A picture containing text, clipart, vector graphics

Description automatically generated

**CSC580**

**PARALLEL PROCESSING**

**INDIVIDUAL ASSIGNMENT 1**

**PREPARED FOR:**

HASLIZATUL FAIRUZ BINTI MOHAMED HANUM

**GROUP:**

CS2305C

**PREPARED BY:**

MUHAMMAD ARIF FIKRI BIN MUSTAPA

(2020461922)

**Task 1:**

Describe the four variations of processor architecture as described by Flynn.

1. **Draw appropriate diagrams to support your answer.**



1. **Explain the components and functions of each architecture.**

Single Instruction, Single Data (SISD) :

The majority of today's SISD uniprocessor systems are pipelined. A SISD computer may contain several functional units, but they are all overseen by a single control unit. Computers using the SISD architecture can only process scalar instructions.

An SISD computing system is a uniprocessor machine capable of executing a single instruction, which operates on a single data stream. In SISD, machine instructions are processed sequentially. hence computers adopting this model are popularly called sequential computers (Buyya & Selvi, 2013).

Primary memory must store all of the instructions and data to be processed. The rate at which the computer can send information internally limits the processing element's speed in the SISD model. IBM PCs, Macintoshes, and workstations are the most common SISD systems (Buyya & Selvi, 2013).

Instructions are decoded by the Control Unit and then the Control Unit sends the instructions to the processing units for execution. Data Stream flows between the processors and memory bi-directionally.



Single Instruction, Multiple Data (SIMD) :

The control unit broadcasts the same instruction to all of the processing parts, which are ALUs, although they work on separate data sets from different data streams. The control unit, as shown in the diagram, sends a common instruction stream to each processing part. The shared memory sub-system, which includes several modules, is critical.

A SIMD operation can have a large number of elements, such as the 4 to 16 elements in short vector instructions, or a tiny number, such as hundreds, as in streaming vector processors. SIMD processors are also known as array processors because they have a shared controller and an array of functional components.



Multiple Instruction, Single Data (MISD) :

There are n processor elements in the MISD computer architecture. Each processor element is given its own set of instructions to carry out on the same data stream and its derivatives. The output of one processor unit becomes the input of the following processor element in the series in this case.



Multiple Instruction, Multiple Data (MIMD) :

MIMD computer category covers multiple computer system and multiprocessor systems. There are two types of MIMD computers which are tightly coupled or Uniform Memory Access (UMA) and loosely coupled or Non-Uniform Memory

Access (NUMA). MIMD computer is called tightly coupled or Uniform Memory Access (UMA) if the degree of interaction among the processor is high. MIMD computer is called loosely coupled or Non-Uniform Memory Access (NUMA) if the degree of interaction among processors is low.

The MIMD architecture consists of a group of N tightly connected processors. Each CPU has memory that is shared by all processors but cannot be accessed directly by other processors. The processors of the MIMD architecture work independently and asynchronously. Various processors may be performing various instructions on various pieces of data at any given time.



1. **Describe with an example the advantages of each architecture.**

Single Instruction, Single Data (SISD) :

The advantages of SISD is its requires less power because the sequential processor takes data from a single address in memory and performs a single instruction on the data. In other word, it is a single core computer. Von neumann single CPU computer is the example of computer with SISD architecture.

Single Instruction, Multiple Data (SIMD) :

SIMD very efficient when you need to perform the same instruction on large amounts of data. These instructions can be performed sequentially, taking advantage of pipelining, or in parallel using multiple processors. Pipeline computer is the example of SIMD architecture.

Multiple Instruction, Single Data (MISD) :

MISD is excellent for situation where fault tolerance is critical. Used on flight control systems where fault detection is critical.

Multiple Instruction, Multiple Data (MIMD) :

MIMD is ideal for scenarios requiring a wide range of processor and data intensive operations (such as video editing, game rendering). Multiple autonomous processors perform operations on difference pieces of data, either independently or as part of shared memory space. Modern laptops /desktop nowadays is an example of computer with MIMD architecture.

**Task 2:**

Describe the phases to transform an algorithm so that it can be executed in parallel machine.

1. **Identify one search or sort algorithm and describe the instructions.**

Quick Sort :

Quick sort is an extremely efficient sorting technique that divides a large data array into smaller ones. A huge array is divided into two arrays, one of which contains values smaller than the provided value, say pivot, on which the partition is based, and the other of which contains values greater than the pivot value. Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

Step 1 − Choose the highest index value has pivot.

Step 2 − Take two variables to point left and right of the list excluding pivot.

Step 3 − left points to the low index.

Step 4 − right points to the high.

Step 5 − while value at left is less than pivot move right.

Step 6 − while value at right is greater than pivot move left.

Step 7 − if both step 5 and step 6 does not match swap left and right.

Step 8 − if left ≥ right, the point where they met is new pivot.

1. **Draw diagram(s) to show the steps to transform the algorithm.**

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

1. **Discuss the performance of the parallel algorithm.**

The shared-address-space version of the quicksort algorithm must complete four stages given an array of n elements and p processes:

1. calculate and broadcast the pivot;
2. locally rearrange the array assigned to each process;
3. decide where the local elements will go in the globally rearranged array;
4. perform the global rearranging

Using an efficient recursive doubling technique for shared-address-space broadcast, the first step can be completed in time Q(log p). The classic quicksort technique for splitting around a pivot element can be used to complete the second phase in time Q(n/p). Using two prefix sum procedures, the third step can be completed in Q(log p). Finally, the fourth step, which needs us to replicate the local pieces to their eventual destination, can be completed in at most Q(n/p) time.

The complexity of the quicksort algorithm's shared-address-space formulation is determined by two factors. The first is the time it takes to split a given array into smaller-than-pivot and larger-than-pivot sub-arrays, and the second is the degree to which the various pivots used result in balanced partitions. To make our analysis easier, we'll assume that pivot selection always results in balanced divisions in this section.

1. **Discuss two issues that may affect the performance of the algorithm.**

We bypassed pivot selection in the parallel quicksort method. Pivot selection is exceptionally difficult, because it has a considerable impact on the algorithm's effectiveness. Consider the scenario in which the first pivot is the sequence's greatest member. In this example, one of the processes will be assigned only one element after the first split, while the remaining p - 1 processes will be assigned n - 1 elements.

As a result, we have a problem whose size has been reduced by one element, but the sorting procedure will only involve p - 1 processes. Despite the fact that this is a made-up example, it demonstrates a key issue with parallelizing the quicksort method. The split should ideally be done so that each segment contains a significant portion of the original array.

One technique to choose pivots is to do it at random, as seen below. During the (n)th split, one process from each of the process groups chooses one of its elements at random to serve as the pivot for this partition. This is similar to the sequential quicksort algorithm's random pivot selection. Despite the fact that this method appears to work for sequential quicksort, it is not well suited for parallel quicksort.

Consider the scenario in which a faulty pivot is chosen at some time. This results in a partitioning in which one subsequence is much larger than the other in sequential quicksort. If all subsequent pivot picks are good, one bad pivot will only add an amount equal to the length of the subsequence to the entire work.

As a result, the performance of sequential quicksort will not be greatly harmed. However, in the parallel formulation, a bad pivot can cause partitioning, causing a process to become idle, which will last throughout the algorithm's execution. A better pivot selection approach can be derived if the initial distribution of items in each process is uniform. The n/p elements that are originally stored at each process provide a representative sample of all n elements in this example.

**Task 3:**

The objective of this exercise is to demonstrate the collective communication routines using MPI.COMM\_WORLD.Scatter() and MPI.COMM\_WORLD.Gather() methods. The following MPI program is using the scattering and gathering techniques to perform the following tasks :

• Generate an array of n numbers (generated randomly) on the root process (process 0).

• Scatter the numbers to p processes, given each process an equal amount of numbers.

• Each process computes the TOTAL of their subset of the numbers.

• Gather all the TOTAL to the root process.

• The root process then computes the TOTAL of these numbers.

**a) Compile and run the following MPI program. Observe and discuss the program and the output from the program.**

|  |
| --- |
| CODING :  import java.util.\*;  import mpi.\*;  public class MPISumArray {  public static void main(String args[]) throws Exception  {  MPI.Init(args);  int rank= MPI.COMM\_WORLD.Rank(); // store the rank#  int size = MPI.COMM\_WORLD.Size(); //# of processes ie. -np 4  int master =0;  int z=1;  int unitSize=25; // the length for each unit  int inputSize= unitSize\*size; // the length of input  double sendbuf[]= new double[inputSize]; // send buffer  double recvbuf[]=new double[unitSize]; // receive buffer  double sum[]= new double[unitSize]; // array to store sum  double global\_sum= 0;  long startTime=0; // start execution time  long elapsedTime=0; // elapsed time    if(rank==master)  {  startTime=System.nanoTime(); // measure start time    //generate a random array of numbers  Random random = new Random();  System.out.println("Random Number produced by Process :"+ master);    for(int i=0; i<sendbuf.length; i++)  {  // it will generate random number from zero to 100  sendbuf[i]= random.nextInt(100);  System.out.print((i+1)+") "+sendbuf[i]+"\t");    if(i==(sendbuf.length-1))  {  System.out.print(".");  System.out.println("\n\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_");  break;  }    if(z==5) // format output, 5 data in a line  {  System.out.println(" ");  z=0;  }  z++;  }  }  //distribute number to all processes to get summation of numbers  MPI.COMM\_WORLD.Scatter(sendbuf, 0, unitSize, MPI.DOUBLE, recvbuf, 0, unitSize, MPI.DOUBLE,  master);    for(int j=0;j<unitSize;j++)  {  sum[0]+=recvbuf[j]; // calculate total  }    for(int x=0;x<size;x++)  {  if(rank==x)  {  System.out.println("\n\n");    //to display the numbers that the processes receive  System.out.println("\n\nProcess "+rank+ " received : \t");  for(int j=0;j<recvbuf.length;j++)  {  System.out.print(rank+":"+recvbuf[j]);  if(j==(recvbuf.length)-1)  {  System.out.print(". ");  break;  }  else  System.out.print(", ");  }    System.out.println("\nsum for rank "+rank+": "+sum[0]);  break;  }  }      // to gather the sum each process to a master  MPI.COMM\_WORLD.Gather(sum, 0, 1, MPI.DOUBLE , sendbuf, 0, 1, MPI.DOUBLE, master);    if(rank==master)  {  for(int i=0;i<size;i++)  global\_sum+=sendbuf[i];  System.out.println("Total sum of all produced numbers = "+global\_sum);  }  if(rank==master)  {  elapsedTime=System.nanoTime()-startTime; // measure time  System.out.println("Total Execution Time : "+(elapsedTime/1000000.0)+"  milliseconds ");  }  MPI.Finalize();  }  } |
| OUTPUT : |

**b.) Produce a serial program to calculate and display the total and average of n (range of 100 -500) numbers in an array of data (integer or double).**

|  |
| --- |
| CODING :  import java.util.\*;  public class SerialProgramming {  public static void main(String[] args) {  int inputSize = 100;  double[] n = new double[inputSize];  double sum = 0;  double average = 0;  long startTime, elapsedTime;  int z = 1;  startTime = System.nanoTime();  Random random = new Random();  System.out.println("Random Number produced by Process: ");  for(int i=0; i<n.length; i++){  n[i]=random.nextInt(100);  System.out.print((i+1)+")"+n[i]+"\t\t");  if(i==(n.length-1)){  System.out.println();  System.out.println("---------------------------------------------------------------------------");  break;  }  if(z == 5){  System.out.println();  z = 0;  }  z++;  }  for(int i=0; i<n.length; i++){  sum = sum + n[i];  }  average = sum/n.length;  System.out.println();  System.out.println("Sum: "+sum);  System.out.println("Average: "+average);  elapsedTime = System.nanoTime()-startTime;  System.out.println("Total execution time: "+(elapsedTime/1000000.0)+" milliseconds");  }  } |
| OUTPUT : |
|  |

**c) Modify the parallel program which uses the scattering and gathering**

**techniques (refer to slides) to perform the following tasks given below:**

• Generate an array of n numbers on the root process (process 0).

(Note: specify the value of n in the range of 100 – 50000)

• Scatter the numbers to p processes, given each process an equal amount

of numbers.

(Note: specify the value of p in the range of 4 - 10)

• Each process computes the AVERAGE of their subset of the numbers.

• Gather all the AVERAGES to the root process.

• The root process then computes the AVERAGE of these numbers to get the final average.

|  |
| --- |
| CODING :  package mpisumarray;  import java.util.\*;  import mpi.\*;  public class ParallelProgramming {  public static void main(String[] args) throws Exception {    MPI.Init(args);  int rank = MPI.COMM\_WORLD.Rank();  int size = MPI.COMM\_WORLD.Size();  int master = 0;  int z = 1;  int inputSize = 100;  int unitSize = inputSize/size;  double sendbuf[] = new double[inputSize];  double recvbuf[] = new double[unitSize];  double sum[] = new double[unitSize];  double average[] = new double[unitSize];  double global\_sum = 0;  double global\_average = 0;  long startTime = 0;  long elapsedTime = 0;    if(rank==master)  {  startTime = System.nanoTime();  Random random = new Random();  System.out.println("Random Number produced by Process: " +  master);  for(int i=0; i<sendbuf.length; i++){  sendbuf[i] = random.nextInt(100);  System.out.print((i+1)+")"+sendbuf[i]+"\t\t");    if(i==(sendbuf.length-1))  {  System.out.println();  System.out.println("---------------------------------------------------------------------------");  break;  }    if(z==5){  System.out.println();  z=0;  }  z++;  }  }    //distribute number to all processes to get summation of numbers  MPI.COMM\_WORLD.Scatter(sendbuf, 0, unitSize, MPI.DOUBLE,  recvbuf, 0, unitSize, MPI.DOUBLE, master);    for(int j=0; j<unitSize; j++)  {  sum[0]+=recvbuf[j]; // calculate total  }  average[0] = sum[0];//inputSize    for(int x=0; x<size; x++)  {  if(rank==x)  {  System.out.println("\n\nProcess "+rank+" received :");    for(int j=0; j<recvbuf.length; j++)  {  System.out.print(rank+":"+recvbuf[j]);  if(j==(recvbuf.length)-1)  {  System.out.println(". ");  break;  }  else  System.out.print(", ");  }  System.out.println("\nSum for rank "+rank+": "+sum[0]);  System.out.println("\nAverage for rank "+rank+": "+average[0]);  break;  }  }    // to gather the sum each process to a master  MPI.COMM\_WORLD.Gather(sum, 0, 1, MPI.DOUBLE, sendbuf, 0, 1,  MPI.DOUBLE, master);  if(rank == master){  for(int i=0; i<size; i++)  global\_sum+=sendbuf[i];  System.out.println("Total sum of all produced numbers = "+global\_sum);  }    // to gather the average each process to a master  MPI.COMM\_WORLD.Gather(average, 0, 1, MPI.DOUBLE, sendbuf, 0, 1,  MPI.DOUBLE, master);  if(rank==master){  for(int i=0; i<size; i++)  global\_average+=sendbuf[i];  System.out.println("Total average of all produced numbers = "+global\_average);  }    if(rank==master){  elapsedTime = System.nanoTime()-startTime;  System.out.println("Total Execution Time: "+(elapsedTime/1000000.0)+" milliseconds");  }  MPI.Finalize();  }  } |
| OUTPUT :  Table  Description automatically generated |

**d) Measure, record and analyze the execution time for both the serial and parallel programs. Discuss the finding from this measurement.**

Execution Analysis :

Based on the 3 tests for each programming:

|  |  |
| --- | --- |
| **Parallel Programming** | **Serial Programming** |
|  |  |
|  |  |
|  |  |

 Execution time taken by serial programming is less than parallel programming and it shows that serial programming is faster than parallel programming.

 It is because serial programming makes less communication that leads to less processing compared to parallel programming, thus make the execution time shorter.

 Threading has some overhead and typically does not pay off for short calculations in parallel makes it take more time compared to serial.