MONASH University

FIT3143 Lab Week 11

Lecturers: ABM Russel (MU Australia) and Vishnu Monn (CE, MU Malaysia)

PARALLEL SORT & SEARCH USING MPI

OBJECTIVES

- The purpose of this lab is to apply data parallelism for sort and search operations

INSTRUCTIONS

- Download and set up the Linux VM [Refer to Lab Week 1]
- Setup eFolio (including Git) and share with tutor and partner [Refer to Lab Week 1]

TASK

DESCRIPTION:

- Analysing a serial sorting algorithm.
- Analysing a parallel sorting algorithm using MPI.
- Design and implement a parallel search algorithm using MPI (Bonus).

WHAT TO SUBMIT:

1. E-folio document containing algorithm or code description, analysis of results, screenshot of the running programs and git repository URL.
2. Code in the Git.

EVALUATION CRITERIA

- **This Lab-work is not assessed.** Nevertheless, we do encourage you to attempt the questions in this lab.

MONASH University

LAB ACTIVITIES

Task 1 – Merge Sort Serial Code – Worked example

Merge sort is a comparison-based sorting algorithm. In most implementations it is stable, meaning that it preserves the input order of equal elements in the sorted output. It is an example of the divide and conquer algorithmic paradigm. It was invented by John von Neumann in 1945.

Write a serial-based merge sort code in C.

Sample solution:


```
//----------------------------------------------------------------
-----------------------------------------------------
// Merge Sort Code in serial implementation
//
// Author:
http://www.c.happycodings.com/Sorting_Searching/code11.html
//          - Initial version
//
//----------------------------------------------------------------
-----------------------------------------------------
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// Data
#define MAXARRAY 200

// Function prototype
void mergeSort(int[], int, int);

// Main program
int main(void)
{
     int data[MAXARRAY];
     int i = 0;

     // Load random data into the array
     // Note: srand() function is not used here.
     // Hence, random number generated will be same every time the
application is executed.
     // This makes it easier to view the sorted results.
     for(i = 0; i < MAXARRAY; i++)
     {
```

```c
            data[i] = rand() % 100;
        }

        // Print data before sorting
        printf("Before Sorting:\n");
        for(i = 0; i < MAXARRAY; i++)
        {
            printf(" %d", data[i]);
        }
        printf("\n");

        // Call the merge sort function
        mergeSort(data, 0, MAXARRAY - 1);

        // Print data after sorting
        printf("\n");
        printf("After sorting using Mergesort:\n");
        for(i = 0; i < MAXARRAY; i++)
        {
            printf(" %d", data[i]);
        }
        printf("\n");

        return 0;
}
```



```c
// Function definition
void mergeSort(int inputData[], int startPoint, int endPoint)
{
        int i = 0;
        int length = endPoint - startPoint + 1;
        int pivot  = 0;
        int merge1 = 0;
        int merge2 = 0;
        int working[MAXARRAY] = {0};

        if(startPoint == endPoint)
        {
            return;
        }
        pivot  = (startPoint + endPoint) / 2;

        // Recursive function call
        mergeSort(inputData, startPoint, pivot);
        mergeSort(inputData, pivot + 1, endPoint);

        for(i = 0; i < length; i++)
        {
            working[i] = inputData[startPoint + i];
        }

        merge1 = 0;
```

```
	merge2 = pivot - startPoint + 1;

	for(i = 0; i < length; i++)
	{
		if(merge2 <= endPoint - startPoint)
			if(merge1 <= pivot - startPoint)
				if(working[merge1] > working[merge2])
				{
					inputData[i + startPoint] = 
working[merge2++];
				}
				else
				{
					inputData[i + startPoint] = 
working[merge1++];
				}
			else
			{
				inputData[i + startPoint] = working[merge2++];
			}
		else
		{
			inputData[i + startPoint] = working[merge1++];
		}
	}
}
```

MONASH University

Task 2 – Merge Sort Parallel Code using MPI – Worked example

Modify the serial code in Task 1 into a parallel code using Message Passing Interface (MPI) in C. For this question, you are free to choose the type of MPI implementation. However, only the process with rank 0 should print out the data before and after the sorting process.

Sample solution:

```
// MPI MergeSort
//******************************************************************
*********************************************
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

// Define the size of data
#define N 200

// Function Prototype
void mergeSort(int* data, int startPoint, int endPoint);
void merge(int *A, int sizeA, int *B, int sizeB);

// Main Function
int main(int argc, char* argv[])
{
     // Variable declaration
     int i;
     int *data = NULL;   // Initialize data pointer to null
     int scale = 0;
     int currentLevel = 0;    // current level of tree
     int maxLevel = 0;   // maximum level of tree = LOG2 (number
of Processors)

     int pivot = 0;      // middle point of data array
     int length = 0;          // length of data array
     int rightLength = 0;     // length of child node data array

     int p;         // Number of Processors
     int myRank;    // Processor's rank
     MPI_Status status;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD,&p);
     MPI_Comm_rank(MPI_COMM_WORLD,&myRank);


     if(myRank == 0){
          // Root Node:
```

```
          maxLevel = log ((double)p) / log(2.00); // Calculate the
maximum level of binary tree
          length = N;     // Set the length of root node to N
          data = (int*)malloc(length * sizeof(int)); // Create
dynamic array buffer with size length

          // srand is not used to keep a constant set of random
values at each program execution for better debugging
          for(i = 0; i<N;i++)
               data[i] = rand()%100;

          printf("\n----------------------------------------------
-------------------------------\n");
          printf("Unsorted Data: \n");
          for(i = 0; i<N; i++){    // Prints out unsorted data
value
               if(i%10 == 0)  // 10 elements in a row
                    printf("\n");
               printf("%d\t",data[i]);
          }
          printf("\n----------------------------------------------
-------------------------------\n");
     }

     // Broadcast maxLevel to all other processors
     MPI_Bcast(&maxLevel, 1, MPI_INT, 0, MPI_COMM_WORLD);

     // Divide the data to child node
     for(currentLevel = 0; currentLevel <=maxLevel;
currentLevel++){
          scale = pow(2.00, currentLevel);

          // Parent node
          if(myRank/scale <1){
               if((myRank+scale)<p){                    // if child
node exist (child node rank < number of processors)
                    pivot = length / 2;                     //
Divide data length into half
                    rightLength = length - pivot;      // Set data
length for child node
                    length = pivot;
     // Set new data length for parent node

                    // Send child node length to the corresponding
child node
                    // tag = currentLevel
                    MPI_Send(&rightLength,1,MPI_INT,
myRank+scale,currentLevel,MPI_COMM_WORLD);

                    // Send the right half of data array
```

```
                    MPI_Send((int *)
data+pivot,rightLength,MPI_INT,myRank+scale,currentLevel,MPI_COMM_
WORLD);
               }
          }
          // Child node
          else if(myRank/scale < 2){
               // Receive length from parent node
               MPI_Recv(&length, 1, MPI_INT, myRank-
scale,currentLevel, MPI_COMM_WORLD, &status); //tag = currentLevel

               // Create new dynamic data buffer with length
received from parent
               data = (int*)malloc(length * sizeof(int));

               // Receive data array from parent
               MPI_Recv(data, length, MPI_INT,myRank-scale,
currentLevel, MPI_COMM_WORLD,&status);
          }
     }
```



```
     // All processors mergeSort their own data chunk with
respective length
     mergeSort(data,0,length-1);
```



```
     // Merge the sorted data from child node to the root node
starting
     // Begin the progress from the lowest level of the tree
structure
```



```
     for(currentLevel = maxLevel;currentLevel>=0;currentLevel--){
          scale = pow(2.00, currentLevel);
          if(myRank/scale<1){                              //
Parent node receive sorted data from child node
               if(myRank+scale<p)                          // If
child node exist (child node rank < number of processors)
               {
                    MPI_Recv(&rightLength, 1, MPI_INT,
myRank+scale, currentLevel, MPI_COMM_WORLD,&status);
                    MPI_Recv((int *) data+ length, rightLength,
MPI_INT, myRank+scale, currentLevel, MPI_COMM_WORLD, &status);

                    merge(data, length, (int *)data+length,
rightLength);  // Merge the data array
                    length+=rightLength; // Update the length of
merged data array
               }
          }
          // Child node sends sorted data to parent node
          // tag = current level
          else if(myRank/scale<2)
          {
               // Send the length of sorted data
```

```
                MPI_Send(&length, 1, MPI_INT, myRank-
scale,currentLevel, MPI_COMM_WORLD);
                MPI_Send(data, length, MPI_INT, myRank-scale,
currentLevel, MPI_COMM_WORLD);
           }
      }

      // Root node prints out the sorted data
      if(myRank == 0){
           printf("Sorted Data: \n");
           for(i = 0; i<length;i++){
                if(i%10 == 0)
                     printf("\n");
                printf("%d\t", data[i]);
           }
           printf("\n");
      }

      MPI_Finalize(); // Finalize MPI
      free(data);     // Free the data buffer

      return 0;
}
// End of main program

// Function mergeSort
void mergeSort(int* data, int startPoint, int endPoint)
{
      int pivot = (startPoint + endPoint)/2;

      // if last element then return
      if(startPoint == endPoint)
           return;
      //Recursive function call
      mergeSort(data, startPoint, pivot);
      mergeSort(data,pivot+1,endPoint);

      // Merge the sorted data from both sides into the data buffer
      merge(data+startPoint, pivot - startPoint +1, data+pivot+1,
endPoint-pivot);
}
// End of function mergeSort

// Function merge
void merge(int *A, int sizeA, int *B, int sizeB)
{
      int sizeC = sizeA + sizeB;
      int *C = (int*)malloc(sizeC * sizeof(int));
      int countA;
      int countB;
      int countC;
```

```
	// Merging the element from array A and array B into C in
ascending order
	for(countA = 0, countB = 0, countC = 0; countC< sizeC;
countC++){
		if(countA>=sizeA)			// If all the
element from A is stored into C
			C[countC] = B[countB++]; // store the remaining
element from B into C
		else if (countB>=sizeB)		// If all the element
from B is stored into C
			C[countC] = A[countA++];	// store the
remaining element from  A into C
		else
		{	// Store the element with smaller value into C then
increment the corresponding pointer array (A or B)
			if(A[countA]<=B[countB])
				C[countC] = A[countA++];
			else C[countC] = B[countB++];
		}
	}
	// Copy the merged data from C into A and B
	for(countA = 0; countA < sizeA; countA++)
		A[countA] = C[countA];

	for(countC = countA, countB = 0; countC < sizeC; countC++,
countB++)
		B[countB] = C[countC];

	// Free memory of C
	free(C);
}
// End of function merge
```


Task 3 – Analysing both Tasks 1 and 2

Analyse the sample solution in both Tasks 1 and 2. In your e-Folio sheet:

a) Describe the Serial Merge Sort algorithm from Task 1. Include illustrations, pseudocode and/or flowcharts to describe the algorithm.
b) Modify the code in Task 1 to perform sorting of a large array of numbers. Write the sorted results to a file. Perform a theoretical speed up analysis of the serial Merge Sort Algorithm.
c) Describe the Parallel Merge Sort algorithm from Task 2. Include illustrations, pseudocode and/or flowcharts to describe the algorithm.
d) Modify the code in Task 2 to perform sorting of a large array of numbers. Write the sorted results to a file. Perform an actual speed up analysis and compare your results with the theoretical speed up.

Note: You may use MonARCH to run Tasks 1 and 2.

ADDITIONAL OPTIONAL ACTIVITY (SELF PRACTICE)

Figure 1 represents a file (**data.txt**), which contains a set of unique product IDs and corresponding price. The first row in **data.txt** contains ***N*** total number of product records, and the subsequent ***N*** rows contain a list of product IDs and corresponding product price.

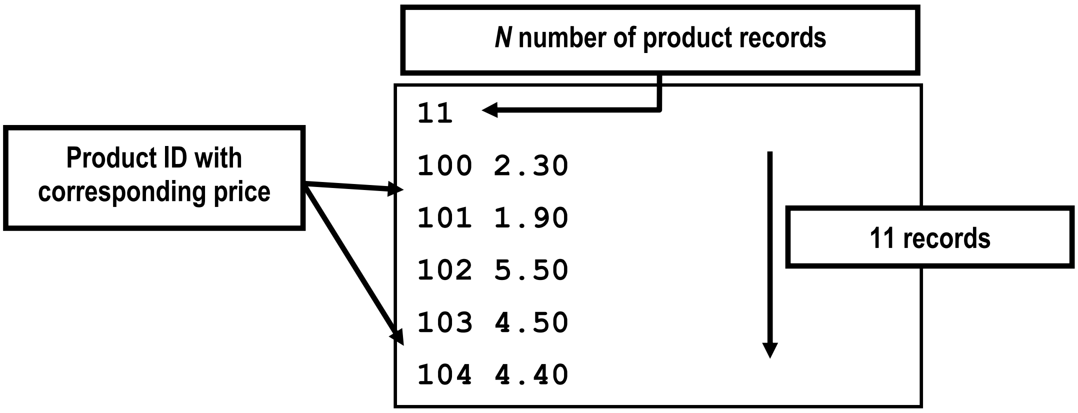

Figure 1: Content of data.txt

Note: The file, **data.txt** is located only at the root node processor. Other processors cannot access this file. *Create your own* **data.txt** *based on the sample format as per the above figure*.

Using the C programming language:

(a) Write a function, **`BinSearch()`** to locate the index of a searched product based on an input product ID buffer using **binary search algorithm**. The following function prototype is provided:

Product ID to look for

Input buffer IDs to search

```c
int BinSearch(int key, int *pId, int start, int end);
```

Start and End indexes

The **`BinSearch()`** function will **return the index of the searched** product if a matching is found, and **-1** if the name is not found. The binary (iterative) search algorithm is as follows (You can also use the recursive method):

```
while start index < end index
    find the pivot (i.e. the index of the middle element of the range of elements to be
              searched)
    compare the element at pivot with the key
        if element at pivot is less than the key
```

```
        end index = pivot – 1
    else if element at pivot is greater than the key
        start index = pivot + 1
    else
        return pivot
return -1
```

Note: There is <u>no need</u> to apply any MPI design in the **`BinSearch()`** function.

(b) Using the function of part (a), write a **`main()`** function **with MPI implementation** to do the following:

(i) The root node loads the contents of **data.txt** into two separate dynamic buffers. The first buffer contains the product IDs and the second buffer contains the corresponding product prices. The size of these buffers is based on the first item as read from the **data.txt** file.

As the root node can only access the content of **data.txt** file, hence, <u>the content of the product ID buffer is to be equally divided among the processors.</u>

(ii) Prompt the user for an input product ID to search. Only the root node can request an input product ID from the user.

(iii) Perform the binary search operation based on the equally divided content among the processors. Use the **`BinSearch()`** function as implemented in part (a).

Note: <u>Do not</u> modify the implementation of **`BinSearch()`** function.

(iv) The search results are returned to the root node. If a match is found, the root node prints the price of the searched product. Otherwise, the root node prints a message indicating that the searched product ID does not exist.

Write parts (a) and (b) above as a single program. **Figures 2 & 3** displays a sample execution of this program. Test your program using **OpenMPI** with <u>three or four processes. Assume that there are no repetitions of the product IDs in **data.txt**.</u>

```
Total products: 11
Number of processors: 4
Product ID to search >> 104
Price of product ID: 104 is RM 4.40.
```

Figure 2: Sample Program Execution

```
Total products: 11
Number of processors: 4
Product ID to search >> 7788
No such product.
```

Figure 3: Another sample Program Execution

Note: Underline statements in **Figures 2 & 3** denote a user's input.