upon any combination of the previously mentioned parallel programming models.

Multiple programs – Tasks execute different programs simultaneously. The programs can be threads, message passing, data parallel, or hybrid

Multiple data – All tasks may use different data

MPMD applications are not as common as SPMD applications, but may be better suited for certain types of problems, particularly those that lend themselves better to functional decomposition than domain decomposition.

A close-up of a red and blue square

Description automatically generated

**Please list the three primary API components in OpenMP and give simple examples.**

Three components – Compiler Directives, Runtime library routines, environment variables

Compiler directives have syntax as sentinel, directive-name, [clause..].

example: In c/c++ it’s #pragma omp parallel default (shared) private (beta, pi)

Runtime library routine – For c/c++, all of the runtime library routines are actual subroutines. Example in c/c++ as

#include<omp.h>

Int omp\_get\_num\_threads(void)

These routines are used for a variety of purposes – setting and querying the number of threads and querying the thread's unique ID.

Environment variable – In sh/bash is

export OMP\_NUM\_THREADS = 8(Discovery)

This environment variable can be used to control such things as: setting the number of threads, specifying how loop iterations are divided

**Please write the steps to use gcc compiler to compile a C file “hello.c” with OpenMP flag; set the environment variable of 4 threads on the bash-shell Linux operating system; and run this executable.**

gcc -o omp\_helloc – fopenmp omp\_hello.c

export OMP\_NUM\_THREADS = 4

./omp\_helloc