Wilfrid Laurier University

Department of Physics and Computer Science

MPI Programming for Multiplication Tables

Group 1
Heba Adi | 201653820 | adix3820@mylaurier.ca
Maham Farooq | 200498180 | faro8180@mylaurier.ca
Majed Hadida | 200436110 | hadi6110@mylaurier.ca
Pranav Gangwani | 193097120 | gang7120@mylaurier.ca
Yash Rojiwadia | 203039360 | roji9360@mylaurier.ca

CP431 – Parallel Programming

Professor Illias Kotsireas

April 7th, 2023

Table of Contents

Introduction

In this report, we aim to develop an efficient implementation of a parallel algorithm for computing the number of different elements in an N × N multiplication table. This problem is interesting because it allows us to quantify the degree of repetition in the multiplication table and also provides a valuable opportunity to explore parallel computing techniques. Our approach involves taking advantage of the symmetry of the multiplication table and using parallelization to optimize performance. Through our work, we hope to provide insights into the efficiency of parallel algorithms for this type of problem and potentially identify new avenues for further research in this area.

Examples

Example 1

Verify by hand that M(5) = 14.

	1	**2**	**3**	**4**	**5**
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25

It is evident that M(5) = 14 because we can count all the non-repeat squares above the main diagonal (including the main diagonal). The values to be counted are in black font.

Example 2

Verify by hand that M(10) = 42.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

It is evident that M(10) = 42 because we can count all the non-repeat squares above the main diagonal (including the main diagonal). The values to be counted are in black font.

Code Explanation

```c
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

static const int TAG_ARRAY = 0;
static const int TAG_LAST = 1;
static const int SLICER = 30000;
```

This section of code includes the necessary header files for MPI (Message Passing Interface), standard input/output library, standard library, and math library. The constant integers TAG_ARRAY, TAG_LAST, and SLICER are declared and initialized with the values 0, 1, and 30000, respectively. These constants will be used throughout the program to define different tags and control the parallel computation. TAG_ARRAY is used to tag messages that include data arrays, while TAG_LAST is used to tag the last message sent in a communication operation. SLICER is the constant that indicates the total number of bits per process used. The number is chosen arbitrarily to evenly split the significantly large productArray which is then communicated to the root process later in the code.

```c
int main(int argc, char *argv[])
{
  int myRank;
  int numProc;

  double startTime = 0.0;
  double endTime = 0.0;

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

  if (argc < 2){
    if (myRank == 0){
      printf("Invalid arguments, use ./main n\n");
    }

    MPI_Finalize();
    return 0;
  }
```

The variables startTime and endTime are also declared and initialized with the value 0.0, which will be used later to measure the execution time of the program. The function MPI_Init() initializes the MPI environment, and MPI_Comm_rank() and MPI_Comm_size() are used to obtain the rank and size of the current process, respectively. The program then checks if the user provided at least one argument. If no arguments are provided, the program prints an error message to the console from process 0 and exits the program using MPI_Finalize().

```
startTime = MPI_Wtime();

long n = atol(argv[1]);
unsigned long matrix = n * n;

unsigned long matrixH = ((matrix - n) / 2) + n;

unsigned long coreW[numProc];
for (int i = 0; i < numProc; i++){
  coreW[i] = floor((float) matrixH / numProc);

  if (i < matrixH % numProc){
    coreW[i] += 1;
  }
}

MPI_Barrier(MPI_COMM_WORLD);

unsigned long endp = 0;
unsigned long counter = 1;

for (int i = 0; i < myRank; i++){
  endp += coreW[i];
}
```

This code section starts by using MPI_Wtime() to get the start time of the program execution. The program then reads the first argument passed to the program, converts it to a long integer using atol(), and computes the total number of elements in the multiplication table by multiplying ‘n’ with itself. This value is stored in the ‘matrix’ variable. The variable ‘matrixH’ is then calculated by halving the matrix after subtracting the diagonal which represents the upper triangle. The diagonal is then re-added in afterwards within the equation. Next, an array ‘coreW’ is declared with a length of ‘numProc’. The ‘coreW’ array is used to store the number of elements that each process will compute. The for loop iterates over each process and calculates the workload of each process based on the number of elements in the matrix and the number of processes. MPI_Barrier() is used to synchronize all processes before proceeding to the next part of the program. After the synchronization, two variables ‘endp’ and ‘counter’ are declared and initialized with the values 0 and 1, respectively. A second for loop is then used to compute the ending position of its previous processor and starting position for the current processor by iterating over the array of the number of elements that each processor will compute until the current processor.

```
unsigned long i = 1;
unsigned long j = 1;

while (counter <= endp){
  counter++;
  i++;
  if (i == (n + 1)){
      j++;
      i = j;
  }
}


unsigned long section = coreW[myRank];
unsigned char *productArray = (unsigned char*) calloc(matrix + 1, sizeof(unsigned char));

if (productArray == NULL){
    printf("Failed to allocate product array\n");
}
```

This code section initializes two variables ‘i’ and ‘j’ with the value 1. These variables are used to keep track of the row and column of the current element being processed in the multiplication table. The while loop iterates over all elements in the matrix up to the starting position of the current process. The loop increments the counter, ‘i’, and ‘j’ variables as the position of the current element is updated. If the ‘i’ column reaches ‘n + 1’ which is the last position in the row, ‘j’ row value is incremented by 1 which enables continuation to the next row. Then, the ‘i’ value is reset to the value of ‘j’ as we only consider half of the matrix with the diagonal included. After the loop, the variable ‘section’ is set to the workload of the current process, which was previously computed. The program then dynamically allocates an array ‘productArray’ of size ‘matrix + 1’ to hold the product of each element in the multiplication table. The program then checks if the memory allocation was successful. If ‘productArray’ is ‘NULL’, it prints an error message to the console.

```
unsigned long product;
counter = 0;
while (counter < section){
  product = i * j;


  if (product > matrix) {

  product = matrix;
  }

  productArray[product] = 't';

  i++;
  counter++;
  if (i == (n + 1)){
    j++;
    i = j;
  }
}
```

This code section computes the product of each element within the current process's assigned section of the multiplication table, and stores the result in the ‘productArray’ using the product value as an index. The loop iterates over all elements within the assigned section, which is determined by the ‘section’ variable. For each element, the program calculates the product of the corresponding row ‘i’ and column ‘j’ using the expression ‘product = i * j;’. If the product is greater than the maximum value of the ‘matrix’, which is matrix, the product is set to ‘matrix’ to avoid out-of-bounds array access. The program then sets the corresponding element of the ‘productArray’ to the character 't'. This is used to indicate that the element corresponding to the product has been seen in the multiplication table. The loop increments the ‘i’ and ‘counter’ variables, and if ‘i’ equals ‘n+1’, it increments ‘j’ and resets ‘i’ to ‘j’, just like in the previous code section.

```
unsigned long temp = matrix + 1;

unsigned char* slice;
i = 0;
j = temp % SLICER;
unsigned long k = temp / SLICER;

if (myRank != 0) {
        while (k > 0) {
                slice = &productArray[i];
                MPI_Send(slice, SLICER, MPI_UNSIGNED_CHAR, 0, TAG_ARRAY, MPI_COMM_WORLD);
                i += SLICER;
                k--;
        }
        slice = &productArray[i];
        MPI_Send(slice, j, MPI_UNSIGNED_CHAR, 0, TAG_LAST, MPI_COMM_WORLD);
        free(productArray);
}
```

This code section is responsible for sending the computed ‘productArray’ data from each process to the root process (rank 0) for final processing. The first line of the code initializes the ‘temp’ variable with the maximum index in the ‘productArray’ plus one. The program then initializes the variables ‘i’, ‘j’, and ‘k’. The ‘j’ variable determines the remainder by setting the modulo of temp and the constant ‘SLICER’. The ‘k’ variable determines the quotient by dividing temp by SLICER. The ‘i’ variable is initialized to zero and will be used to keep track of the current position in ‘productArray’. The program then enters a conditional block that checks if the current process is not the root process (rank 0). If the process is not rank 0, the program enters a loop that iterates ‘k’ times. In each iteration, the program sends a slice of ‘productArray’ data to rank 0 using the MPI_Send function. The size of each slice is the constant ‘SLICER’, and the slice is in the ‘productArray’ starting from index ‘i’. The program increments ‘i’ by ‘SLICER’ at each ‘j’ iteration. After the loop, the program sends a final slice to rank 0 with size ‘j’. The slice is in the ‘productArray’ starting from index ‘i’. The MPI_Send function uses the TAG_LAST tag to signal that this is the final slice of data to be sent. Finally, the program frees the memory allocated for the ‘productArray’ if the current process is not rank 0.

```
if (myRank == 0) {

  unsigned char *recvArray =
  (unsigned char*) calloc(matrix + 1, sizeof(unsigned char));
  if (recvArray == NULL) {
    printf("Failed to allocate recvArray\n");
  }
  for (int rank = 1; rank < numProc; rank++) {

    i = 0;
    k = temp / SLICER;
    while (k > 0) {
      MPI_Recv(recvArray, SLICER, MPI_UNSIGNED_CHAR, rank, TAG_ARRAY, MPI_COMM_WORLD, NULL);
      for (counter = 0; counter < SLICER; counter++) {
        if (recvArray[counter]) {
          productArray[i] = 't';
        }
        i++;
      }
      k--;
    }
    MPI_Recv(recvArray, j, MPI_UNSIGNED_CHAR, rank, TAG_LAST, MPI_COMM_WORLD, NULL);
    for (counter = 0; counter < j; counter++) {
      if (recvArray[counter]) {
        productArray[i] = 't';
      }
      i++;
    }
  }
```

This section of the code is executed only by the process with rank 0. It receives the data that was sent by the other processes (with rank 1 to rank numProc-1) and merges it into the final result. The process with rank 0 first allocates memory for the receive array ‘recvArray’, which is also of size ‘matrix+1’ bytes. Then, it enters a loop that goes over all the ranks from 1 to numProc-1, and for each rank, it receives data in ‘SLICER’ sized chunks, until all of the data sent by that rank is received. Inside the loop, ‘i’ is set to 0, and ‘k’ is set to the number of ‘SLICER’-sized chunks that the rank sent to rank 0. Then, there is another loop that receives each ‘SLICER’-sized chunk of data sent by the rank using the MPI_Recv function. For each element in the received data, the loop checks whether it is nonzero. If it is, it sets the corresponding element in ‘productArray’ to 't'. After the ‘SLICER’-sized chunks are all received, the loop receives the last chunk of data, which is of size j. This is done separately, using another call to MPI_Recv. Then, the loop goes over each element in this last chunk of data, and sets the corresponding element in ‘productArray’ to 't' if it is nonzero. In this way, the process with rank 0 collects all of the data sent by all of the other processes, and merges it into the final result stored in ‘productArray’.

```
            unsigned long totalCount = 0;
            for (counter = 1; counter <= matrix; counter++) {
                if (productArray[counter]) {
                    totalCount++;
                }
            }

            free(recvArray);
            free(productArray);
            printf("Number (N): %lu\n", n);
            printf("Total no. of different elements: %lu\n", totalCount);
            endTime = MPI_Wtime();
            printf("Time elapsed: %lf seconds\n", endTime - startTime);
            printf("----------------------------------------------------------------\n" );
        }
    MPI_Finalize();
    return 0;
}
```

This part of the code calculates the total number of distinct elements in the ‘productArray’. It first initializes a counter called ‘totalCount’ to 0, then loops through every index of the ‘productArray’ from 1 to ‘matrix’ (inclusive). For each index, if the value at that index is non-zero (i.e., 't'), then ‘totalCount’ is incremented by 1. After the loop finishes, the function frees the memory used by the ‘recvArray’ and ‘productArray’, prints out the value of ‘n’, the total number of distinct

elements (totalCount), and the time elapsed for the program to run, and then finalizes the MPI environment and returns 0 to indicate successful completion of the program.

Load Balancing

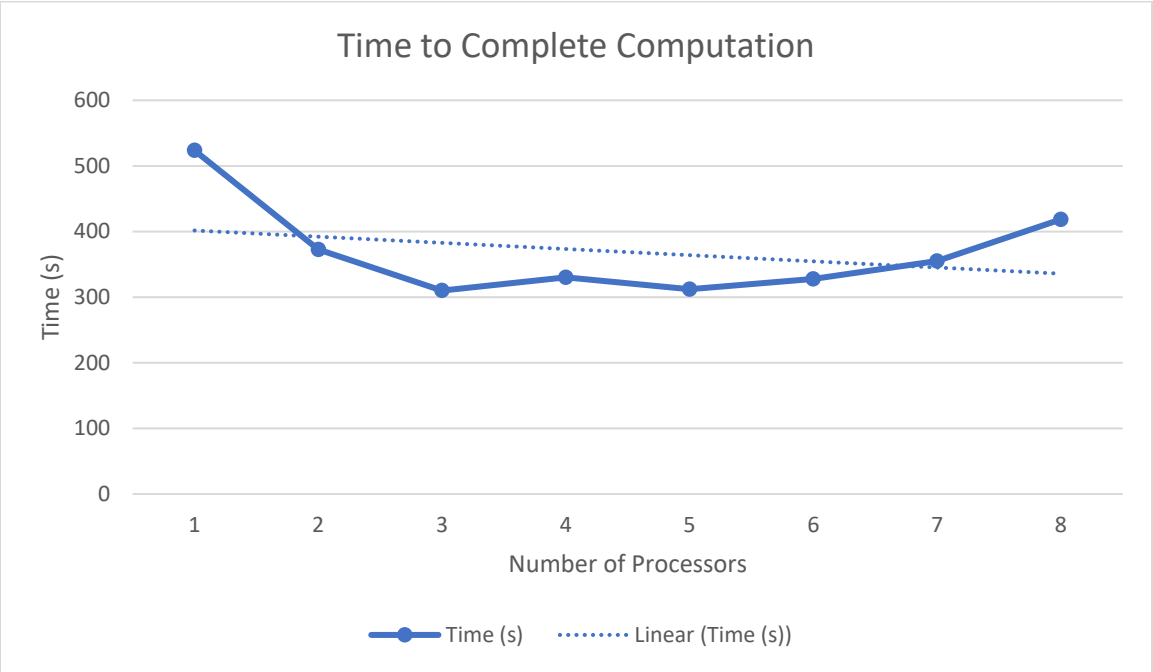
	1	2	3	4	5
1	1	2	3	4	5
2		4	6	8	10
3			9	12	15
4				16	20
5					25

The computation of M(N) for large N involves a significant amount of computation that can benefit from parallel processing. However, to achieve optimal performance, load balancing is critical to ensure that each processor is assigned an equal amount of work. Let's take a simple example to understand our implementation. Let's have 5 x 5 matrix as shown above in picture. We first divided the matrix in half and then include the diagonal. We have 15 numbers, now let's divide the total number of numbers to calculate with number of processors, which is 4, giving us the 3 as quotient and remainder as 3. We will increment each processor with additional numbers to calculate till the remaining 3 numbers are exhausted. This is the distribution shown in the above picture with colors. Now, we will save the no. of numbers each processor calculates in an array (coreW). We will then calculate the current processor's starting position by endp variable which has sum of all the no. of numbers previous processors are calculating. So, here let's take processor 3, the coreW array will be {4,4,4,3} and endp will be 4 + 4 = 8. We will then advance the 'i' and 'j' variables for row and columns respectively incrementing by endp (8 numbers in this example) to appropriate row and columns, in this case, we advance 'i' to 5, and 'j' to 2 as starting position for processor 3. Then we will calculate the product of 'i' and 'j' and save it in productArray. Here the first value will be 10 which will be saved at index 9 in productArray as flag 't' (True). Similarly, it will go on for 3 more values for this processor. This process will be running in all the processors calculating their own respective parts.

Benchmarking

The following table depicts how the algorithm runs based on the number of processors used. The N value is 100000.

Number Of Processors	Time (seconds)
1	523.5901
2	372.5815
3	310.0173
4	330.4320
5	312.0063
6	327.5956
7	355.0999
8	418.3302



The benchmarking results showed a clear relationship between the number of processors used and the time taken to compute the number of distinct products in the multiplication table. As the number of processors increased, the computation time decreased significantly. For instance, when using two processors, the running time was 372.58 seconds, while with three processors, the time reduced to 310.017 seconds. This observation can be explained by the fact that parallel processing enables multiple processors to handle different parts of the data simultaneously, resulting in faster computation and increased system performance.

However, as the number of processors increased, there was also an increase in processor overhead, which refers to the additional time required to manage and synchronize the actions of multiple processors. While adding more processors can significantly decrease the running time, there may be a point where the overhead of managing multiple processors outweighs the benefits of parallel processing, resulting in longer overall computation times. This saturation occurs after 3 processors where any additional processors can’t outweigh their own overhead, thus increasing overall computation time with increasing number of processors.

Results

1 Core

The following results is retrieved when running the code on **1 core**:

```
Number of Cores: 1
Number (N): 5
Total no. of different elements: 14
Time elapsed: 0.000174765 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 10
Total no. of different elements: 42
Time elapsed: 0.000184213 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 20
Total no. of different elements: 152
Time elapsed: 0.000201699 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 40
Total no. of different elements: 517
Time elapsed: 0.000184085 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 80
Total no. of different elements: 1939
Time elapsed: 0.000224580 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 160
Total no. of different elements: 7174
Time elapsed: 0.000376062 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 320
Total no. of different elements: 27354
Time elapsed: 0.001063744 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 640
Total no. of different elements: 103966
Time elapsed: 0.003527007 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 1000
Total no. of different elements: 248083
Time elapsed: 0.008834236 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 2000
Total no. of different elements: 959759
Time elapsed: 0.046311467 seconds
-------------------------------------------------------
```

```
Number of Cores: 1
Number (N): 8000
Total no. of different elements: 14509549
Time elapsed: 0.839209679 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 16000
Total no. of different elements: 56705617
Time elapsed: 8.293983539 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 32000
Total no. of different elements: 221824366
Time elapsed: 47.490260424 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 40000
Total no. of different elements: 344461977
Time elapsed: 77.388199017 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 50000
Total no. of different elements: 534772294
Time elapsed: 125.342248942 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 60000
Total no. of different elements: 766265747
Time elapsed: 183.604790276 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 70000
Total no. of different elements: 1038159725
Time elapsed: 252.562154439 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 80000
Total no. of different elements: 1351433133
Time elapsed: 332.182944503 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 90000
Total no. of different elements: 1704858126
Time elapsed: 422.511353656 seconds
-------------------------------------------------------
Number of Cores: 1
Number (N): 100000
Total no. of different elements: 2099198630
Time elapsed: 523.590066602 seconds
-------------------------------------------------------
```

3 Cores

The following results is retrieved when running the code on 3 cores:

```
Number of Cores: 3
Number (N): 5
Total no. of different elements: 14
Time elapsed: 0.000203902 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 10
Total no. of different elements: 42
Time elapsed: 0.000271378 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 20
Total no. of different elements: 152
Time elapsed: 0.000280608 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 40
Total no. of different elements: 517
Time elapsed: 0.000310844 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 80
Total no. of different elements: 1939
Time elapsed: 0.000391914 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 160
Total no. of different elements: 7174
Time elapsed: 0.000772723 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 320
Total no. of different elements: 27354
Time elapsed: 0.001964694 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 640
Total no. of different elements: 103966
Time elapsed: 0.007185174 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 1000
Total no. of different elements: 248083
Time elapsed: 0.016988158 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 2000
Total no. of different elements: 959759
Time elapsed: 0.069974121 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 8000
Total no. of different elements: 14509549
Time elapsed: 1.395050852 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 16000
Total no. of different elements: 56705614
Time elapsed: 7.586812389 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 32000
Total no. of different elements: 221824366
Time elapsed: 31.558672871 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 40000
Total no. of different elements: 344461994
Time elapsed: 49.339919896 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 50000
Total no. of different elements: 534772262
Time elapsed: 76.777099028 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 60000
Total no. of different elements: 766265747
Time elapsed: 111.434203586 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 70000
Total no. of different elements: 1038159781
Time elapsed: 151.676322015 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 80000
Total no. of different elements: 1351433197
Time elapsed: 198.548886228 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 90000
Total no. of different elements: 1704858126
Time elapsed: 250.702229995 seconds
-----------------------------------------------------------
Number of Cores: 3
Number (N): 100000
Total no. of different elements: 2099198630
Time elapsed: 310.017328040 seconds
-----------------------------------------------------------
```

8 Cores

The following results is retrieved when running the code on 8 cores:

Number of Cores: 8 Number (N): 5 Total no. of different elements: 14 Time elapsed: 0.000249804 seconds	Number of Cores: 8 Number (N): 8000 Total no. of different elements: 14509549 Time elapsed: 2.234310321 seconds
Number of Cores: 8 Number (N): 10 Total no. of different elements: 42 Time elapsed: 0.000286830 seconds	Number of Cores: 8 Number (N): 16000 Total no. of different elements: 56705617 Time elapsed: 9.608893369 seconds
Number of Cores: 8 Number (N): 20 Total no. of different elements: 152 Time elapsed: 0.000330119 seconds	Number of Cores: 8 Number (N): 32000 Total no. of different elements: 221824366 Time elapsed: 39.683598378 seconds
Number of Cores: 8 Number (N): 40 Total no. of different elements: 517 Time elapsed: 0.000458851 seconds	Number of Cores: 8 Number (N): 40000 Total no. of different elements: 344461977 Time elapsed: 61.954571102 seconds
Number of Cores: 8 Number (N): 80 Total no. of different elements: 1939 Time elapsed: 0.000620379 seconds	Number of Cores: 8 Number (N): 50000 Total no. of different elements: 534772294 Time elapsed: 97.132058092 seconds
Number of Cores: 8 Number (N): 160 Total no. of different elements: 7174 Time elapsed: 0.001291136 seconds	Number of Cores: 8 Number (N): 60000 Total no. of different elements: 766265747 Time elapsed: 140.145832494 seconds
Number of Cores: 8 Number (N): 320 Total no. of different elements: 27354 Time elapsed: 0.003893245 seconds	Number of Cores: 8 Number (N): 70000 Total no. of different elements: 1038159725 Time elapsed: 194.239622231 seconds
Number of Cores: 8 Number (N): 640 Total no. of different elements: 103966 Time elapsed: 0.014057253 seconds	Number of Cores: 8 Number (N): 80000 Total no. of different elements: 1351433133 Time elapsed: 260.764070429 seconds
Number of Cores: 8 Number (N): 1000 Total no. of different elements: 248083 Time elapsed: 0.033273684 seconds	Number of Cores: 8 Number (N): 90000 Total no. of different elements: 1704858126 Time elapsed: 335.210650671 seconds
Number of Cores: 8 Number (N): 2000 Total no. of different elements: 959759 Time elapsed: 0.131912025 seconds	Number of Cores: 8 Number (N): 100000 Total no. of different elements: 2099198630 Time elapsed: 418.330237625 seconds