Experiment :3

//Aim :. Develop a distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP. Demonstrate by displaying the intermediate sums calculated at different processors.

theory:

**Open MP**

The provided code demonstrates the usage of OpenMP (Open Multi-Processing) to parallelize the computation of summing elements in an array. Let's explain the code and its relation to OpenMP and distributed systems:

Import Statements:

The code includes the necessary headers for standard input/output operations (<stdio.h>) and OpenMP (<omp.h>).

Variables and Array Initialization:

The code defines variables N and n to represent the total number of elements in the array and the number of processors, respectively.

An array arr is initialized with 10 elements.

Chunk Calculation:

The variable chunk represents the number of elements that each processor will handle. It is calculated by dividing the total number of elements (N) by the number of processors (n).

Print Array:

The code contains a loop to print the elements of the array.

Parallel Region and Thread-Specific Computation:

The #pragma omp parallel directive creates a parallel region, instructing the compiler to parallelize the following block of code.

The num\_threads(n) clause specifies that the parallel region should be executed using n threads.

Thread-Specific Operations:

Inside the parallel region, each thread retrieves its unique thread ID using omp\_get\_thread\_num().

Based on the thread ID, each thread calculates its start and end indices to iterate over the assigned chunk of elements in the array.

Each thread calculates a local sum by iterating over its assigned chunk of elements in the array.

Print Intermediate Sum:

Each thread prints its thread ID and the intermediate sum it calculated.

Critical Section and Final Sum Calculation:

The #pragma omp critical directive defines a critical section, ensuring that only one thread can execute the block of code within it at a time.

Inside the critical section, each thread adds its local sum to the shared sum variable, updating the final sum.

Print Final Sum:

After the parallel region, the code prints the final sum, which is the accumulated sum of all the intermediate sums calculated by the threads.

OpenMP is a programming model that enables shared-memory multiprocessing, allowing developers to parallelize code execution across multiple threads. In this code, OpenMP is used to distribute the computation of summing array elements among multiple threads. Each thread calculates a local sum for a subset of the array, and then the local sums are combined into a final sum using a critical section to avoid race conditions.

While OpenMP itself is primarily designed for shared-memory systems, the concepts of parallelism and distributed computation can be related in the sense that they both aim to improve performance and utilize multiple processing resources. However, OpenMP is not specifically designed for distributed systems where multiple nodes communicate over a network. For distributed systems, other parallel programming models, such as MPI (Message Passing Interface), are typically used to facilitate communication and coordination among distributed nodes.

OpenMP (Open Multi-Processing) and MPI (Message Passing Interface) are both programming models used for parallel computing, but they have different approaches and purposes. Here's a brief explanation of the differences between OpenMP and MPI:

1. Programming Paradigm:
   * OpenMP: OpenMP follows the shared-memory programming paradigm, where multiple threads of execution share a common memory space.
   * MPI: MPI follows the message passing programming paradigm, where multiple processes communicate by explicitly sending and receiving messages.
2. Usage:
   * OpenMP: OpenMP is typically used for parallelizing shared-memory applications on a single machine or a shared-memory system. It provides directives and API calls that allow developers to specify parallel regions and distribute work among threads.
   * MPI: MPI is designed for distributed memory systems, where multiple processes run on separate machines or nodes. It enables communication and coordination between processes by explicitly sending messages.
3. Data Sharing:
   * OpenMP: In OpenMP, threads have shared memory, meaning they can access and modify shared data directly without explicitly sending or receiving messages.
   * MPI: In MPI, each process has its own memory space, and data must be explicitly passed between processes using send and receive operations.
4. Parallelism Model:
   * OpenMP: OpenMP is best suited for parallelizing loop iterations and tasks within a single program. It focuses on exploiting thread-level parallelism within a shared-memory environment.
   * MPI: MPI allows for both task-level parallelism and data parallelism. It enables the distribution of work across multiple processes and facilitates communication and synchronization between them.

Implementation:

#include <stdio.h>

#include <omp.h>

int main() {

int N = 10; // Total number of elements in the array

int n = 4; // Number of processors

int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // Input array

int chunk = N / n; // Number of elements each processor will handle

int sum = 0; // Final sum of all elements

for (int i = 0; i < 10; ++i)

{

printf("%d,",arr[i]) ;

}

printf("\n");

#pragma omp parallel num\_threads(n)

{

int thread\_id = omp\_get\_thread\_num();

int start = thread\_id \* chunk;

int end = start + chunk;

int local\_sum = 0; // Intermediate sum calculated by each processor

// Calculate local sum

for (int i = start; i < end; i++) {

local\_sum += arr[i];

}

printf("Thread %d: Intermediate sum = %d\n", thread\_id, local\_sum);

#pragma omp critical

{

sum += local\_sum; // Add local sum to the final sum

}

}

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

return 0;

}

Output:

manas@LAPTOP-4VJQF5TU:/mnt/c/Users/manas/Downloads/ASS$ **g++ ASS7.c -o ASS7.exe -fopenmp**

manas@LAPTOP-4VJQF5TU:/mnt/c/Users/manas/Downloads/ASS$ **./ASS7.exe**

1,2,3,4,5,6,7,8,9,10,

Thread 0: Intermediate sum = 3

Thread 3: Intermediate sum = 15

Thread 2: Intermediate sum = 11

Thread 1: Intermediate sum = 7

Final sum = 36

Do not use this 👎

**Installation of OPENMPI**

1. Download openmpi-4.1.4.tar.bz2 from http://www.open-mpi.org

2. Goto the terminal (Command prompt)

3. update using

sudo apt-get update

sudo apt install gcc {if not already installed}

4. Goto the directory which contains the downloaded file (download folder)

5. Extract the files using (download folder)

tar -jxf openmpi-4.1.4.tar.bz2

6. The directory openmpi-4.1.4 is created

7. Go into the source directory through terminal

Command prompt > cd /home/it/download/openmpi-1.4.4

7. Configure, compile and install by executing the following commands

Command prompt//home/it/download/openmpi-1.4.4 > ./configure --prefix=$HOME/opt/openmpi

make all

make install

8. Now openmpi folder is created in ‘opt‘ folder of Home directory.

9. Now the folder LP5 can be deleted (optional)

10. Update the PATH and LD\_LIBRARY\_PATH environment variable using

echo "export PATH=\$PATH:\$HOME/it/opt/openmpi/bin" >> $HOME/it.bashrc

echo "export LD\_LIBRARY\_PATH=\$LD\_LIBRARY\_PATH:\$HOME/it/opt/openmpi/lib">>$HOME/it.bashrc

11. Compile the program using

mpicc name of the program

example : mpicc hello1.c

12. Execute the program using

mpirun -np N ./a.out

example : mpirun -np 2 ./a.out

example : mpirun -np 3 ./a.out

example : mpirun -np 4 ./a.out

/\*\*\*\*\*\* The cores 2 and will be in waiting mode … Press Ctrl+z to end the execution \*\*\*\*\*\*\*/

Assignment program: Add 20 numbers in an array using 4 cores

C:\>cd /opt/openmpi/bin

C:\opt/openmpi/bin> edit program\_name.c

Example :

C:\opt/openmpi/bin> edit add.c

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <stdio.h>

#include "mpi.h"

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

{

int rank, size;

int num[20]; //N=20, n=4

MPI\_Init(&argc, &argv);

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

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

for(int i=0;i<20;i++)

num[i]=i+1;

if(rank == 0){

int s[4];

printf("Distribution at rank %d \n", rank);

for(int i=1;i<4;i++)

MPI\_Send(&num[i\*5], 5, MPI\_INT, i, 1, MPI\_COMM\_WORLD); //N/n i.e. 20/4=5

int sum=0, local\_sum=0;

for(int i=0;i<5;i++)

{

local\_sum=local\_sum+num[i];

}

for(int i=1;i<4;i++)

{

MPI\_Recv(&s[i], 1, MPI\_INT, i, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

}

printf("local sum at rank %d is %d\n", rank,local\_sum);

sum=local\_sum;

for(int i=1;i<4;i++)

sum=sum+s[i];

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

}

else

{

int k[5];

MPI\_Recv(k, 5, MPI\_INT, 0, 1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);

int local\_sum=0;

for(int i=0;i<5;i++)

{

local\_sum=local\_sum+k[i];

}

printf("local sum at rank %d is %d\n", rank, local\_sum);

MPI\_Send(&local\_sum, 1, MPI\_INT, 0, 1, MPI\_COMM\_WORLD);

}

MPI\_Finalize();

return 0;

}

Distribution at rank 0

local sum at rank 1 is 40

local sum at rank 2 is 65

local sum at rank 3 is 90

local sum at rank 0 is 15

final sum = 210

/\*\*\*\*\*\* students can be asked to take dynamic values for N, n and array \*\*\*\*\*\*\*\*\*\*\*\*/

**Run : Compile the program using**

C:mpicc name of the program

example : opt/openmpi/bin> mpicc add.c

12. Execute the program using

opt/openmpi/bin> mpirun -np N ./a.out

example : opt/openmpi/bin> mpirun -np 2 ./a.out

example : opt/openmpi/bin> mpirun -np 3 ./a.out

example : opt/openmpi/bin> mpirun -np 4 ./a.out

