JUNAID GIRKAR | 60004190057 | BE COMPS A2 | HPC | EXP 3

EXPERIMENT 3

AIM: Implement a parallel program to demonstrate the cube of N number within a set range.

THEORY:
The term parallel programming may be used interchangeable with parallel processing or in conjunction with parallel computing, which refers to the systems that enable the high efficiency of parallel programming.

In parallel programming, tasks are parallelized so that they can be run at the same time by using multiple computers or multiple cores within a CPU. Parallel programming is critical for large scale projects in which speed and accuracy are needed. It is a complex task, but allows developers, researchers, and users to accomplish research and analysis quicker than with a program that can only process one task at a time.

Parallel programming works by assigning tasks to different nodes or cores. In High Performance Computing (HPC) systems, a node is a self-contained unit of a computer system contains memory and processors running an operating system. Processors, such as central processing units (CPUs) and graphics processing units (GPUs), are chips that contain a set of cores. Cores are the units executing commands; there can be multiple cores in a processor and multiple processors in a node.

With parallel programming, a developer writes code with specialized software to make it easy for them to run their program across on multiple nodes or processors. A simple example of where parallel programming could be used to speed up processing is recoloring an image. A developer writes the code to break up the overall task of to change the individual aspects of an image by segmenting the image into equal parts and then assigns the recoloring of each part to a different parallel task, each running on their own compute resources. Once the parallel tasks have completed, the full image is reassembled.

Parallel processing techniques can be utilized on devices ranging from embedded, mobile, laptops, and workstations to the world's largest supercomputers. Different computer languages provide various technologies to enable parallelism. For C, C++ and Fortran, OpenMP, open multi-processing, provides a cross-platform API for developing parallel applications that enable running parallel tasks across cores of a CPU. When processes need to communicate between different computers or nodes, a technology such as MPI, message passing interface, is typically used. There are benefits to both models. Multiple cores on a single node share memory. Shared memory is typically faster for exchanging information than message passing between nodes over a network. However, there's a limit to how many cores a single node can have. As projects get larger, developers may use both types of parallelism together. One of the challenges that developers face though is properly decomposing their algorithm and parallelizing across multiple nodes and multiple cores for maximum performance and debugging their parallel application when it does not work correctly.

Parallel Programming Usage:
• Advanced graphics in the entertainment industry
• Applied physics
• Climate research
• Electrical engineering
• Financial and economic modeling
• Molecular modeling
• National defense and nuclear weaponry
• Oil and gas exploration
• Quantum mechanics

CODE:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int n = 20;
int a2[1000];
int tmp2;
int tmp;
int cntr;
int nR;
int prnt_cntr = 0;
int main(int argc, char* argv[]) {
        int pid, np, elements_per_process, n_elements_received; MPI_Status status;
        int index, i;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &pid);
        MPI_Comm_size(MPI_COMM_WORLD, &np);
        elements_per_process = n / np;
        if (pid == 0) {
                if (np > 1) {
                        for (i = 1; i < np - 1; i++) {
                                index = i * elements_per_process;
                                MPI_Send(&elements_per_process, 1, MPI_INT, i, 0,
                                        MPI_COMM_WORLD);
                                MPI_Send(&index, 1, MPI_INT, i, 0,
MPI_COMM_WORLD);
                        }
                        index = i * elements_per_process;
                        int elements_left = n - index;
                        MPI_Send(&elements_left, 1, MPI_INT, i, 0,
                                MPI_COMM_WORLD);
                        MPI_Send(&index, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
                        int cube = 0;
                        for (i = 0; i < elements_per_process; i++) {
                                cube = (i) * (i) * (i);
                                printf("%d^3=%d \n", prnt_cntr, cube);
                                prnt_cntr += 1;
                        }
                        for (i = 1; i < np; i++) {
                                MPI_Recv(&cntr, 1, MPI_INT, i, 0, MPI_COMM_WORLD,
                                        &status);
                                MPI_Recv(&a2, cntr, MPI_INT, i, 0, MPI_COMM_WORLD,
                                        &status);
                                int sender = status.MPI_SOURCE;
                                for (int j = 0; j < cntr; j++) {
                                        printf("%d^3 = %d \n", prnt_cntr, a2[j]);
                                        prnt_cntr += 1;
                                        }
                              }
                    }
          }
          else {
                    MPI_Recv(&n_elements_received, 1, MPI_INT, 0, 0,
                              MPI_COMM_WORLD, &status);
                    MPI_Recv(&tmp2, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
                    int partial_sum = 0;
                    for (int i = 0; i < n_elements_received; i++) {
                              a2[i] = (tmp2 + i) * (tmp2 + i) * (tmp2 + i);
                              nR = n_elements_received;
                    }
                    MPI_Send(&nR, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
                    MPI_Send(&a2, nR, MPI_INT, 0, 0, MPI_COMM_WORLD);
          }
          MPI_Finalize();
          return 0;
}
```

OUTPUT:

```
C:\Users\jarvis\source\repos\Exp3\x64\Debug>mpiexec —n 4 Experiment3.exe
0^3=0
1^3=1
2^3=8
3^3=27
4^3=64
5^3=125
6^3 = 216
7^3=343
8^3=512
9^3=729
10^3=1000
11^3=1331
12^3=1728
```

```
13^3=2197
14^3=2744
15^3=3375
16^3=4096
17^3=4913
18^3=5832
19^3=6859
```

CONCLUSION: MPI was used to teach the concept of parallel programming. A parallel program above showed how the program can be run in a parallel environment and compute the cube for a given set of values. Hence, parallelism improves the valuable time consumption and computes the results faster than any other way of computation.