Implementation of Computational Algorithms using Parallel Programming

Youssef Bassil

Researcher, LACSC – Lebanese Association for Computational Sciences, Beirut, Lebanon

ABSTRACT

Parallel computing is a type of computation in which many processing are performed concurrently often by dividing large problems into smaller ones that execute independently of each other. There are several different types of parallel computing. The first one is the shared memory architecture which harnesses the power of multiple processors and multiple cores on a single machine and uses threads of programs and shared memory to exchange data. The second type of parallel computing is the distributed architecture which harnesses the power of multiple machines in a networked environment and uses message passing to communicate processes actions to one another. This paper implements several computational algorithms using parallel programming techniques namely distributed message passing. The algorithms are Mandelbrot set, Bucket Sort, Monte Carlo, Grayscale Image Transformation, Array Summation, and Insertion Sort algorithms. All these algorithms are to be implemented using C#.NET and tested in a parallel environment using the MPI.NET SDK and the DeinoMpi API. Experiments conducted showed that the proposed parallel algorithms have faster execution time than their sequential counterparts. As future work, the proposed algorithms are to be redesigned to operate on shared memory multi-processor and multi-core architectures.

KEYWORDS: Parallel Computing, Distributed Algorithms, Message Passing

Table 1: Mandelbrot Testing Results

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential execution time</td>
<td>18s 578ms</td>
</tr>
<tr>
<td>Parallel execution time</td>
<td>8s 78ms</td>
</tr>
<tr>
<td>Speedup factor</td>
<td>$t_s / t_p = 18578/8078 = 2.3$</td>
</tr>
</tbody>
</table>

Figure 1 shows the execution of the Mandelbrot set program over 2 cores. The master drew the pixels in purple while the slave drew it in red.
B. Source Code

```csharp
private void Start()
{
    int width = 640, height = 480;
    double complexReal, complexImag;
    double MIN_REAL = -2; // FIXED
    double MAX_REAL = 2; // FIXED
    double MIN_IMG = -2; // FIXED
    double MAX_IMG = 2; // FIXED

    Bitmap bitmap1 = new Bitmap(width, height);
    DateTime t1 = DateTime.Now; // Start time
    string[] args = null;
    using (new MPI.Environment(ref args))
    {
        Communicator comm = Communicator.world;
        int region = height / num_proc;
        if (comm.Rank == 0) // MASTER
        {
            for (int i = 0; i < num_proc; i++)
            {
                comm.Send(i, 0); // send the height_From to RANK i with TAG 0
                comm.Send(i, region, 1); // send the height_To to RANK i with TAG 1
            }
        }
        for (int x = 0; x < width; x++)
        {
            for (int y = 0; y < height / 2; y++)
            {
                complexReal = MIN_REAL + x * (MAX_REAL - MIN_REAL) / width;
                complexImag = MIN_IMG + y * (MAX_IMG - MIN_IMG) / height;
                int iteration = cal_pixel(complexReal, complexImag);
                if (iteration == maxIteration)
                {
                    bitmap1.SetPixel(x, y, Color.BlueViolet);
                }
            }
        }
    }
    Bitmap bitmap2 = comm.Receive<Bitmap>(1, 1);
    DateTime t2 = DateTime.Now; // Stop time
    TimeSpan duration = t2 - t1;
    timeLabel.Text = "Time: " + duration.Seconds + "s" +
    duration.Milliseconds + "ms";

    // Display the Mandelbrot Set
    pictureBox1.BackgroundImage = (Image)bitmap1;
    pictureBox2.BackgroundImage = (Image)bitmap2;
}
else // ANY SLAVE
{
    int height_From = comm.Receive<int>(0, 0);
    int height_To = comm.Receive<int>(0, 1);
    Bitmap bitmap2 = new Bitmap(width, height);
    for (int x = 0; x < width; x++) // x = x co-ordinate of pixel
    {
        for (int y = height_From; y < height_To; y++)
        {
            complexReal = MIN_REAL + x * (MAX_REAL - MIN_REAL) / width;
            complexImag = MIN_IMG + y * (MAX_IMG - MIN_IMG) / height;
            int iteration = cal_pixel(complexReal, complexImag);
            if (iteration == maxIteration)
            {
                bitmap2.SetPixel(x, y, Color.Red);
                else bitmap2.SetPixel(x, y, Color.Black);
            }
        }
    }
    comm.Send(bitmap2, 0, 1); // send the bitmap to RANK 0 with TAG 1
}

private int cal_pixel(double complexReal, double complexImag)
{
    double lengthsq, temp;
    double real = 0, imag = 0; // Always Initial Values
    int iteration = 0;
    do
    {
        temp = (real * real) - (imag * imag) + complexReal;
        imag = 2 * real * imag + complexImag; // Fixed Formula
        real = temp;
        lengthsq = real * real + imag * imag; // Fixed Formula
        iteration++;
    } while (lengthsq < 4.0) && (iteration < maxIteration);
    return iteration;
}

II. BUCKET SORT ALGORITHM

Bucket sort, or bin sort, is a sorting algorithm that works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm [4]. The proposed parallel algorithm is primary based on a binary approach. The MSB (Most Significant Bit) of each randomly generated number will indicate the allocation bucket. Upon end, each bucket is sorted apart using the Bubble sort algorithm. As for the parallel design, each slave node will be responsible for one bucket to sort. In case of having the number of slaves less than the number of buckets, each slave will then handle more than one bucket at the same time. Eventually, the master node displays the results as a single sorted list of digits. The execution time of the proposed parallel algorithm is recorded and reported by the master node.
```
A. Implementation & Experiments
The proposed algorithm is implemented under MS Visual C# 2015 and the MS .NET Framework 3.5. The message passing interface used is the proprietary MPI.NET SDK. As a testing platform, a single computer has been used with Intel Core Dual Core 1.66Ghz CPU and 512MB of DDR2 RAM. Table 2 delineates the results obtained

<table>
<thead>
<tr>
<th>Table 2: Bucket Sort Testing Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of iterations</td>
</tr>
<tr>
<td>Sequential execution time</td>
</tr>
<tr>
<td>Parallel execution time</td>
</tr>
<tr>
<td>Speedup factor t_s / t_p = 10437/3875 = 2.7</td>
</tr>
</tbody>
</table>

B. Source Code
private void Start()
{
    // Generate Random Numbers to SORT
    Random rand = new Random();
    int[] list = new int[30000];
    for (int i = 0; i < list.Length; i++)
    {
        list[i] = rand.Next(0, 255);
        BucketSort(list);
    }
}
public void BucketSort(int[] list)
{
    // requires 3-bits
    Array[] buckets = new Array[8]; // 8 buckets ->
    for (int i = 0; i < buckets.Length; i++)
    {
        buckets[i] = new Array(); // create object
    }
    DateTime t1 = DateTime.Now; // Start Time
    for (int i = 0; i < list.Length; i++)
    {
        string number = ConvertToBinary(list[i]);
        string MSB = number.Substring(0, 3) // taking the
        int integer = ConvertToDecimal(MSB);
        buckets[integer].Add(list[i]); // add number to the corresponding bucket
    }
    // Update GUI Labels with numbers
    for (int i = 0; i < buckets.Length; i++)
    {
        label7.Text = label7.Text + buckets[i].ToString() + ",";
    }
    // At this point all BUCKETS are filled with numbers
    string[] args = null;
    using (new MPI.Environment(ref args))
    {
        Communicator comm = Communicator.world;
        if (comm.rank == 0) // MASTER
        {
            this.Text = "MASTER"; // Set TitleBar
            string sortedList = ";
            // send the first 4 buckets to the slave
            comm.Send(buckets[0], 1, i);
            // send to RANK 1 with TAG i+1
            // SORT bucket #5 to bucket #8
            for (int i = 4; i < buckets.Length; i++)
            {
                sortedList = sortedList + BubbleSort(buckets[i]);
            }
            outputTextBox.Text = comm.Receive<string>(1, 5) + sortedList;
            DateTime t2 = DateTime.Now; // Stop Time
            TimeSpan duration = t2 - t1;
            timeLabel.Text = "Time: " + duration.Seconds + "," + duration.Milliseconds + "ms"
        } else // SLAVE
        {
            this.Text = "SLAVE"; // Set TitleBar
            string sortedList = ";
            Array[] buckets_SLAVE = new Array[4];
            for (int i = 0; i < buckets_SLAVE.Length; i++)
            {
                buckets_SLAVE[i] = comm.Receive<Array>(0, i);
                sortedList = sortedList + BubbleSort(buckets_SLAVE[i]);
            }
            comm.Send(sortedList, 0, 5);
        }
    }
}
private string BubbleSort(Array[] bucket)
{
    // Bubble Sort
    // converting Array[] object to a regular int array
    int[] array = new int[bucket.Count];
    for (int i = 0; i < bucket.Count; i++)
    {
        array[i] = Convert.ToInt32(bucket[i].ToString());
    }
    int temp;
    for (int i = 0; i < array.Length; i++)
    {
        for (int j = 0; j < array.Length; j++)
        {
            if (array[i] < array[j])
            {
                temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }
}
// Displaying the sorted numbers
string sortedList = ""
for (int i = 0; i < array.Length; i++)
    sortedList = sortedList + array[i] + ",";
return sortedList;
}

III. MONTE CARLO ALGORITHM
The Monte Carlo is a computational algorithm that relies on repeated random sampling to compute its results [5]. Monte Carlo methods are often used when simulating physical and mathematical systems. Because of their reliance on repeated computation and random or pseudo-random numbers, Monte Carlo methods are most suited to calculation by a computer. In this problem, we are using the Monte Carlo method to estimate to value of Pi.

The proposed algorithm is mainly a parallel implementation of the renowned Monte Carlo problem. Since there are a maximum number of iterations after which the algorithm should stop, it is natural to partition the number of iterations per singular nodes. In this sense, each node including the master node will be responsible for a specific number of iterations less than the total maximum of iterations. Finally, the master will collect back the results and display the final value of Pi.

A. Implementation & Experiments
The proposed algorithm is implemented under MS Visual C# 2015 and the MS .NET Framework 3.5. The message passing interface used is the proprietary MPI.NET SDK. As a testing platform, a single computer has been used with Intel Core Dual Core 1.66Ghz CPU and 512MB of DDR2 RAM. Table 3 delineates the results obtained.

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>50000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential execution time</td>
<td>7s 359ms</td>
</tr>
<tr>
<td>Parallel execution time</td>
<td>3s 890ms</td>
</tr>
<tr>
<td>Speedup factor</td>
<td>( t_s / t_p = 7359/3890 = 1.9 )</td>
</tr>
</tbody>
</table>

B. Source Code
private void Start()
{
    Random rand = new Random();
    string[] args = null;
    using (new MPI.Environment(ref args))
    {
        Communicator comm = Communicator.world;
        if (comm.Rank == 0) // MASTER
        {
            this.Text = "MASTER";
            DateTime t1 = DateTime.Now; // Start Time
            comm.Send(max_iterations, 1, 0); // To RANK 1 with TAG 0
            double x, y, z, Pi;
            int count = 0;
            for (int i = 0; i < max_iterations/2; i++)
            {
               x = (double)rand.Next(32767) / 32767;
               y = (double)rand.Next(32767) / 32767;
               z = x * x + y * y;
               if (z <= 1) count++;
            }
            int countREC = comm.Receive<int>(1, 1); // From RANK 1 with TAG 1
            Pi = (double)(count+countREC) / max_iterations * 4;
            PiLabel.Text = "\(Pi = \) \pi + Pi;"
            DateTime t2 = DateTime.Now; // Stop Time
            TimeSpan duration = t2 - t1;
            timeLabel.Text = "Time: " + duration.Seconds + " s " +
                              duration.Milliseconds + "ms"
        }
        else // SLAVE
        {
            this.Text = "SLAVE";
            int max_iterationsREC = comm.Receive<int>(0, 0);
            double x, y, z, Pi;
            int count = 0;
            for (int i = max_iterationsREC / 2; i <
                max_iterationsREC; i++)
            {
               x = (double)rand.Next(32767) / 32767;
               y = (double)rand.Next(32767) / 32767;
               z = x * x + y * y;
               if (z <= 1) count++;
            }
            comm.Send(count, 0, 1); // To RANK 0 with TAG 1
        }
    } //end of using STATEMENT
}

IV. GRAYSCALE IMAGE TRANSFORMATION
Digital Image Transformations are a fundamental part of computer graphics. Transformations are used to scale objects, to shape objects, and to position objects [6]. In this problem, we are converting a 24-bit colored image into an 8-bit grayscale image.

The proposed parallel algorithm will embarrassingly assign different regions of the picture to each of the available and active nodes. Each node will work on its dedicated part then the transformed pixels are sent back to the master node. The master node eventually displays the complete transformed image.

A. Implementation
The proposed algorithm is implemented under MS Visual C# 2015 and the MS .NET Framework 3.5. The message passing interface used is the proprietary MPI.NET SDK. As a testing platform, a single computer has been used with Intel Core Dual Core 1.66Ghz CPU and 512MB of DDR2 RAM. Table 4 delineates the results obtained
Table 4: Image Transformation Testing Results

<table>
<thead>
<tr>
<th>Image size</th>
<th>698x475 pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential exec.</td>
<td>0s 953ms</td>
</tr>
<tr>
<td>Parallel exec.</td>
<td>0s 718ms</td>
</tr>
<tr>
<td>Speedup factor</td>
<td>t_s / t_p = 953/718 = 1.3</td>
</tr>
</tbody>
</table>

Figure 2 depicts two transformed regions of the same image. The master nodes handled the left part; while, the slave nodes handled the right part.

![Figure 2: Grayscale Image Transformation Program](image)

B. Source Code

```csharp
private void Start()
{
    DateTime t1 = DateTime.Now; // Start time
    string[] args = null;
    using (new MPI.Environment(ref args))
    {
        Communicator comm = Communicator.world;
        if (comm.Rank == 0) // MASTER
        {
            Bitmap bitmap1 = new Bitmap(pictureBox1.Image, pictureBox1.Width, pictureBox1.Height);
            comm.Send(pictureBox1.Width / 2, comm.Rank, pictureBox1.Width / 2, pictureBox1.Height); // send to RANK 1

            for (int y = 0; y < pictureBox1.Height; y++)
            {
                for (int x = 0; x < pictureBox1.Width; x++)
                {
                    Color c = bitmap1.GetPixel(x, y);
                    int grayPixel = (int)(c.R * 0.3 + c.G * 0.59 + c.B * 0.11);

                    bitmap2.SetPixel(x, y, Color.FromArgb(grayPixel, grayPixel, grayPixel));
                }
            }

            pictureBox1.Image = (Image)bitmap1;

            DateTime t2 = DateTime.Now; // Stop time
            TimeSpan duration = t2 - t1;
            timeLabel.Text = "Time: " + duration.Seconds + " s " + duration.Milliseconds + " ms";
        }
        else // SLAVE
        {
            int width_Rec = comm.Receive<int>(0, 0);
            Bitmap bitmap2 = new Bitmap(pictureBox1.Image, pictureBox1.Width, pictureBox1.Height);

            for (int y = 0; y < bitmap2.Height; y++)
            {
                //Formula: grayPixel = 0.3*RED + 0.59*GREEN + 0.11*BLUE
                int grayPixel = (int)(c.R * 0.3 + c.G * 0.59 + c.B * 0.11);

                bitmap2.SetPixel(x, y, Color.FromArgb(grayPixel, grayPixel, grayPixel));
            }
        }
    }
}
```

V. ARRAY SUMMATION

The problem of array summation is to add together 5,000,000 numbers contained in a one-dimensional array [7]. The master node would broadcast the content of the initial array to all the available slaves. Each slave would then add together each two contiguous integers and send the partial sum back to the master node. After long run, the master node adds all those accumulated partial sums to get a final result.

### A. Implementation

The proposed algorithm is implemented under MS Visual C++ 6.0 [8]. The message passing interface used is the proprietary MPI 2.0 standard DeinoMPI [9]. As a testing platform, two computers connected by a 100Mbps Ethernet have been used with Intel Core Dual Core 1.66Ghz CPU and 512MB of DDR2 RAM. Table 5 delineates the results obtained.

<table>
<thead>
<tr>
<th>Number to add</th>
<th>5000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential execution time</td>
<td>1s 798ms</td>
</tr>
<tr>
<td>Parallel execution time</td>
<td>0s 323ms</td>
</tr>
<tr>
<td>Speedup factor</td>
<td>t_s / t_p = 1798/323 = 5.56</td>
</tr>
</tbody>
</table>

B. Source Code

```csharp
void main(int argc, char* argv[])
{
    int my_rank; // Holds my rank: 0 for master and other numbers for slaves
    int num_proc; // Holds the number of processors available
    MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);

    int partition_size = 5000000/num_proc; // Partition Numbers among processes

    if (my_rank == 0) // MASTER
    {
        int data[5000000] = ;
    }
```
Table 6: Parallel Insertion Sort Testing Results

<table>
<thead>
<tr>
<th>Number to sort</th>
<th>Speedup factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>t_s / t_p = 2203/1102 = 1.99</td>
</tr>
</tbody>
</table>

B. Source Code

```c
void main(int argc, char* argv[])
{
    int my_rank; // Holds my rank: 0 for master and other numbers for slaves
    int num_proc; // Holds the number of processors available

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
    int max=-1;

    if (my_rank == 0) // MASTER
    {
        int data[500] = 0; // List to sort
        int number;
        MPI_Recv(&data[0], 1, MPI_INT, my_rank+1, 0, &status);
        if (max <= 0) // 1st time
            max = number;
        else if (my_rank != num_proc-1)
        {
            if (number <= max)
                MPI_Send(&number, 1, MPI_INT, my_rank+1, 0);
            else
            {
                // send to Pi+1
                MPI_Send(&max, 1, MPI_INT, my_rank+1, 0);
                max = number;
            }
        }
    }
}
```

VI. INSERTION SORT ALGORITHM

Insertion sort is a simple sorting algorithm, it is a comparison sort in which the sorted array is built one entry at a time. In abstract terms, every iteration removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input [10].

In the proposed parallel algorithm, the master node will send the 1st input to slave node P, P will then check if the received number is smaller than a max value, if yes, it will send it to Pi+1, otherwise, it will send the max to Pi+1 and assign max a new value that is the number received. The algorithm is repeated until the whole list is sorted.

A. Implementation

The proposed algorithm is implemented under MS Visual C++ 6.0. The message passing interface used is the proprietary MPI 2.0 standard DeinoMPI. As a testing platform, two computers connected by a 100Mbps Ethernet have been used with Intel Core Dual Core 1.66Ghz CPU and 512MB of DDR2 RAM. Table 6 delineates the results obtained.
VII. CONCLUSIONS & FUTURE WORK
This paper presented several computing algorithms that were originally designed for single processing. These algorithms are respectively the Mandelbrot set, the Bucket Sort, the Monte Carlo, the Grayscale Image Transformation, the Array Summation, and the Insertion Sort algorithm. All these algorithms were redesigned to execute in a parallel computing environment namely distributed message passing systems. They were implemented using C# .NET, the MPI.NET SDK, and the DeinoMPI API. Experiments showed that the proposed parallel algorithms have a substantial speed-up in execution time by multitude of factors.

As future work, the proposed algorithms are to be rewritten for shared memory architectures making the use of multi-threading, multi-processor, and multi-core systems.

Acknowledgment
This research was funded by the Lebanese Association for Computational Sciences (LACSC), Beirut, Lebanon, under the "Parallel Programming Algorithms Research Project - PPARP2019."

References


