****

4th January 2020

By Prateek Tripathy

Group – Return 0;

**REport on high performance computing**

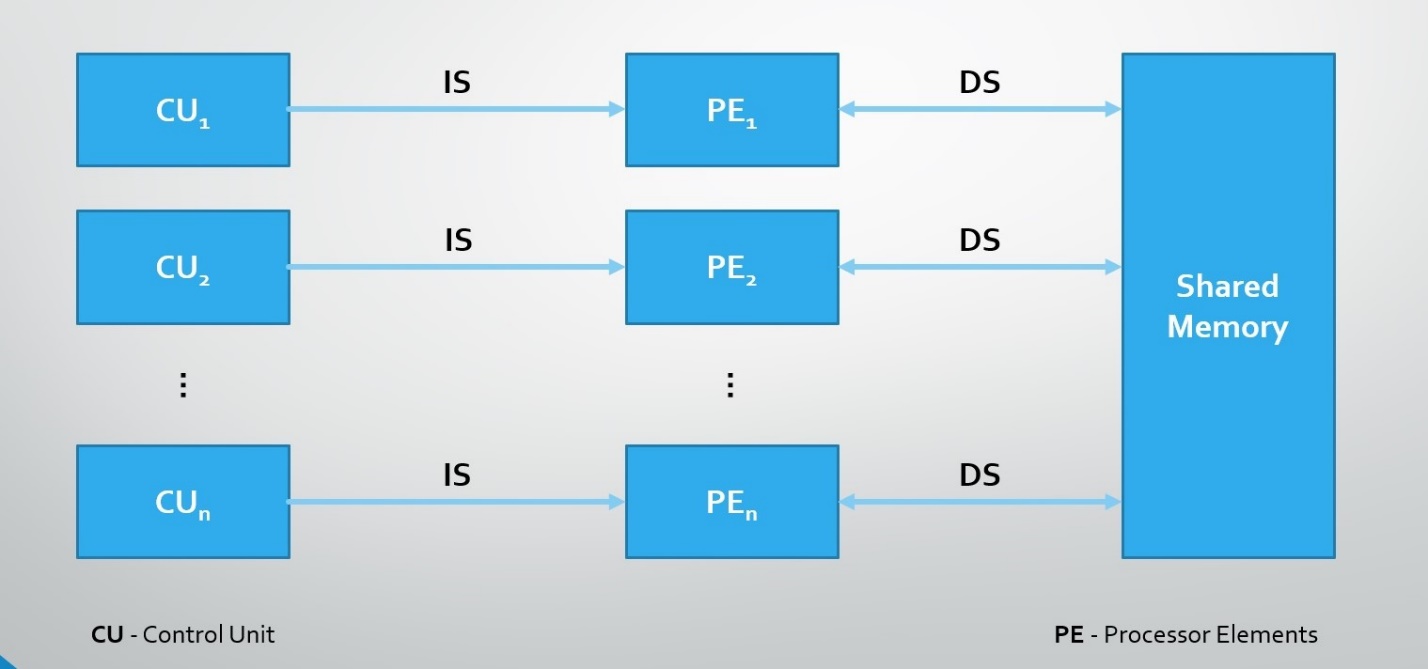
| **Table of content** | | |
| --- | --- | --- |
| Chapter Number | Topic | Page |
| 1 | Introduction | 3 |
| 2 | Pointers | 18 |
| 3 | OpenMP and MPI programs | 25 |
| 4 | Quantum Espresso, ITac and VTune | 57 |

1. **Introduction to parallel computing**

Parallel computing is the simultaneous execution of the same task, split into subtasks, on multiple processors in order to obtain the results faster. Traditionally, a task sent to a computer was accomplished one process at a time and this was termed as serial computing. Parallel computing is a method in computation in which two or more processors (or processor cores) handle different parts (processes) of an overall task simultaneously.

**How is parallel computing is done?**

An operating system can ensure that different tasks and user programmes are run in parallel on the available cores. However, for a serial software programme to take full advantage of the multi-core architecture ,the programmer needs to restructure and parallelise the code. A speed-up of application software runtime can no longer be achieved through frequency scaling, hence programmers parallise their software code to take advantage of the increasing computing power of multicore architectures.



**Concept of Temporal Parallelism:**

Temporal parallelism involves partitioning the processing task into a number of steps, which when applied sequentially to each unit of information produce the same results as the original task. In other words, the task is partitioned in time, with each step of the task being applied to a separate unit of information. A typical example is "assembly line" manufacturing. The application of temporal parallelism in computing produces pipelined structures.

**Concept of Data Parallelism :**

Data Parallelism means concurrent execution of the same task on each multiple computing core.

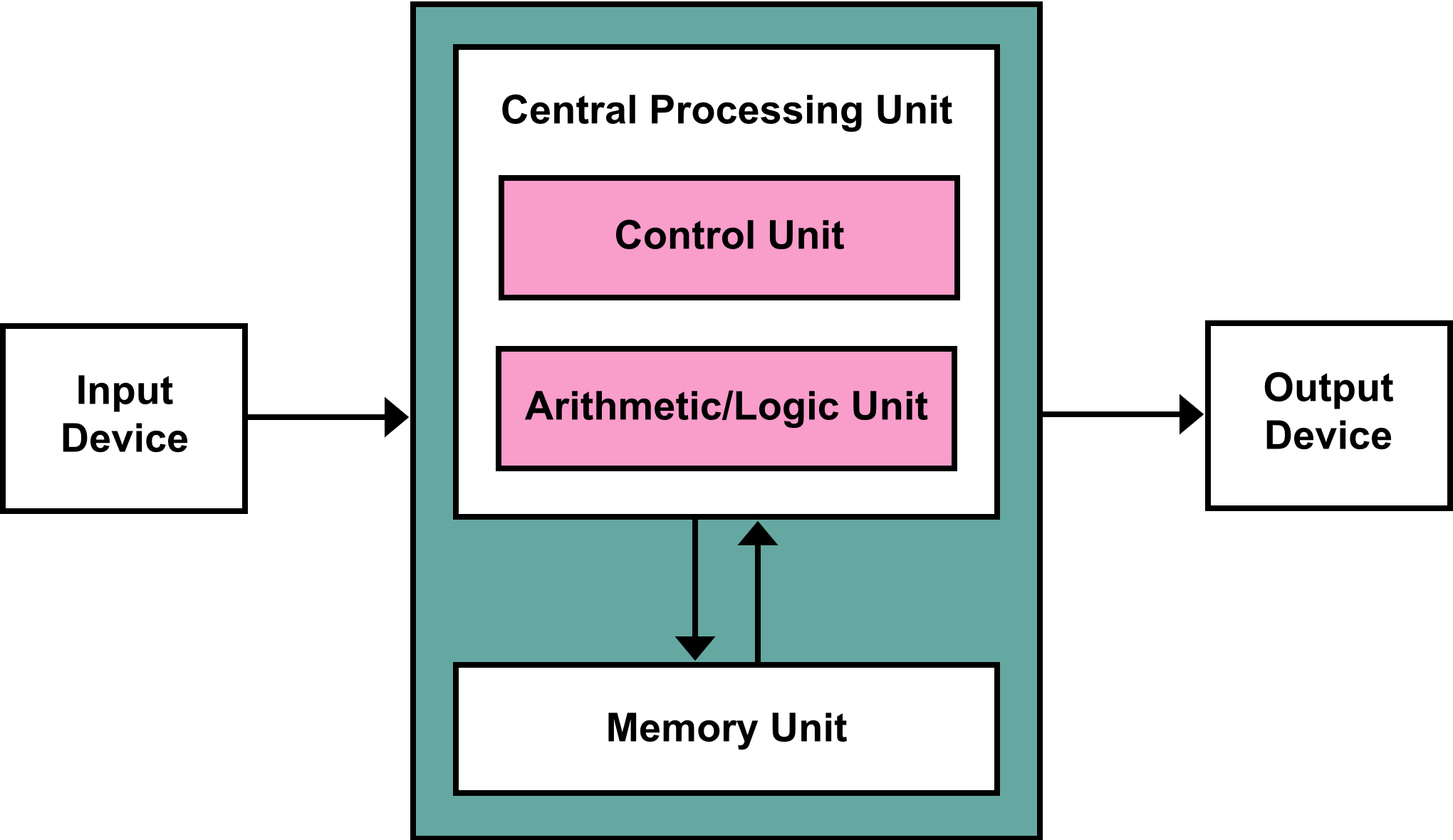
**Fundamentals of Computing and Multithreading:**

* Computer architecture is a set of rules and methods that describe the functionality, organization, and implementation of computer systems. Examples – the von Neumann architecture.
* A thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process.

**The von Neumann Architecture:**

* It is the computer architecture based on descriptions by the Hungarian mathematician and physicist John von Neumann and others who authored the general requirements for an electronic computer in their 1945 papers - First Draft of a Report on the EDVAC.
* The term "von Neumann architecture" has evolved to mean any stored-program computer in which an instruction fetch and a data operation are kept in a shared memory and cannot occur at the same time because they share a common bus.
* Differs from earlier computers which were programmed through "hard wiring".

Von Neumann Architecture Scheme:



**The von Neumann Bottleneck:**

* Due to the data memory and the program memory sharing a single bus in this von Neumann architecture, the limited throughput between the CPU and the memory compared to the memory available in most cases.
* Because the single bus can only access one of the two classes of memory at a time, throughput is lower than the rate at which the CPU can work.

# Havard Architechture

# The Harvard architecture is a computer architecture with separate storage and signal pathways for instructions and data. It contrasts with the von Neumann architecture, where program instructions and data share the same memory and pathways. In a Harvard architecture, there is no need to make the two memories share characteristics. In particular, the word width, timing, implementation technology, and memory address structure can differ. In some systems, instructions for pre-programmed tasks can be stored in read-only memory while data memory generally requires read-write memory. Also, a Harvard architecture machine has distinct code and data address spaces: instruction address zero is not the same as data address zero. Instruction address zero might identify a twenty-four-bit value, while data address zero might indicate an eight-bit byte that is not part of that twenty-four-bit value.

# Harvard architecture

* **Flynn’s Classical Taxonomy**

A screenshot of a cell phone

Description automatically generated

* Is one of the most widely used classification of parallel computers.

# SISD:

# Short for single instruction, single data. A type of parallel computing architecture that is classified under Flynn's taxonomy. A single processor executes a single instruction stream, to operate on data stored in a single memory. There is often a central controller that broadcasts the instruction stream to all the processing elements.

# MISD:

# Short for multiple instruction, single data. A type of parallel computing architecture that is classified under Flynn's taxonomy. Each processor owns its control unit and its local memory, making them more powerful than those used in SIMD computers. Each processor operates under the control of an instruction stream issued by its control unit, therefore the processors are potentially all executing different programs on different data while solving different sub-problems of a single problem. This means that the processors usually operate asynchronously.

# SIMD:

# Short for single instruction, multiple data. A type of parallel computing architecture that is classified under Flynn's taxonomy. A single computer instruction performs the same identical action (retrieve, calculate, or store) simultaneously on two or more pieces of data. Typically, this consists of many simple processors, each with a local memory in which it keeps the data which it will work on. Each processor simultaneously performs the same instruction on its local data progressing through the instructions in lock-step, with the instructions issued by the controller processor. The processors can communicate with each other in order to perform shifts and other array operations.

# MIMD :

# Short for multiple instruction, multiple data. A type of parallel computing architecture that is classified under Flynn's taxonomy. Multiple computer instructions, which may or may not be the same, and which may or may not be synchronized with each other, perform actions simultaneously on two or more pieces of data. The class of distributed memory MIMD machines is the fastest growing segment of the family of high-performance computers.

# Why use Multithreading?

With the introduction of multiple cores, multithreading has become extremely important in terms of the efficiency of your application. With multiple threads and a single core, your application would have to transition back and forth to give the illusion of multitasking.

With multiple cores, your application can take advantage of the underlying hardware to run individual threads through a dedicated core, thus making your application more responsive and efficient. Again, multithreading basically allows you to take full advantage of your CPU and the multiple cores, so you don’t waste the extra horsepower.

Developers should make use of multithreading for a few reasons:

* Higher throughput
* Responsive applications that give the illusion of multitasking.
* Efficient utilization of resources. Thread creation is light-weight in comparison to spawning a brand new process and for web servers that use threads instead of creating a new process when fielding web requests, consume far fewer resources.

# Simultaneous multithreading

# The most advanced type of multithreading applies to superscalar processors. Whereas a normal superscalar processor issues multiple instruction from a single thread every CPU cycle, in simultaneous multithreading (SMT) a superscalar processor can issue instructions from multiple threads every CPU cycle. Recognizing that any single thread has a limited amount of instruction-level parallelism, this type of multithreading tries to exploit parallelism available across multiple threads to decrease the waste associated with unused issue slots.

# For example:

# Cycle i: instructions j and j + 1 from thread A and instruction k from thread B are simultaneously issued.

# Cycle i + 1: instruction j + 2 from thread A, instruction k + 1 from thread B, and instruction m from thread C are all simultaneously issued.

# Cycle i + 2: instruction j + 3 from thread A and instructions m + 1 and m + 2 from thread C are all simultaneously issued.

**Fundamentals of Hyperthreading**

Modern processors can only handle one instruction from one program at any given point in time. Each instruction that is sent to the processor is called a thread. What I mean is that even though it looks like you're multitasking with your computer (running more than one program at a time) you're really not .

Dual CPU based systems can work on two independent threads of information from the software but each processor is still limited at working on one thread at any given moment though. The software must be able to dish out two separate pieces of information like Win2000 or Adobe Photoshop for a dual processor system to be really used, by the way.

For each [processor core](https://en.wikipedia.org/wiki/Processor_core) that is physically present, the [operating system](https://en.wikipedia.org/wiki/Operating_system) addresses two virtual (logical) cores and shares the workload between them when possible. The main function of hyper-threading is to increase **the number of independent instructions in the pipeline**; it takes advantage of [superscalar](https://en.wikipedia.org/wiki/Superscalar_processor) architecture, in which multiple instructions operate on separate data [in parallel](https://en.wikipedia.org/wiki/Parallel_computing). With HTT, one physical core appears as two processors to the operating system, allowing [concurrent](https://en.wikipedia.org/wiki/Concurrent_computing) scheduling of two processes per core. In addition, two or more processes can use the same resources: If resources for one process are not available, then another process can continue if its resources are available.

# Shared Memory:

# Shared memory generally have in common the ability for all processors to access the memory as a shared address space.

# Changes in a memory location effected by one processor are visible to all other processors.

# 

# Distributed Memory:

# Distributed memory generally requires a communication network to connect inter-processor memory.

# Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.

# 

# Hybrid Distributed Memory:

# Hybrid Distributed Memory implements both shared and distributed memory architectures.

# Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

# 

# Algorithm Paradigm

# An algorithm is a step by step procedure for solving a problem. Paradigm refers to the “pattern of thought” which governs scientific apprehension during a certain period of time. A paradigm can be viewed as a very high level algorithm for solving a class of problems. Various algorithms paradigms include Brute Force Paradigm, Divide and Conquer, Backtracking, Greedy Algorithm and Dynamic Programming Paradigm. Various Algorithms Paradigms are used to solve many types of problem according to the type of problem faced.

# Clusters: What Is an HPC Cluster?

# An HPC cluster consists of hundreds or thousands of compute servers that are networked together. Each server is called a node. The nodes in each cluster work in parallel with each other, boosting processing speed to deliver high-performance computing.

# HPC Use Cases

# Deployed on premises, at the edge, or in the cloud, HPC solutions are used for a variety of purposes across multiple industries. Examples include:

# 1. Research labs. HPC is used to help scientists find sources of renewable energy, understand the evolution of our universe, predict and track storms, and create new materials.

# 2. Media and entertainment. HPC is used to edit feature films, render mind-blowing special effects, and stream live events around the world.

# 3. Oil and gas. HPC is used to more accurately identify where to drill for new wells and to help boost production from existing wells.

# 4. Artificial intelligence and machine learning. HPC is used to detect credit card fraud, provide self-guided technical support, teach self-driving vehicles, and improve cancer screening techniques.

# 5. Financial services. HPC is used to track real-time stock trends and automate trading.

# 6. HPC is used to design new products, simulate test scenarios, and make sure that parts are kept in stock so that production lines aren’t held up.

# 7. HPC is used to help develop cures for diseases like diabetes and cancer and to enable faster, more accurate patient diagnosis.

# Performance. Delivers up to 1 million random read IOPS and 13GB/sec sustained (maximum burst) write bandwidth per scalable building block. Optimized for both flash and spinning media, the NetApp HPC solution includes built-in technology that monitors workloads and automatically adjusts configurations to maximize performance.

# Reliability. Fault-tolerant design delivers greater than 99.9999% availability, proven by more than 1 million systems deployed. Built-in Data Assurance features help make sure that data is accurate with no drops, corruption, or missed bits. Easy to deploy and manage. Modular design, on-the-fly (“cut and paste”) replication of storage blocks, proactive monitoring, and automation scripts all add up to easy, fast and flexible management.

# Scalability. A granular, building-block approach to growth that enables seamless scalability from terabytes to petabytes by adding capacity in any increment—one or multiple drives at a time. Lower TCO. Price/performance-optimized building blocks and the industry’s best density per delivers low power, cooling, and support costs, and 4-times lower failure rates than commodity HDD and SSD devices.

# Green Computing:

# Green computing is a contemporary research topic to address climate and energy challenges. For instance, in order to provide electricity for large-scale cloud infrastructures and to reach exascale computing, we need huge amounts of energy. Thus, green computing is a challenge for the future of cloud computing and HPC. Alternatively, clouds and HPC provide solutions for green computing and climate change.Green Computing provides an incentive for computing engineers to come up with such an method so that HPCs can be highly efficient and green for the environment at the same time by being energy efficient.

# For a data center like the MGHPCC, energy efficiency means minimizing the amount of non-computing “overhead” energy used for cooling, lighting, and power distribution. Energy modeling during the design phase estimated a 43% reduction in energy costs compared to the baseline standard (ASHRAE Standard 90.1-2007), and a 44% reduction in lighting power density for building exteriors below the baseline standard.

# Paradigm as Shared Memory:

# Usually indicated as Multithreading Programming

# • Commonly implemented in scientific computing using the OpenMP standard (directive based)

# • Thread management overhead

# • Limited scalability

# • Write access to shared data can easily lead to race conditions and incorrect data

# Total Parallel Overhead:

# The overheads incurred by a parallel program are encapsulated into a single expression referred to as the overhead function. We define overhead function or total overhead of a parallel system as the total time collectively spent by all the processing elements over and above that required by the fastest known sequential algorithm for solving the same problem on a single processing element. We denote the overhead function of a parallel system by the symbol To.

# The total time spent in solving a problem summed over all processing elements is pTP . TS units of this time are spent performing useful work, and the remainder is overhead. Therefore, the overhead function (To) is given by

# Efficiency:

# Only an ideal parallel system containing p processing elements can deliver a speedup equal to p. In practice, ideal behavior is not achieved because while executing a parallel algorithm, the processing elements cannot devote 100% of their time to the computations of the algorithm. As we saw in Example 5.1, part of the time required by the processing elements to compute the sum of n numbers is spent idling (and communicating in real systems). Efficiency is a measure of the fraction of time for which a processing element is usefully employed; it is defined as the ratio of speedup to the number of processing elements. In an ideal parallel system, speedup is equal to p and efficiency is equal to one. In practice, speedup is less than p and efficiency is between zero and one, depending on the effectiveness with which the processing elements are utilized. We denote efficiency by the symbol E. Mathematically, it is given by E = S/P.

# Cost:

# We define the cost of solving a problem on a parallel system as the product of parallel runtime and the number of processing elements used. Cost reflects the sum of the time that each processing element spends solving the problem. Efficiency can also be expressed as the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on p processing elements.

# The cost of solving a problem on a single processing element is the execution time of the fastest known sequential algorithm. A parallel system is said to be cost-optimal if the cost of solving a problem on a parallel computer has the same asymptotic growth (in Q terms) as a function of the input size as the fastest-known sequential algorithm on a single processing element. Since efficiency is the ratio of sequential cost to parallel cost, a cost-optimal parallel system has an efficiency of Q(1)

# 

# Open MP

OpenMP is an Application Program Interface (API). It provides a portable, scalable model for developers of shared memory parallel applications. The API supports C/C++ and Fortran on a wide variety of architectures.

It consists of 3 main parts –

1. Compiler directives (eg #pragma omp parallel)

2. Runtime Library Routines (eg omp\_get\_num\_threads())

3. Environment Variables (eg OMP\_NUM\_THREADS)

**Why Open MP?**

It provides a standard among a variety of shared memory architectures/platforms and establishes a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives. It provides capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach. For High Performance Computing (HPC) applications, OpenMP is combined with MPI for the distributed memory parallelism. This is often referred to as Hybrid Parallel Programming. OpenMP is used for computationally intensive work on each node. MPI is used to accomplish communications and data sharing between nodes.

**How Open MP works-**

1. Provides a standard among a variety of shared memory architectures/platforms.

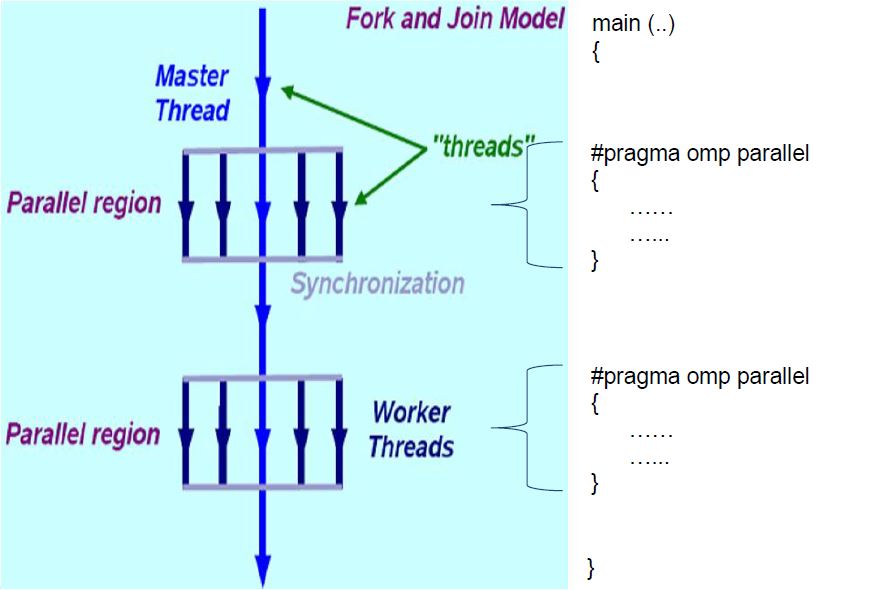
2. Establish a simple and limited set of directives for programming shared memory machines.

3. Significant parallelism can be implemented by using just 3 or 4 directives.

4. Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach

5. For High Performance Computing (HPC) applications, OpenMP is combined with MPI for the distributed memory parallelism. This is often referred to as Hybrid Parallel Programming. OpenMP is used for computationally intensive work on each node

6. MPI is used to accomplish communications and data sharing between nodes.



**What are the main parts of Open MP?**

1. Compiler Directives- It’s an instruction to the compiler to change how it’s compiling the code, rather than a piece of the code itself. #include and #define in C/C++ are considered directives, but they’re instructions to another program - the preprocessor. A true compiler directive might be something like a pragma, which is a compiler-specific command for changing what the compiler does, typically error handling.

OpenMP compiler directives are used for various purposes:

* 1. Spawning a parallel region
  2. Dividing blocks of code among threads
  3. Distributing loop iterations between threads
  4. Serializing sections of code
  5. Synchronization of work among threads.
  6. Syntax- #pragma omp parallel default (shared)

2. Run time Libraries- These routines are used for a variety of purposes:

Setting and querying the number of threads. Setting and querying the dynamic threads features and querying if in a parallel region, and at what level. for eg.

#include<omp.h>

int omp\_get\_num\_threads(void)

3.Environment Variables- OpenMP provides several environment variables for controlling the execution of parallel code at run-time.These environment variables can be used to control such things as:

Setting the number of threads.

Specifying how loop interations are divided

Binding threads to processors.

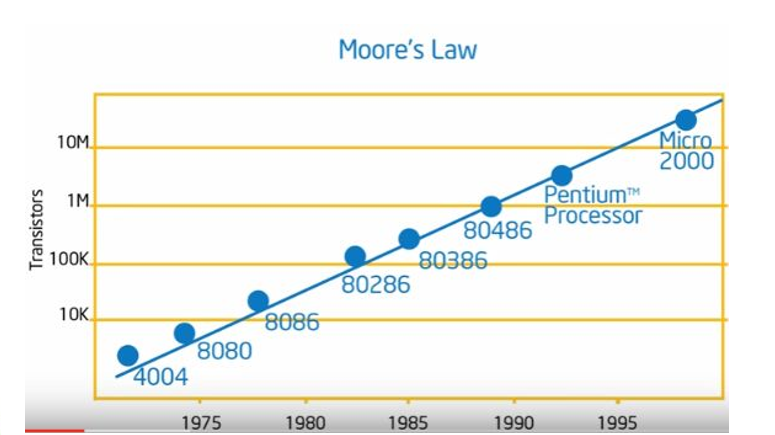
For eg. OMP\_GET\_THREADS, OMP\_STACKSIZE.

**Message Parsing Interface**

The Message Passing Interface (MPI) is a library specification that allows HPC to pass information between its various nodes and clusters. HPC uses OpenMPI, an open-source, portable implementation of the MPI standard. OpenMPI contains a complete implementation of version 1.2 of the MPI standard and also MPI-2. Compilers used by MPI include GNU implementation of C, C++ and Fortran.

**Moore's Law:**

Moore's Law refers to Moore's perception that the number of transistors on a microchip doubles every two years, though the cost of computers is halved. Moore's Law states that we can expect the speed and capability of our computers to increase every couple of years, and we will pay less for them. Another tenet of Moore's Law asserts that this growth is exponential. Today, however, the doubling of installed transistors on silicon chips occurs closer to every 18 months instead of every two years.



* Moore's Law states that the number of transistors on a microchip doubles about every two years, though the cost of computers is halved.
* In 1965, Gordon E. Moore, the co-founder of Intel, made this observation that became Moore's Law.
* Another tenet of Moore's Law says that the growth of microprocessors is exponential.
* Moore's Law states that the number of transistors on a microchip doubles about every two years, though the cost of computers is halved.
* In 1965, Gordon E. Moore, the co-founder of Intel, made this observation that became Moore's Law.
* Another statement of Moore's Law says that the growth of microprocessors is exponential.

Moore's Law has been a driving force of technological and social change, productivity, and economic growth that are hallmarks of the late-twentieth and early twenty-first centuries.

**2. Pointers:**

*How branch predictor algorithm works?*

It is a digital circuit which tries to predict in which way a branch would go before the proper command is to be executed. Its purpose is to improve the flow in the instruction pipeline. Branch predictors play a critical role in achieving highly effective performance in modern pipelined microprocessors such as x86. It is based on speculative algorithm. Branch predictor predicts what the next line of code would be, and speculative algorithm predicts what the output of that code would be, before it is actually being executed.

*Implementation of Branch Prediction.*

Static Branch Prediction- Static prediction is the simplest branch prediction technique because it does not rely on information about the dynamic history of code executing. Instead, it predicts the outcome of a branch based solely on the branch instruction. Evaluates branches in decode stage and have a single cycle instruction fetch.

Dynamic Branch Prediction- Uses information about taken or not taken branches gathered at run time to predict the outcome of a branch.

Random Branch Prediction – Random Branch Predictor uses algorithm which predicts what the next branch would come to the process cycle to be executed at run time. It has a prediction rate of around 50% .

Next line prediction – fetches each line of instruction with a pointer to the next line. The next line predictor points to aligned pointers and predicts its outcome so that the execution time is less.

*What is Cache Coherency?*

Cache coherence is the uniformity of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

When more than 1 cahce are connected with each other, there is a ambiguity of data, and this leads to inconsistency of data. This is called cache coherency.

*Process vs Threads*

A process usually represents an independent execution unit with its own memory area, system resources and scheduling slot.

A thread is typically a "division" within the process which usually share the same memory and operating system resources, and share the time allocated to that process.

Process operations are controlled by PCB which is a kernel data structure. PCB uses the three kinds of functions which are scheduling, dispatching and context save.

For thread,the kernel allocates a stack and a thread control block (TCB) to each thread. Threads are implemented in three different ways:kernel-level threads, user-level threads, hybrid threads. Threads can have three states running, ready and blocked.

A thread can't have individual existence whereas process can exit individually.

A process is heavy weighted, but a thread is light weighted.

*Limitations of Amdahl’s Law*

Limitations of Amdahl's Law

1. Shared resources have to be used serially

2. No task is perfectly parallelizable.

3.There is Load imbalance

4.Task interdependencies must be accounted

5. Coordinating communications among the various processes will require additional code. This adds to the overall execution time.

6. The amount of speedup depends on the size of the problem.

*Loosely Coupled vs Tightly coupled*

**

*Cache Memory and Levels of Cache Memory*

A Cache is used by the CPU to access data from the main memory in short time. It is a small and very fast temporary storage memory. It is designed to speed up the transfer of data or instructions. CPU Cache is located inside or near to the CPU chip. The data/instructions which are most recently or frequently used by the CPU are stored in CPU. A copy of data/instructions is stored as a cache when the CPU uses them for the first time which retrieved from RAM. The next time when CPU needs the data/instruction, it looks in the cache. If the required data/instruction is found there, then it is retrieved from the cache memory instead of main memory.

*Types/Levels of cache memory*

A computer has several different levels of cache memory. All levels of cache memory are faster than the RAM. The cache which is closer to the CPU is always faster than the other levels but it costs more and stores less data than other levels. As multiple processors operate in parallel, and independently multiple caches may possess different copies of the same memory block, this creates cache coherence problem. Cache coherence schemes help to avoid this problem by maintaining a uniform state for each cached block of data.

Types of Cache Memory in a CPU

*Level 1 or L1 Cache Memory*

The L1 cache memory is built on processor chip and it is very fast because it runs on the speed of the processor. It is also called primary or internal cache. It has less memory compared to other levels of cache and can store up to the 64kb cache memory. This cache is made of SRAM (Static RAM). Each time the processor requests information from memory, the cache controller on the chip uses special circuitry to first check if the memory data is already in the cache. If it is present, then the system is spared from the time-consuming access to the main memory. L1 cache is also usually split two ways, into the instruction cache and the data cache. The instruction cache deals with the information about the operation that the CPU has to perform, while the data cache holds the data on which the operation is to be performed.

Examples of L1 cache are accumulator, Program counter and address register, etc

*Level 2 or L2 Cache Memory*

The L2 cache memory is larger but slower than L1 cache. It is used to see recent accesses that are not picked by the L1 cache and it usually stores 64kb to the 2MB cache memory. An L2 cache is also found on the CPU. If L1 and L2 cache are used together, then the missing information that is not present in the L1 cache can be retrieved quickly from the L2 cache. Like L1 caches, L2 caches are composed of SRAM but they are larger. L2 is usually a separate static RAM (SRAM) chip and it is located between the CPU and DRAM (Main memory).

*Level 3 or L3 Cache Memory*

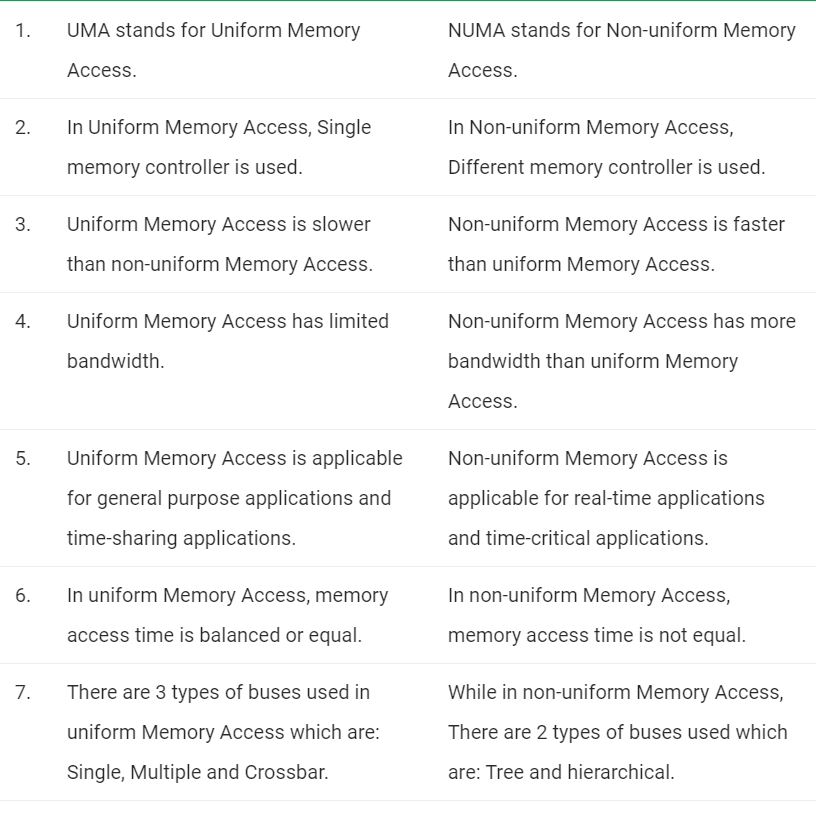
The L3 Cache memory is an enhanced form of memory present on the motherboard of the computer. It is an extra cache built into the motherboard between the processor and main memory to speed up the processing operations. It reduces the time gap between request and retrieving of the data and instructions much more quickly than the main memory. L3 cache is being used with processors nowadays, having more than 3MB of storage in it.

*What are sockets?*

Sockets allow communication between two different processes on the same or different machines. To be more precise, it's a way to talk to other computers using standard Unix file descriptors

To a programmer, a socket looks and behaves much like a low-level file descriptor. This is because commands such as read() and write() work with sockets in the same way they do with files and pipes.

*Uniform Memory Access vs Non Uniform Memory Access*



*What is context Switching?*

Context Switching involves storing the context or state of a process so that it can be reloaded when required and execution can be resumed from the same point as earlier. This is a feature of a multitasking operating system and allows a single CPU to be shared by multiple processes.

*Clustering and how Clusters handle Load Balancing*

A cluster is a group of resources that are trying to achieve a common objective, and are aware of one another. Clustering usually involves setting up the resources (servers usually) to exchange details on a particular channel (port) and keep exchanging their states, so a resource’s state is replicated at other places as well. It usually also includes load balancing, wherein, the request is routed to one of the resources in the cluster as per the load balancing policy.

Load balancing can also happen without clustering when we have multiple independent servers that have same setup, but other than that, are unaware of each other. Then, we can use a load balancer to forward requests to either one server or other, but one server does not use the other server’s resources. Also, one resource does not share its state with other resources. Each load balancer basically does following tasks: Continuously check which servers are up. When a new request is received, send it to one of the servers as per the load balancing policy. When a request is received for a user who already has a session, send the user to the same server.

*What is Interconnect and its types used in HPCs?*

Interconnect is the way by which various computers communicate with each other. The biggest advantage HPCs have over ordinary consumer level computers is the Interconnect technology used by them, which allows them to significantly boost efficiency, and allows them to utilise the resources of other computers. High performance system interconnect technology can be divided into three categories: Ethernet, InfiniBand, and vendor specific interconnects, which includes custom interconnects the recently introduced Intel Omni-Path technology.

Ethernet as an Interconnect- Ethernet is established as the dominant low level interconnect standard for mainstream commercial computing requirements. Above the physical level, the software layers to coordinate communication resulted in TCP/IP becoming widely adopted as the primary commercial networking protocol. Ethernet is established as the dominant low level interconnect standard for mainstream commercial computing requirements. Above the physical level, the software layers to coordinate communication resulted in TCP/IP becoming widely adopted as the primary commercial networking protocol.

Infiniband as an Interconnect- InfiniBand is designed for scalability, using a switched fabric network topology together with remote direct memory access (RDMA) to reduce CPU overhead. The InfiniBand protocol stack is considered less burdensome than the TCP protocol required for Ethernet. This enables InfiniBand to maintain a performance and latency edge in comparison to Ethernet in many high performance workloads, and is generally used in Cluster Computers.

3. Open MP and MPI programs

**1. Write a MPI program that should print your name only if number of processes is even otherwise return error message.**

#include<stdio.h>

#include<mpi.h>

int main() {

int comm\_sz;

int my\_rank;

MPI\_Init(NULL, NULL);

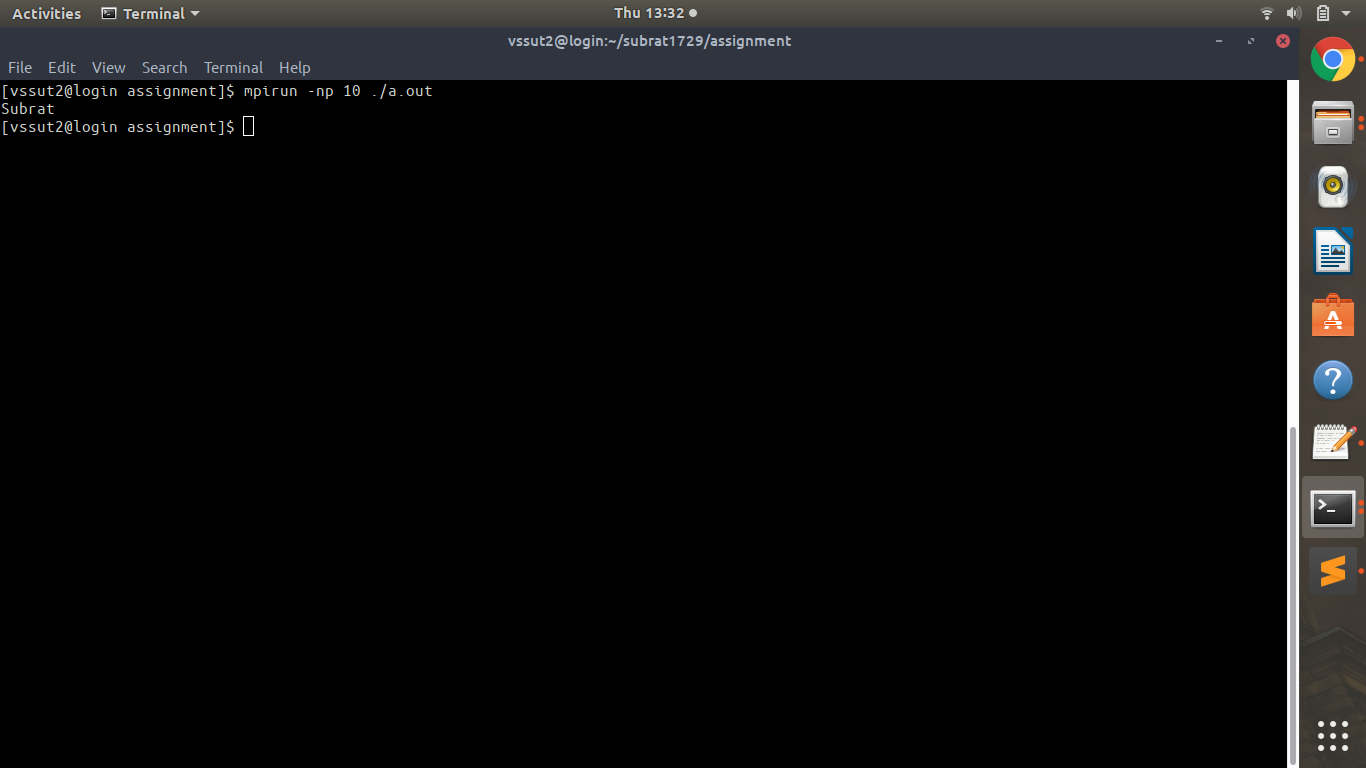
MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); if(comm\_sz%2==0 && my\_rank==0) printf(“Subrat\n");

else if(my\_rank==0)

printf("Error\n");

MPI\_Finalize();

}



**2. Write a MPI program that should determine partner process and then send and receive message (Your name and number) with it.**

#include<stdio.h>

#include<string.h>

#include<mpi.h>

int main() {

int comm\_sz;

int my\_rank;

char name[100];

MPI\_Init(NULL, NULL);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); if(my\_rank==0) {

int n;

strcpy(name, "Subrat"); for(int i=1; i<comm\_sz; i++) {

MPI\_Send(name, 100, MPI\_CHAR, i, 0, MPI\_COMM\_WORLD); MPI\_Recv(&n, 1, MPI\_INT, i, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("%d\n", n);

}

}

else {

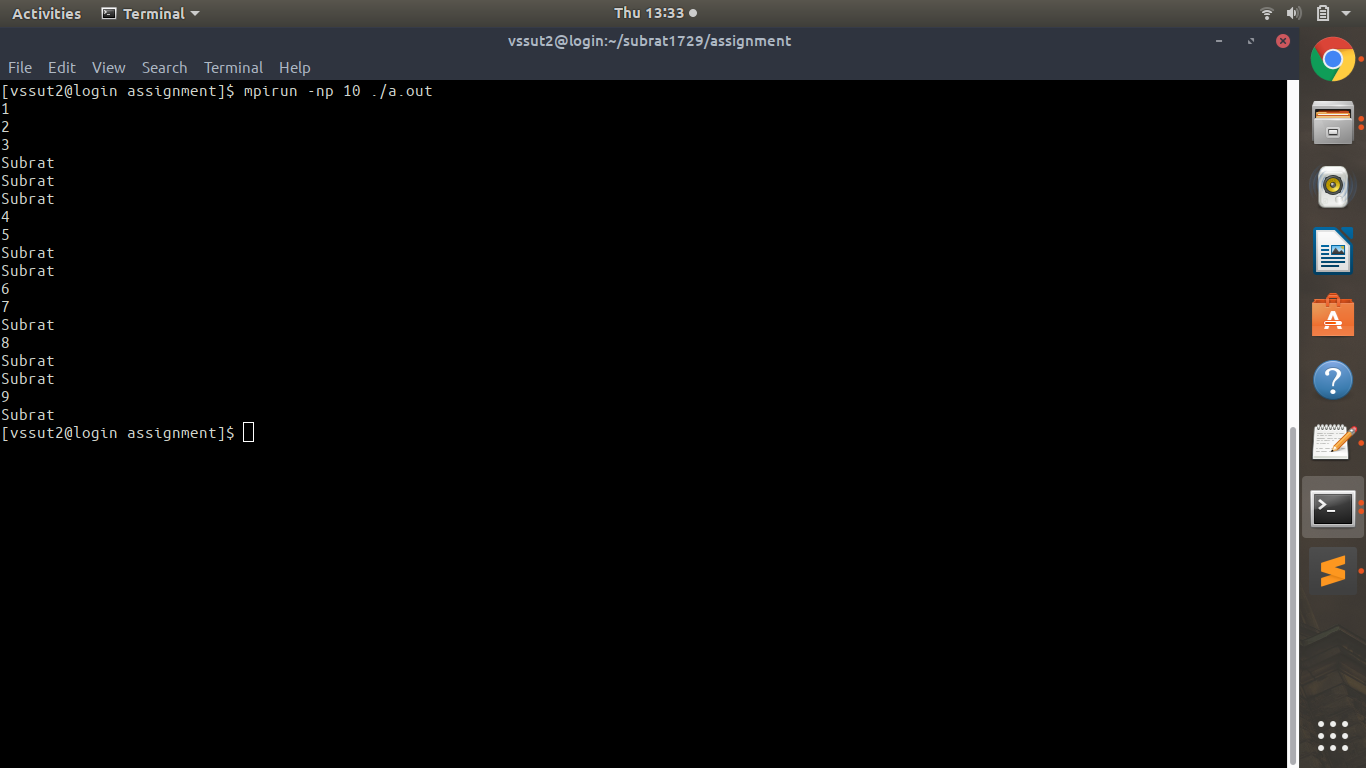
MPI\_Recv(name, 100, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

MPI\_Send(&my\_rank, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD); printf("%s\n", name);

}

MPI\_Finalize();

}



**3. Observe the difference between blocking and non-blocking communication**

1. **Blocking communication:-**

#include<stdio.h>

#include<string.h>

#include<mpi.h>

#define me 0

#define partner 1

#define MAX\_STRING 1000

int main(void)

{

char greeting[MAX\_STRING],greeting1[MAX\_STRING],greeting2[MAX\_STRING],greeting3[MAX\_STRING];

int comm\_sz ;

int my\_rank ;

MPI\_Init(NULL, NULL);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

if(my\_rank==partner)

{

sprintf(greeting, "Welcome to the world of Parallel Computing. greeting2 I am Process no %d out of %d\n", my\_rank, comm\_sz);

MPI\_Send(greeting, strlen(greeting)+1, MPI\_CHAR, me, 0, MPI\_COMM\_WORLD) ;

MPI\_Recv(greeting, MAX\_STRING, MPI\_CHAR,me, 0, MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

printf("%s\n",greeting);

}

else

{

MPI\_Recv(greeting, MAX\_STRING, MPI\_CHAR,partner, 0, MPI\_COMM\_WORLD,MPI\_STATUS\_IGNORE);

printf("%s\n",greeting);

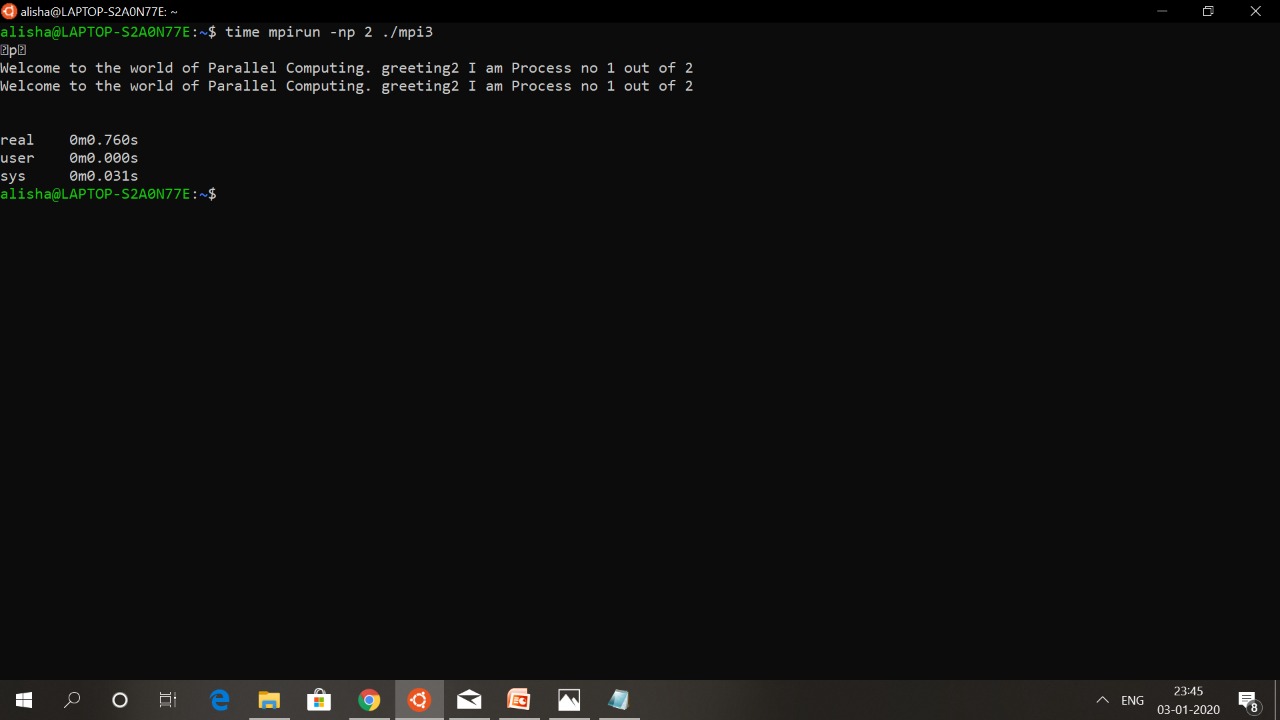
sprintf(greeting, "Welcome to the world of Parallel Computing.greeting I am Process no %d out of %d\n", my\_rank, comm\_sz);

MPI\_Send(greeting, strlen(greeting)+1, MPI\_CHAR, partner, 0, MPI\_COMM\_WORLD) ;

}

MPI\_Finalize();

return 0;

}

1. **Non-blocking communication:-**

#include<stdio.h>

#include<string.h>

#include<mpi.h>

#define me 0

#define partner 1

#define MAX\_STRING 1000

int main(void)

{

char greeting[MAX\_STRING],greeting1[MAX\_STRING],greeting2[MAX\_STRING],greeting3[MAX\_STRING];

int comm\_sz ;

int my\_rank ;

MPI\_Request request,request2;

MPI\_Init(NULL, NULL);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

if(my\_rank==partner)

{

sprintf(greeting, "Welcome to the world of Parallel Computing. greeting2 I am Process no %d out of %d\n", my\_rank, comm\_sz);

MPI\_Isend(greeting, strlen(greeting)+1, MPI\_CHAR, me, 0, MPI\_COMM\_WORLD,&request) ;

printf("%s\n",greeting);

}

else

{

MPI\_Irecv(greeting, MAX\_STRING, MPI\_CHAR,partner, 0, MPI\_COMM\_WORLD,&request2);

printf("%s\n",greeting);

sprintf(greeting, "Welcome to the world of Parallel Computing.greeting I am Process no %d out of %d\n", my\_rank, comm\_sz);

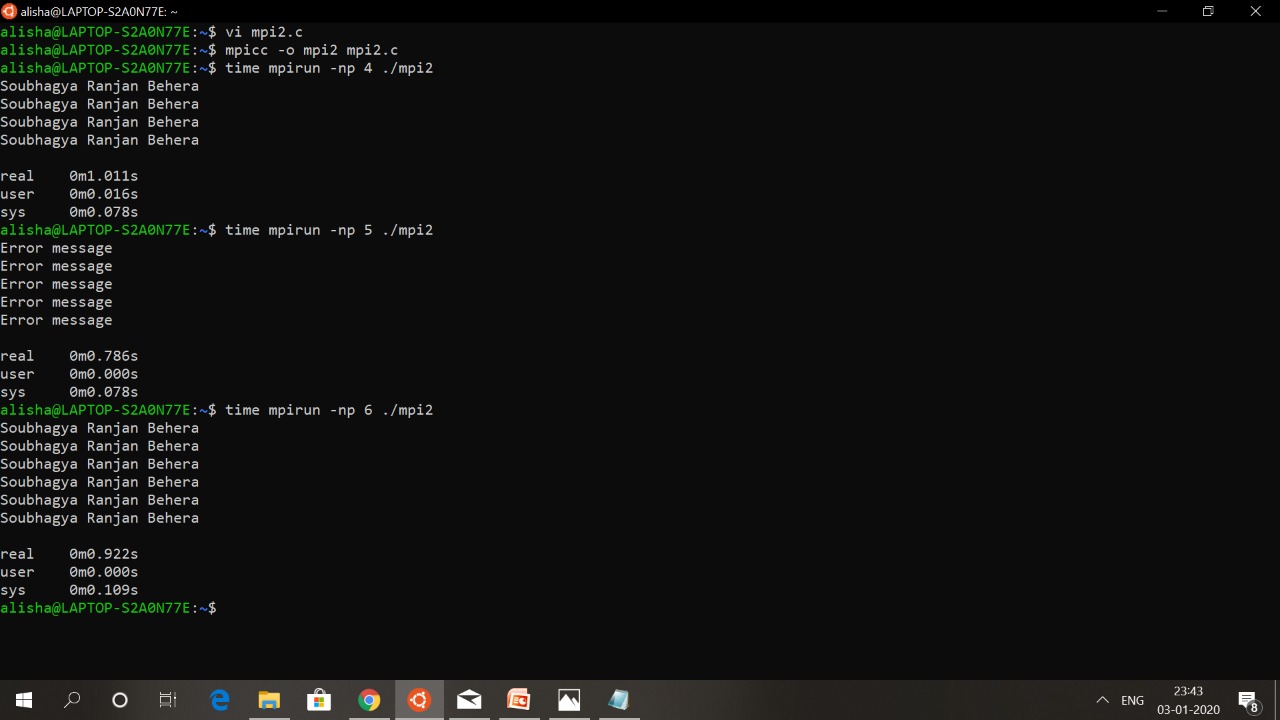
MPI\_Isend(greeting, strlen(greeting)+1, MPI\_CHAR, partner, 0, MPI\_COMM\_WORLD,&request) ;

printf("%s",greeting);

}

MPI\_Finalize();

return 0;

}

**4. Write a C program to calculate the value of pi. (DartBoard Algorithm) hint- divide no of darts.**

#include<stdio.h>

#include<mpi.h>

#define R 97

#define NUM\_SQUARE 10

int main() {

int px, py;

int limit = 2\*R + 1;

int comm\_sz;

int my\_rank;

MPI\_Init(NULL, NULL);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); long ncircle = 0;

for(long i=0; i<NUM\_SQUARE/comm\_sz; i++) {

px = rand()%limit - R;

py = rand()%limit - R;

printf("(%d, %d)\n", px, py); if((px\*px + py\*py) < R\*R) ncircle++;

}

if(my\_rank==0) {

int a[comm\_sz];

long num\_circle = ncircle; for(int q=1; q < comm\_sz; q++) MPI\_Recv(&a[q], 1, MPI\_INT, q, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

for(int i=1; i<comm\_sz; i++) num\_circle += a[i];

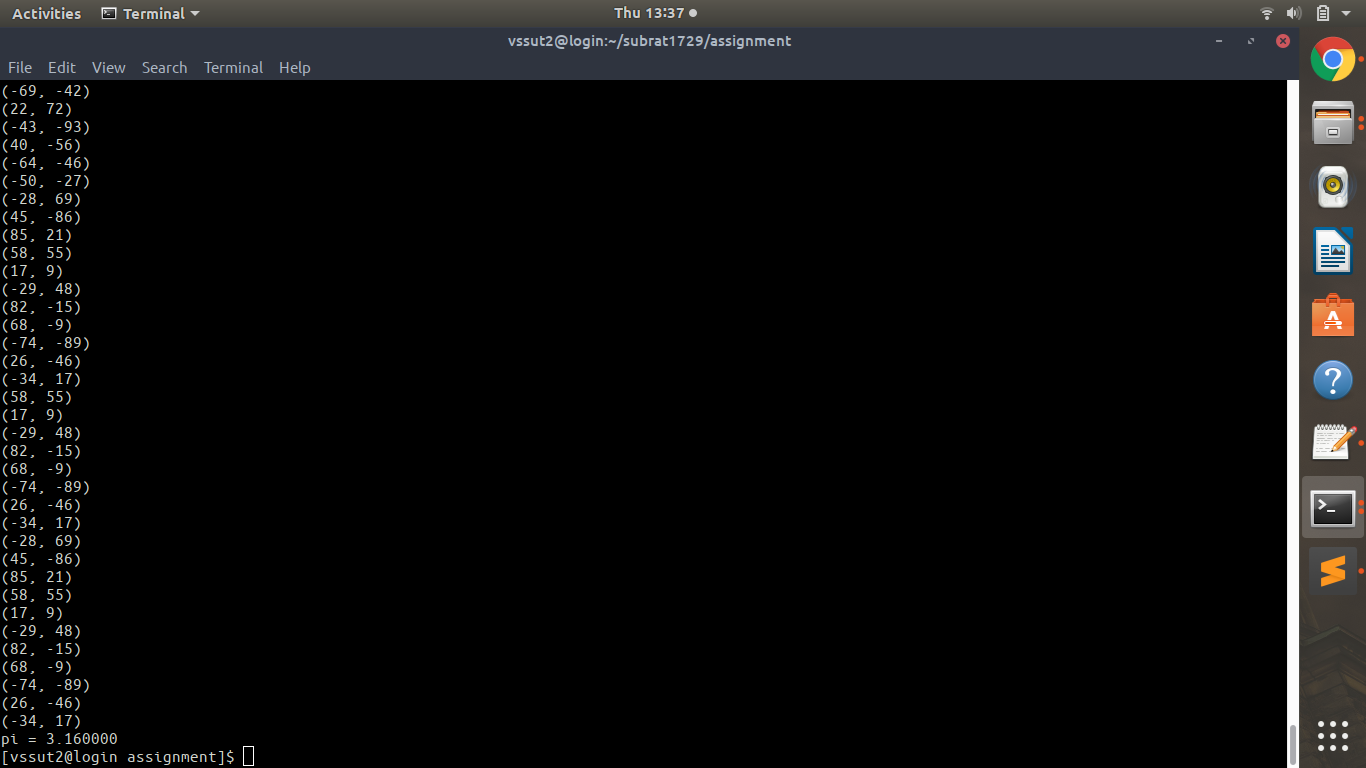
printf("pi = %Lf\n", (long double) (4 \* ((long double) num\_circle/NUM\_SQUARE)));

}

else

MPI\_Send(&ncircle, 1, MPI\_INT, 0, 0, MPI\_COMM\_WORLD); MPI\_Finalize();

}



**5. Write a C-Program that initialise matrix A and B (user’s size), multiply both matrix and store result in matrix C.**

#include "mpi.h"

#include <stdio.h>

#include <stdlib.h>

#define NRA 62 /\* number of rows in matrix A \*/

#define NCA 15 /\* number of columns in matrix A \*/

#define NCB 7 /\* number of columns in matrix B \*/

#define MASTER 0 /\* taskid of first task \*/

#define FROM\_MASTER 1 /\* setting a message type \*/

#define FROM\_WORKER 2 /\* setting a message type \*/

int main (int argc, char \*argv[])

{

int numtasks, /\* number of tasks in partition \*/

taskid, /\* a task identifier \*/

numworkers, /\* number of worker tasks \*/

source, /\* task id of message source \*/

dest, /\* task id of message destination \*/

mtype, /\* message type \*/

rows, /\* rows of matrix A sent to each worker \*/

averow, extra, offset, /\* used to determine rows sent to each worker \*/

i, j, k, rc; /\* misc \*/

double a[NRA][NCA], /\* matrix A to be multiplied \*/

b[NCA][NCB], /\* matrix B to be multiplied \*/

c[NRA][NCB]; /\* result matrix C \*/

MPI\_Status status;

MPI\_Init(&argc,&argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&taskid);

MPI\_Comm\_size(MPI\_COMM\_WORLD,&numtasks);

if (numtasks < 2 ) {

printf("Need at least two MPI tasks. Quitting...\n");

MPI\_Abort(MPI\_COMM\_WORLD, rc);

exit(1);

}

numworkers = numtasks-1;

if (taskid == MASTER)

{

printf("mpi\_mm has started with %d tasks.\n",numtasks);

printf("Initializing arrays...\n");

for (i=0; i<NRA; i++)

for (j=0; j<NCA; j++)

a[i][j]= i+j;

for (i=0; i<NCA; i++)

for (j=0; j<NCB; j++)

b[i][j]= i\*j;

/\* Send matrix data to the worker tasks \*/

averow = NRA/numworkers;

extra = NRA%numworkers;

offset = 0;

mtype = FROM\_MASTER;

for (dest=1; dest<=numworkers; dest++)

{

rows = (dest <= extra) ? averow+1 : averow;

printf("Sending %d rows to task %d offset=%d\n",rows,dest,offset);

MPI\_Send(&offset, 1, MPI\_INT, dest, mtype, MPI\_COMM\_WORLD);

MPI\_Send(&rows, 1, MPI\_INT, dest, mtype, MPI\_COMM\_WORLD);

MPI\_Send(&a[offset][0], rows\*NCA, MPI\_DOUBLE, dest, mtype,

MPI\_COMM\_WORLD);

MPI\_Send(&b, NCA\*NCB, MPI\_DOUBLE, dest, mtype, MPI\_COMM\_WORLD);

offset = offset + rows;

}

/\* Receive results from worker tasks \*/

mtype = FROM\_WORKER;

for (i=1; i<=numworkers; i++)

{

source = i;

MPI\_Recv(&offset, 1, MPI\_INT, source, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&rows, 1, MPI\_INT, source, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&c[offset][0], rows\*NCB, MPI\_DOUBLE, source, mtype,

MPI\_COMM\_WORLD, &status);

printf("Received results from task %d\n",source);

}

/\* Print results \*/

printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf("Result Matrix:\n");

for (i=0; i<NRA; i++)

{

printf("\n");

for (j=0; j<NCB; j++)

printf("%6.2f ", c[i][j]);

}

printf("\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

printf ("Done.\n");

}

if (taskid > MASTER)

{

mtype = FROM\_MASTER;

MPI\_Recv(&offset, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&rows, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&a, rows\*NCA, MPI\_DOUBLE, MASTER, mtype, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&b, NCA\*NCB, MPI\_DOUBLE, MASTER, mtype, MPI\_COMM\_WORLD, &status);

for (k=0; k<NCB; k++)

for (i=0; i<rows; i++)

{

c[i][k] = 0.0;

for (j=0; j<NCA; j++)

c[i][k] = c[i][k] + a[i][j] \* b[j][k];

}

mtype = FROM\_WORKER;

MPI\_Send(&offset, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD);

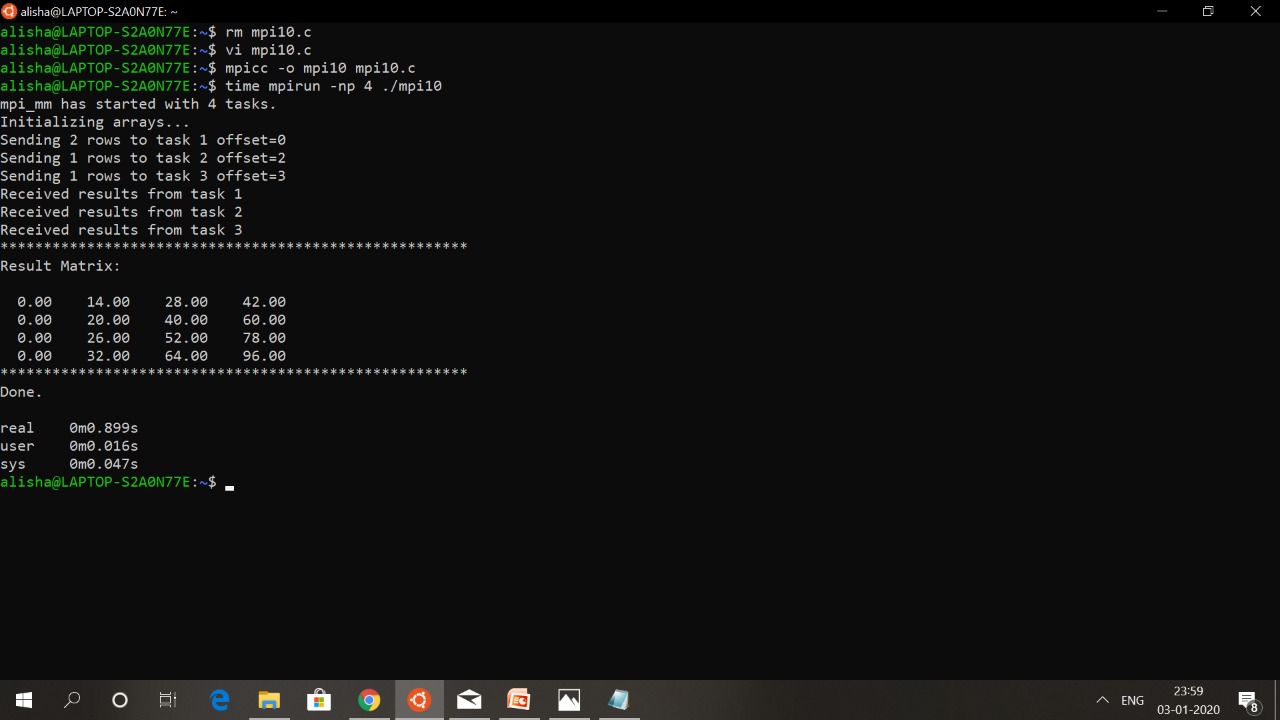
MPI\_Send(&rows, 1, MPI\_INT, MASTER, mtype, MPI\_COMM\_WORLD);

MPI\_Send(&c, rows\*NCB, MPI\_DOUBLE, MASTER, mtype, MPI\_COMM\_WORLD);

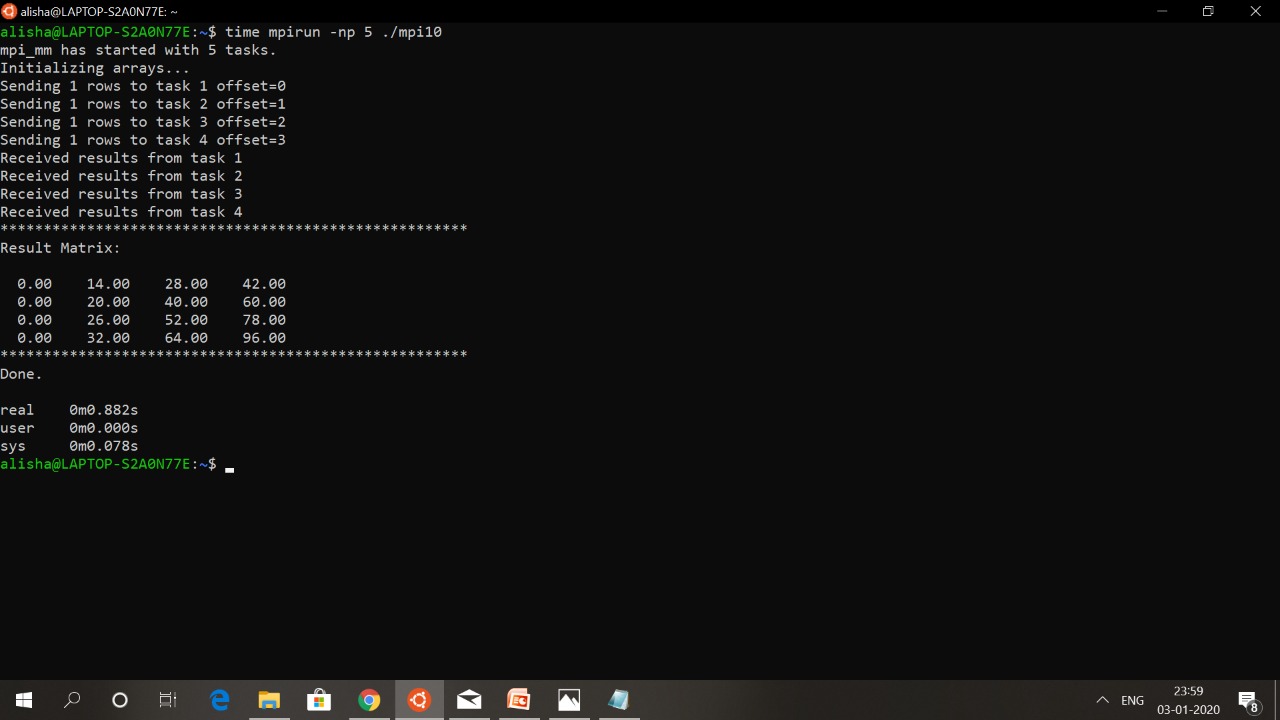
}

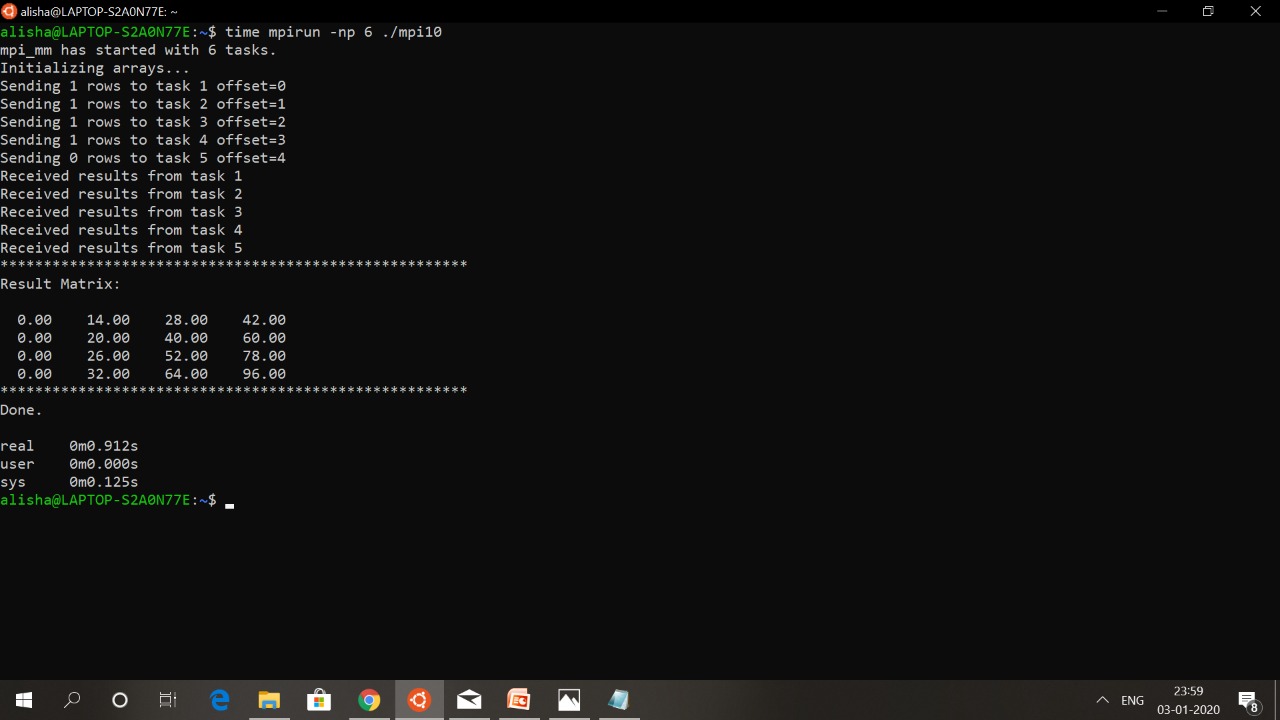
MPI\_Finalize();

}

Output 4 nodes:-

Output 5 nodes:-

Output 6 nodes:-



**6. Write a MPI program that take data(name or number), send that to all the processor and print them.**

#include<stdio.h>

#include<string.h>

#include<mpi.h>

int main() {

int comm\_sz;

int my\_rank;

char name[100];

int num;

MPI\_Init(NULL, NULL);

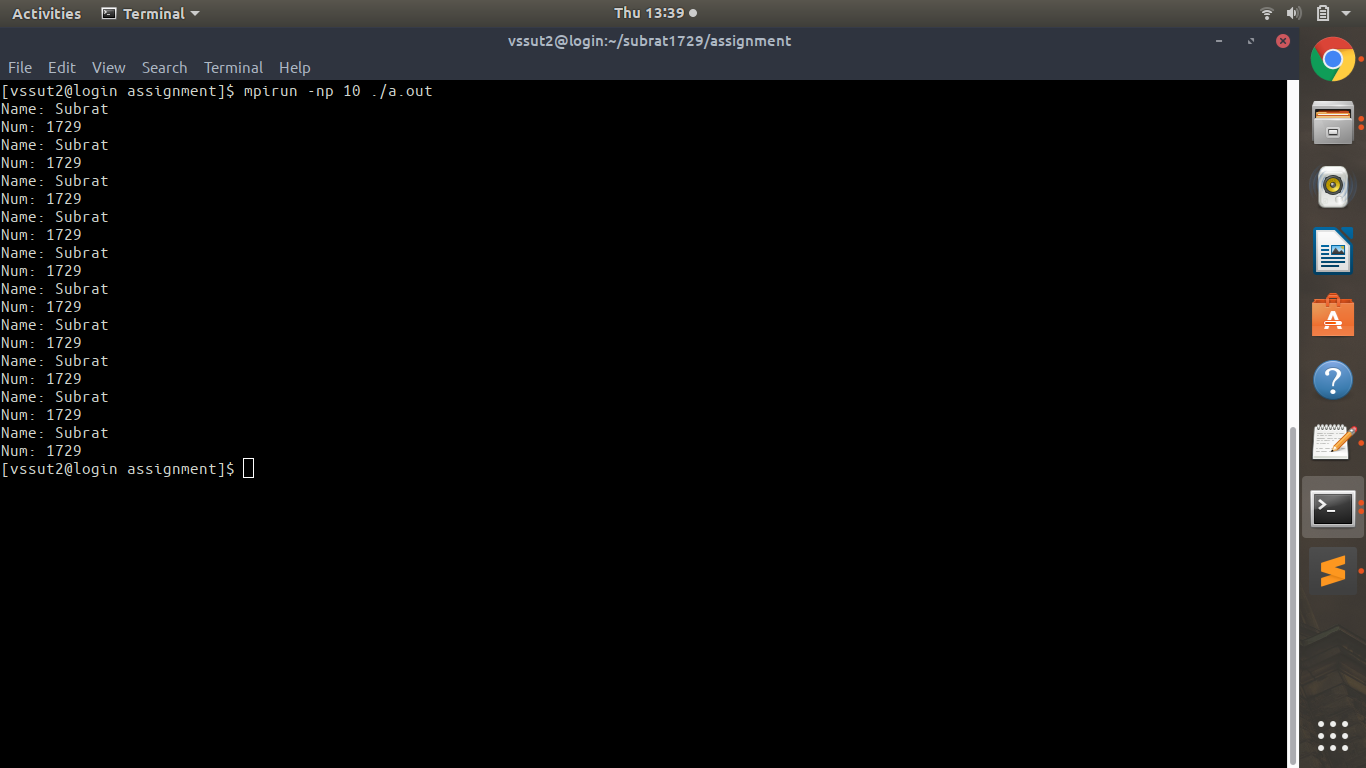
MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); if(my\_rank==0) {

strcpy(name, "Subrat"); num = 1729;

}

MPI\_Bcast(name, 100, MPI\_CHAR, 0, MPI\_COMM\_WORLD); MPI\_Bcast(&num, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); printf("Name: %s\nNum: %d\n", name, num); MPI\_Finalize();

}



**7. Write a MPI program that should return the sum of all processes involved note:-reduce.**

#include<stdio.h>

#include<string.h>

#include<mpi.h>

#define n 10

int main() {

int comm\_sz;

int my\_rank;

int sum = 0, num[100], a[1000]; MPI\_Init(NULL, NULL);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); if(my\_rank==0) {

printf("Enter the numbers:\n"); for (int i = 0; i < n; ++i) scanf("%d", &a[i]);

}

//MPI\_Barrier(MPI\_COMM\_WORLD); MPI\_Scatter(a, n/comm\_sz, MPI\_INT, num, n/comm\_sz, MPI\_INT, 0, MPI\_COMM\_WORLD);

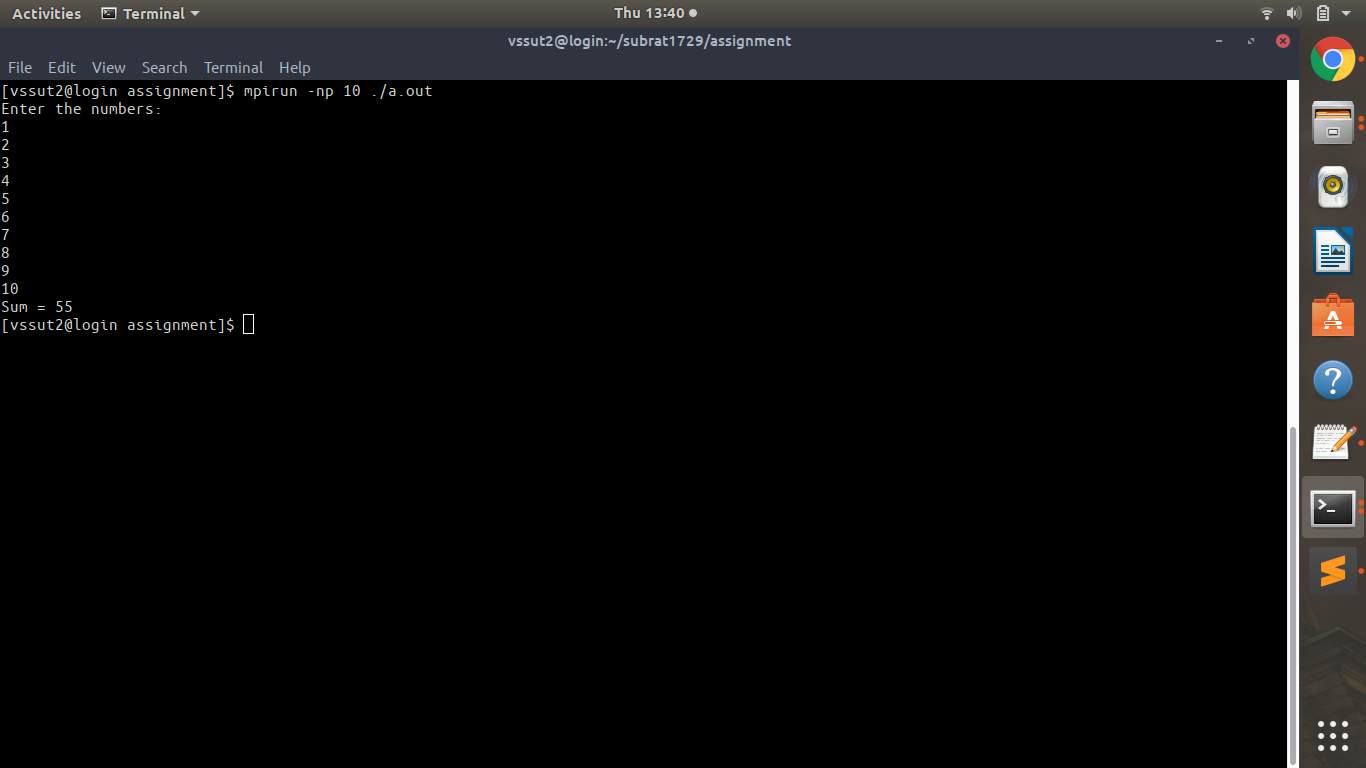
MPI\_Reduce(num, &sum, 1, MPI\_INT, MPI\_SUM, 0, MPI\_COMM\_WORLD); if(my\_rank==0) {

printf("Sum = %d\n", sum);

}

MPI\_Finalize();

}



**8. Write a MPI program that should return the sum of all processes involved note:-reduceall.**

#include<stdio.h>

#include<string.h>

#include<mpi.h>

#define n 10

int main() {

int comm\_sz;

int my\_rank;

int sum = 0, num[100], a[1000]; MPI\_Init(NULL, NULL);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); if(my\_rank==0) {

printf("Enter the numbers:\n"); for (int i = 0; i < n; ++i) scanf("%d", &a[i]);

}

//MPI\_Barrier(MPI\_COMM\_WORLD); MPI\_Scatter(a, n/comm\_sz, MPI\_INT, num, n/comm\_sz, MPI\_INT, 0, MPI\_COMM\_WORLD);

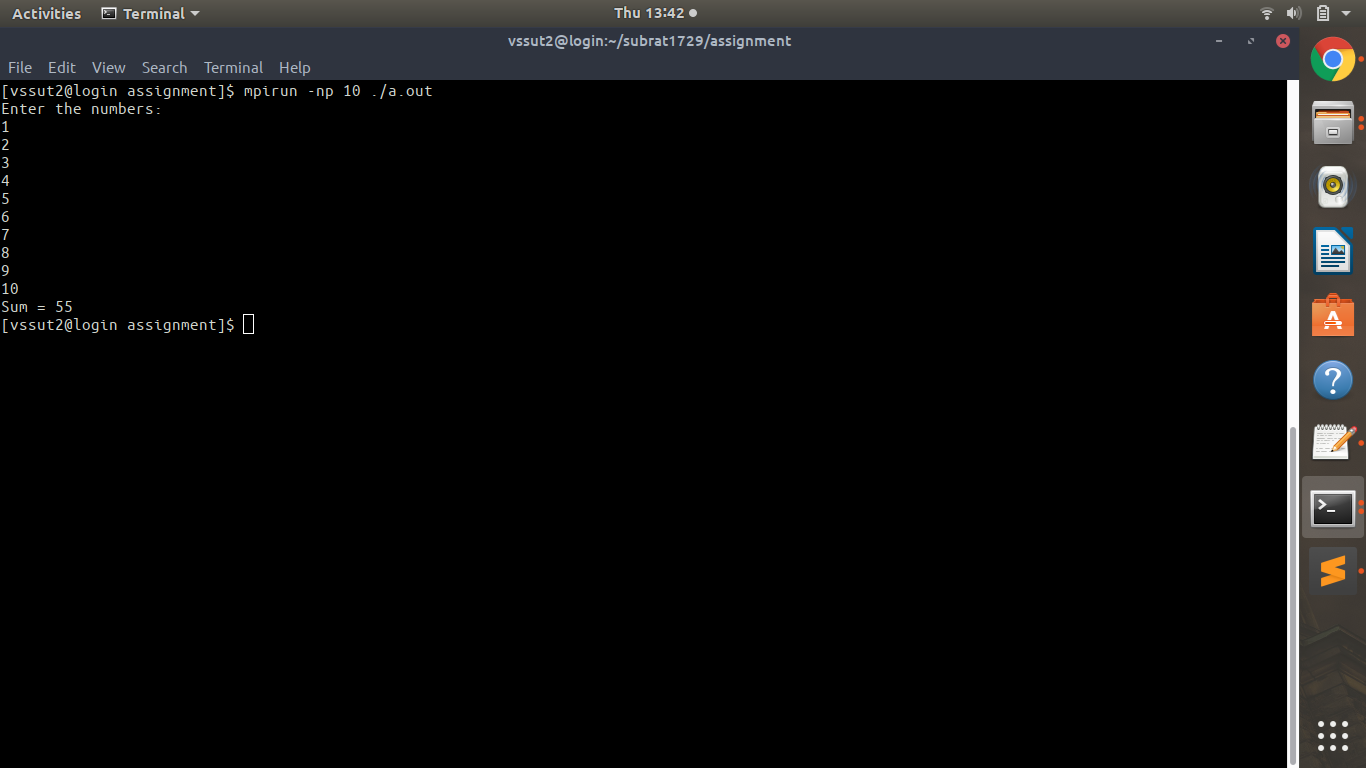
MPI\_Allreduce(num, &sum, 1, MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD); if(my\_rank==0) {

printf("Sum = %d\n", sum);

}

MPI\_Finalize();

}



**9. WAP such that it should initilize an array of 1 - 25 and divide these values among 5 process equally note:- use 2d array.**

#include<stdio.h>

#include<string.h>

#include<mpi.h>

#define n 10

int main() {

int comm\_sz;

int my\_rank;

int a[5][5], num[5];

MPI\_Init(NULL, NULL);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); if(my\_rank==0) {

printf("Enter the numbers:\n"); for (int i = 0; i < 5; i++) {

for(int j=0; j<5; j++) {

scanf("%d", &a[i][j]);

}

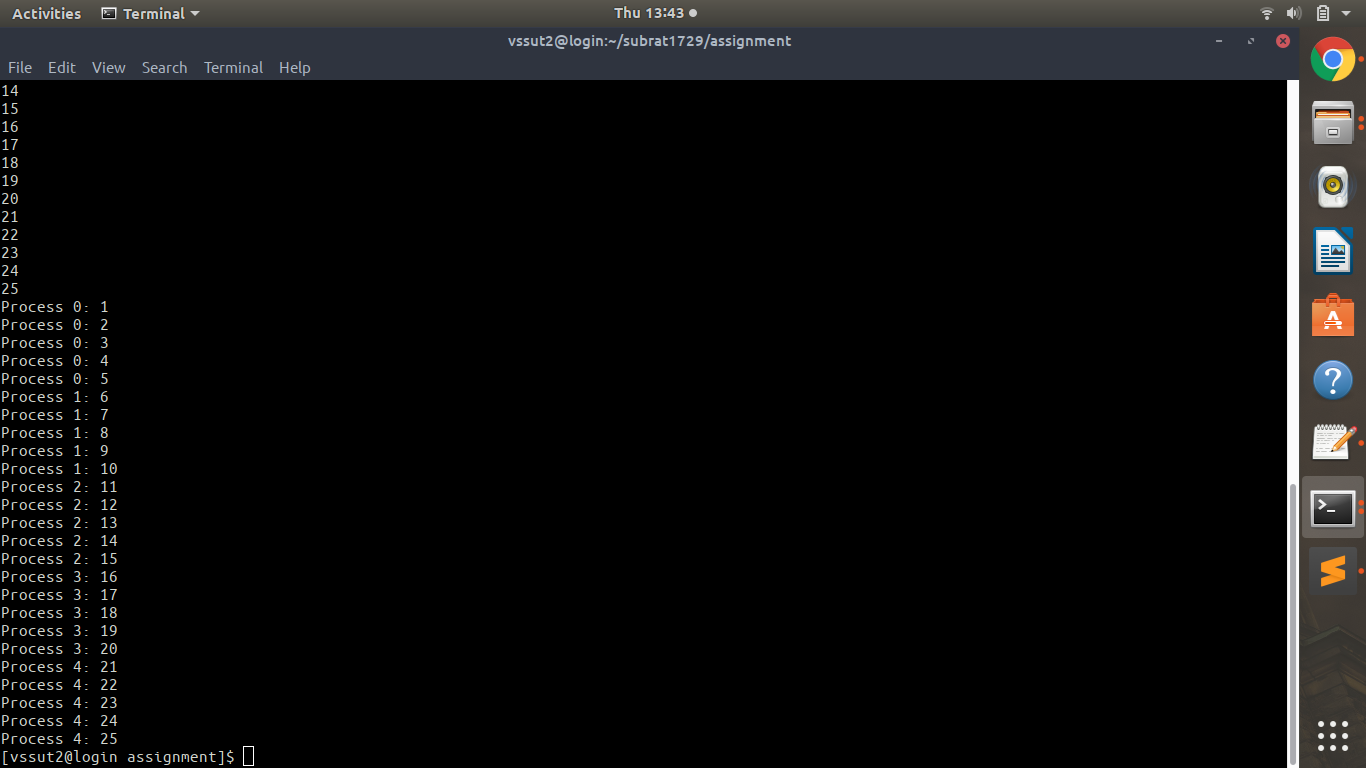
}

}

//MPI\_Barrier(MPI\_COMM\_WORLD); MPI\_Scatter(a, 5, MPI\_INT, num, 5, MPI\_INT, 0, MPI\_COMM\_WORLD); for(int i=0; i<5; i++)

printf("Process %d: %d\n", my\_rank, num[i]); MPI\_Finalize();

}



**10. WAP to decomposition simple data such that the master task should 1st initialize an array and then distribute an equal portion of that array to the other tasks. After the other task receive their portion of data, they should perform an ADDITION Operation on elements of array.**

#include<stdio.h>

#include<string.h>

#include<mpi.h>

#define n 10

int main() {

int comm\_sz;

int my\_rank;

int sum = 0, num[100], a[1000]; MPI\_Init(NULL, NULL);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank); if(my\_rank==0) {

printf("Enter the numbers:\n"); for (int i = 0; i < n; ++i) scanf("%d", &a[i]);

}

MPI\_Scatter(a, n/comm\_sz, MPI\_INT, num, n/comm\_sz, MPI\_INT, 0, MPI\_COMM\_WORLD);

for(int i=0; i<n/comm\_sz; i++) sum += num[i];

int allsum[comm\_sz];

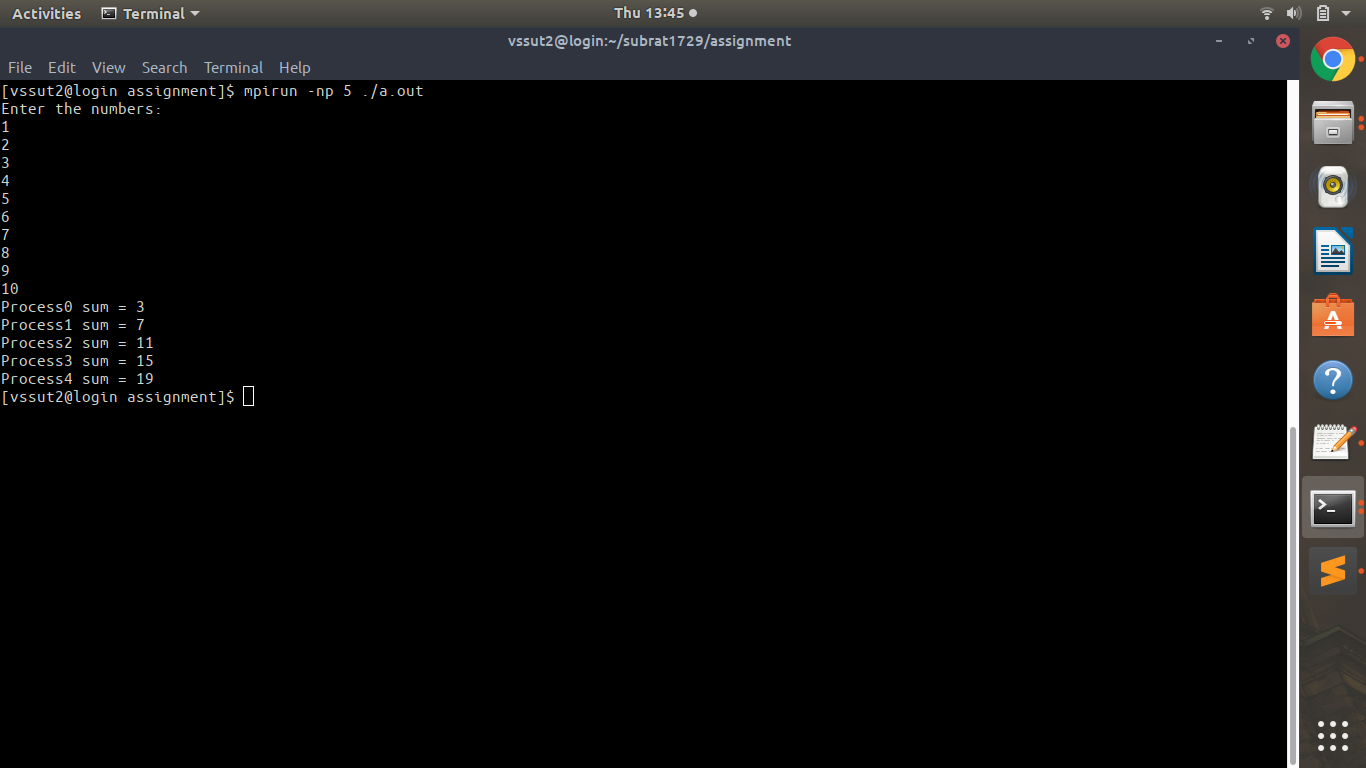
MPI\_Gather(&sum, 1, MPI\_INT, allsum, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); if(my\_rank==0) {

for(int i=0; i<comm\_sz; i++) printf("Process%d sum = %d\n", i, allsum[i]);

}

MPI\_Finalize();

}

****

**Write a program to generate all the permutations given the number of characters to be taken and the size of the string.**

1. **Implementation in serial program:-**

#include<iostream>

#include <omp.h>

using namespace std;

int main(int argc, char const \*argv[])

{

int n, i, j, k;

char ch[36], t = 'a';

for(i = 0; i<36; i++){

ch[i] = t;

t++;

if(t == '{')

t = '0';

}

cout<<"Enter the numbers of characters:- ";

cin>>n;

double before = omp\_get\_wtime();

for(i = 0; i < n\*n\*n; i++)

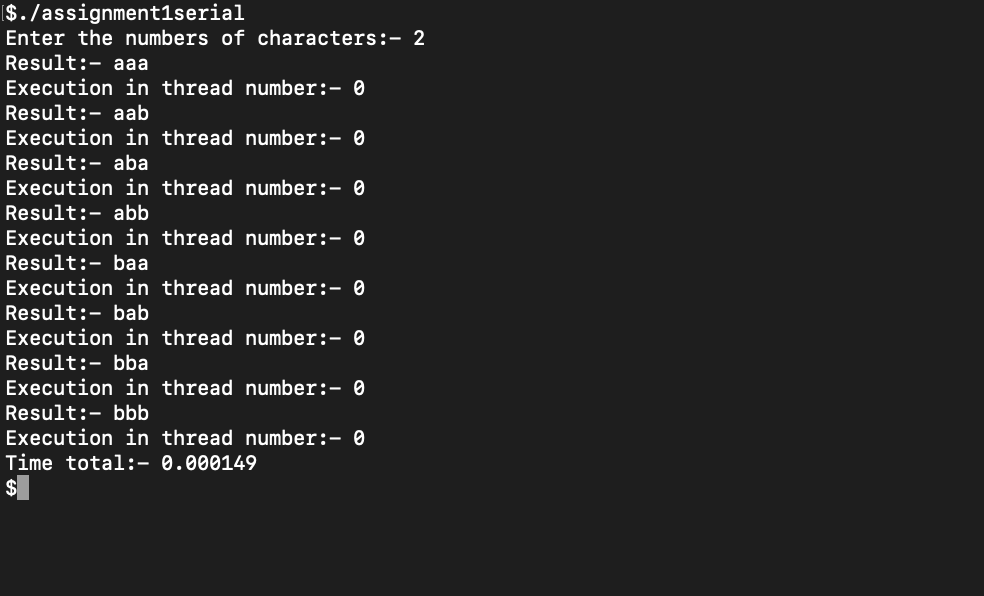
printf("Result:- %c%c%c\nExecution in thread number:- %d\n", ch[(i/(n\*n))%n], ch[(i/n)%n], ch[i%n], omp\_get\_thread\_num());

double after = omp\_get\_wtime();

cout<<"Time total:- "<<after - before<<endl;

return 0;

}

**2.Implementation in OpenMP:-**

#include<iostream>

#include <omp.h>

using namespace std;

int main(int argc, char const \*argv[])

{

int n, i, j, k;

char ch[36], t = 'a';

for(i = 0; i<36; i++){

ch[i] = t;

t++;

if(t == '{')

t = '0';

}

cout<<"Enter the numbers of characters:- ";

cin>>n;

double before = omp\_get\_wtime();

#pragma omp parallel for num\_threads(32)

for(i = 0; i < n\*n\*n; i++)

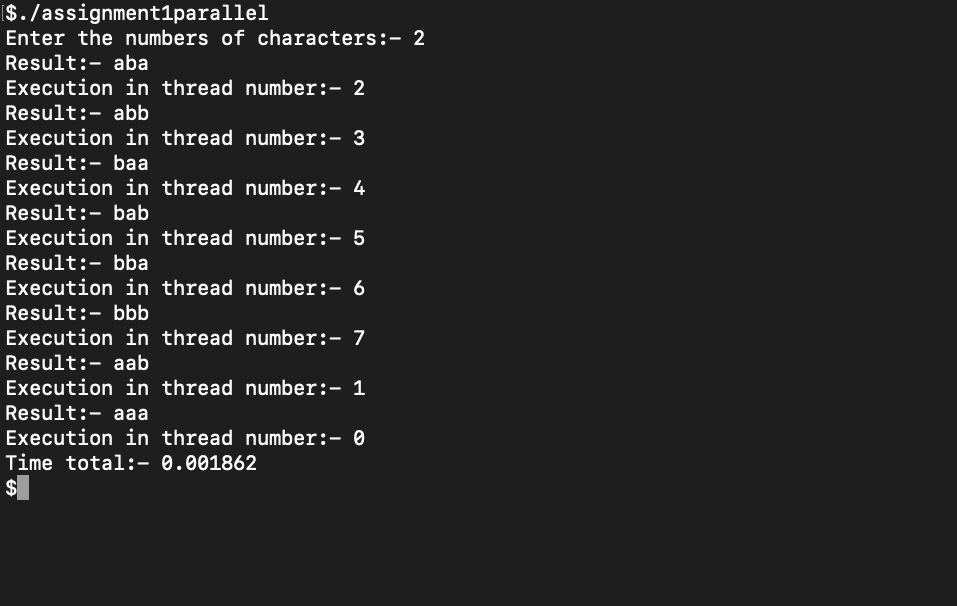
printf("Result:- %c%c%c\nExecution in thread number:- %d\n", ch[(i/(n\*n))%n], ch[(i/n)%n], ch[i%n], omp\_get\_thread\_num());

double after = omp\_get\_wtime();

cout<<"Time total:- "<<after - before<<endl;

return 0;

}



1. **Implementation in MPI:-**

#include<iostream>

#include<omp.h>

#include<mpi.h>

#define MASTER 0

#define MAX\_STRING 1000

#define send\_tag 2001

#define return\_tag 2002

using namespace std;

int main(int argc, char const \*argv[])

{

int n, i, j, k, id, sender;

char ch[36], t = 'a';

MPI\_Status status;

for(i = 0; i<36; i++){

ch[i] = t;

t++;

if(t == '{')

t = '0';

}

cout<<"Enter the numbers of characters:- ";

cin>>n;

int comm\_sz;

int my\_rank, div, end, start, to\_send, to\_recieve, ierr, partial\_ans;

char greeting[MAX\_STRING];

MPI\_Init(NULL, NULL);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &comm\_sz);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &my\_rank);

if(my\_rank != MASTER){

div = (n\*n\*n) / comm\_sz;

for(id=1; id<comm\_sz; id++){

start = i\*div + 1;

end = (i+1)\*div;

if((n-end) < comm\_sz)

end = n-1;

to\_send = end - start + 1;

}

for(i=0; i < div; i++)

sprintf(greeting, "Result:- %c%c%c\nExecution in thread number:- %d\n", ch[(i/(n\*n))%n], ch[(i/n)%n], ch[i%n], omp\_get\_thread\_num());

MPI\_Send(greeting, strlen(greeting)+1, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD);

}

else{

printf("Result:- %c%c%c\nExecution in thread number:- %d\n", ch[(i/(n\*n))%n], ch[(i/n)%n], ch[i%n], omp\_get\_thread\_num());

for(int q=1; q < div; q++){

MPI\_Recv(greeting, MAX\_STRING, MPI\_CHAR, q, 0, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

printf("%s \n", greeting);

}

}

return 0;

}

**4.VTune, ITAC and Quantum Espresso**

**Intel® VTune™**

Intel VTune is an application created by Intel for software performance analysis of serial and multithreaded applications on 32 and 64-bit x86 based machines. VTune Profiler helps analyse the algorithm choices and identify where and how your application can benefit from available hardware resources.

**Code optimization**

VTune assists in various kinds of code profiling including stack sampling, thread profiling and hardware event sampling. The profiler result consists of details such as time spent in each sub routine which can be drilled down to the instruction level. The time taken by the instructions are indicative of any stalls in the pipeline during instruction execution. The tool can be also used to analyse thread and storage performance.

**Intel® Trace Analyzer and Collector**

Intel Trace Collector is a tool for tracing MPI applications. It intercepts all MPI calls and generates tracefiles that can be analysed with Intel Trace Analyzer for understanding the application behaviour. Intel® Trace Collector can also trace non-MPI applications, like socket communication in distributed applications or serial programs.

**Tracing**

In software engineering, tracing essentially is a specialized form of logging to record information about the execution of a program at runtime. This information is typically used by programmers for debugging purposes, and additionally, depending on the type and detail of information contained in a trace log, by experienced system administrators or technical-support personnel and by software monitoring tools to diagnose common problems with software.

**Quantum ESPRESSO**

Quantum ESPRESSO is a suite for electronic-structure calculation and materials modelling at the nanoscale, distributed free software. It is based on density-functional theory, plane wave basis sets, and pseudopotentials. ESPRESSO is an acronym for open-Source Package for Research in Electronic Structure, Simulation, and Optimization.