Dinesh Pamu

B2- 30

ASSIGNMENT NO: 3

Title:

Distributed system, to find sum of N elements in an array by distributing N/n elements to n number of processors MPI or OpenMP.

Problem Statement:

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.

Software and tools:

1. MPI or OpenMP
2. Code Editor
3. C or C++ Compiler
4. Message Passing Interface (MPI) implementation
5. Open Multi-Processing (OpenMP) implementation
6. MPI or OpenMP Libraries
7. Display tool

Theory:

Introduction to Parallel Programming: Parallel programming is a programming paradigm that involves executing multiple tasks simultaneously to improve performance and increase efficiency. Parallel programming is widely used in scientific computing, big data analysis, and machine learning.

Introduction to MPI and OpenMP:

MPI and OpenMP are two parallel computing technologies that are widely used for developing applications that can run on parallel computing architectures.

MPI, which stands for Message Passing Interface, is a parallel computing standard that defines a set of functions for sending and receiving messages between processes in a distributed system. MPI is commonly used for developing high-performance scientific and engineering applications, such as numerical simulations and data analysis.

In MPI, a program is typically divided into a number of separate processes, each of which can run on a separate computing node. These processes communicate with each other through message passing, sending data back and forth as needed to perform the required computations. MPI provides a standardized interface for message passing, allowing developers to write parallel code that can run on a variety of hardware platforms and operating systems.

OpenMP, on the other hand, is a parallel programming model that is designed to simplify the development of shared-memory parallel programs. OpenMP allows developers to add parallelism to existing code by inserting directives into the code that specify how the work should be divided among multiple threads.

In OpenMP, a program runs on a single machine and divides its work among multiple threads running on multiple cores or processors. The threads share memory and communicate with each other using shared variables. OpenMP provides a set of standardized directives that allow developers to control the behavior of the threads, such as how the work is divided, how data is shared, and how synchronization is handled.

Both MPI and OpenMP are widely used for developing high-performance parallel applications, and each technology has its own strengths and weaknesses. MPI is generally considered to be more suitable for developing applications that run on distributed systems, while OpenMP is more suitable for developing applications that run on shared-memory systems. The choice of which technology to use depends on the specific requirements of the application and the characteristics of the computing architecture on which it will run.

Distributed System Development to Find the Sum of N Elements in an Array: To develop a distributed system to find the sum of N elements in an array by distributing N/n elements to n number of processors using MPI or OpenMP and display the intermediate sums calculated at different processors, the following steps can be followed:

1. Initialize the array: Initialize an array of N elements.
2. Distribute the work: Distribute the work of finding the sum of N elements in the array by distributing N/n elements to n number of processors.
3. Calculate the sum: Each processor calculates the sum of the elements assigned to it.
4. Send the intermediate sums: Each processor sends its intermediate sum to the master processor.
5. Calculate the final sum: The master processor calculates the final sum by adding the intermediate sums received from all the processors.
6. Display the intermediate sums: Display the intermediate sums calculated at different processors.

Algorithm:

1. Initialize variables N, n, and sum to 0.

2. Create an array of size N and initialize it with random values.

3. Divide the array into n subarrays of size N/n.

4. Spawn n threads and pass each thread a subarray to compute the sum.

5. In each thread, calculate the sum of the subarray.

6. Use a lock to update the shared variable sum with the local sum computed in each thread.

7. Join all threads and print the final sum.

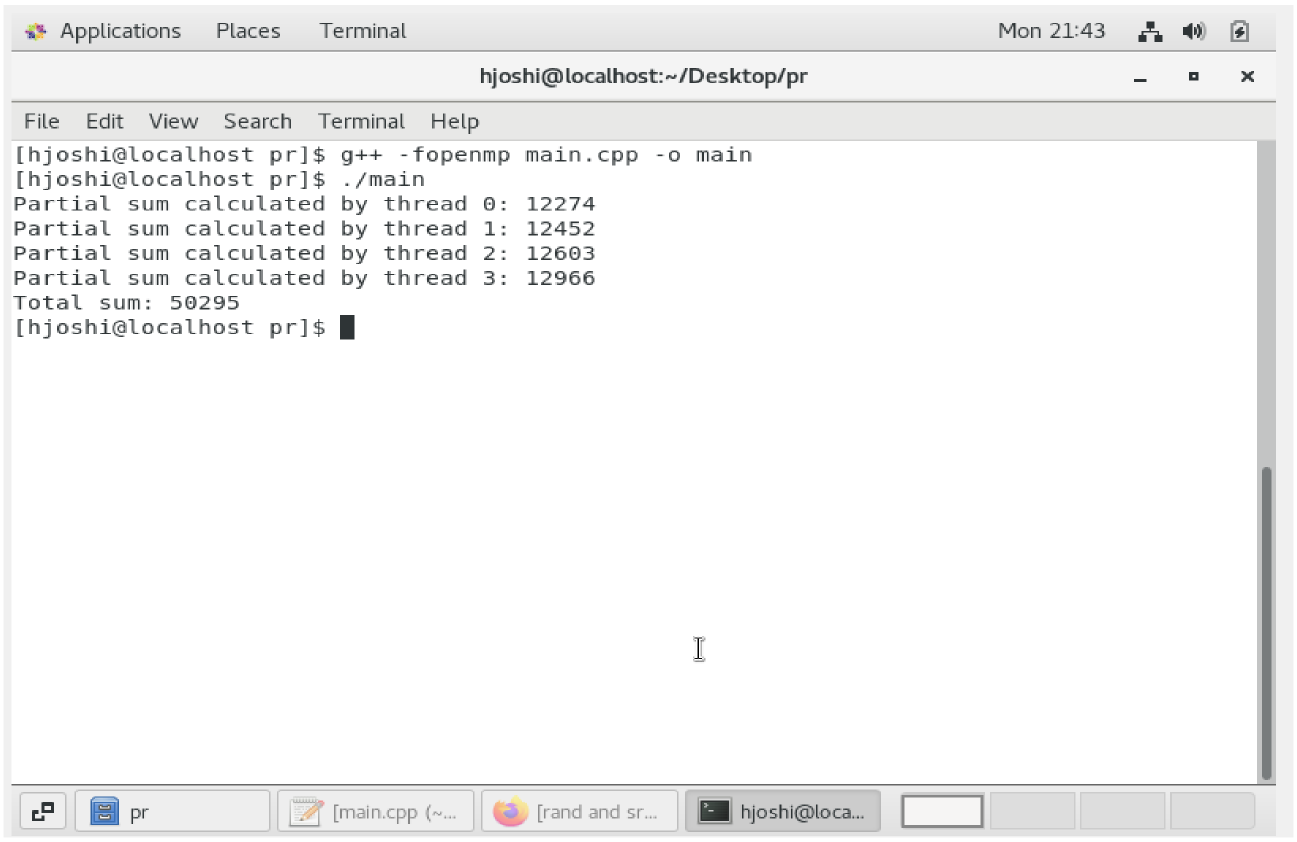
Conclusion:

The lab practical on developing a distributed system to find the sum of N elements in an array by distributing N/n elements to n number of processors using MPI or OpenMP and display the intermediate sums calculated at different processors is an excellent way to introduce students to the concepts of parallel programming, MPI or OpenMP, and distributed system development. By the end of this lab, students will have a clear understanding of how to develop a distributed system to find the sum of N elements in an array using MPI or OpenMP and display the intermediate sums calculated at different processors.

Code:

```
using namespace std;
#include<cstdlib>
#include <iostream>
#include <omp.h>
int main() {
  int N = 1000; // number of elements
  int num_threads = 4; // number of threads to use
  int sum = 0; // final sum
  int chunk_size = N / num_threads; // number of elements to be processed by each thread
  int partial_sums[num_threads]; // array to store partial sums calculated by each thread
  int arr[N]; // initialize array with random values
  for (int i = 0; i < N; i++) {
    arr[i] = rand() % 100;   }
  #pragma omp parallel num_threads(num_threads)
  {
    int thread_num = omp_get_thread_num();
    int start = thread_num * chunk_size; // start index for current thread
    int end = start + chunk_size; // end index for current thread
    int partial_sum = 0; // partial sum for current thread
    for (int i = start; i < end; i++) {
      partial_sum += arr[i];
    }
    partial_sums[thread_num] = partial_sum; }
  // calculate final sum by summing up partial sums calculated by each thread
  for (int i = 0; i < num_threads; i++) {
    sum += partial_sums[i];
    std::cout << "Partial sum calculated by thread " << i << ": " << partial_sums[i] << std::endl;
  }
  std::cout << "Total sum: " << sum << std::endl;
      return 0;
}
```

Output:



```
[hjoshi@localhost pr]$ g++ -fopenmp main.cpp -o main
[hjoshi@localhost pr]$ ./main
Partial sum calculated by thread 0: 12274
Partial sum calculated by thread 1: 12452
Partial sum calculated by thread 2: 12603
Partial sum calculated by thread 3: 12966
Total sum: 50295
[hjoshi@localhost pr]$
```