Implementation of Computational Algorithms using Parallel Programming

Youssef Bassil

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

How to cite this paper: Youssef Bassil “Implementation of Computational Algorithms using Parallel Programming” Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-3 | Issue-3, April 2019, pp.704-710, URL: http://www.ijtsrd.com/papers/ijtsrd22947.pdf

ABSTRACT

Parallel computing is a type of computation in which many processes 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

I. MANDELBROT SET ALGORITHM

The Mandelbrot set is a set of points in the complex plane, the boundary of which forms a fractal. Mathematically, the Mandelbrot set can be defined as the set of complex c-values for which the orbit of 0 under iteration of the complex quadratic polynomial $x_{n+1}=x_n^2+c$ remains bounded [1].

We have designed our parallel algorithm based on generic static assignment approach where each node in a cluster is responsible for a pre-defined set of points. The master will identify the number of available slaves and assign a number of points or pixels to each active slave. Each slave then will apply the Mandelbrot algorithm to decide whether or not a particular pixel belongs to the set. Ultimately results will be collected by the master node which will display graphically the set of pixels. The execution time of the 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 [2]. The message passing interface used is the proprietary MPI.NET SDK [3]. As a testing platform, a single computer has been used with Intel Core Dual Core 1.66Ghz CPU and 512MB of DDR2 RAM. Table 1 delineates the results obtained

| Table 1: Mandelbrot Testing Results |
|-----------------------------------|
| Number of iterations | 20000 |
| Sequential execution time | 18s 578ms |
| Parallel execution time | 8s 78ms |
| Speedup factor | $t_s / t_p = 18578/8078 = 2.3$ |

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_IMAG = -2; // FIXED
    double MAX_IMAG = 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;
        if (comm.Rank == 0) // MASTER
        {
            for (int i=0, z=1; z<num_proc; i=i+region+1, z++)
            {
                comm.Send( i , z, 0); // send the height_From to RANK z with TAG 0
                comm.Send( i+region , z, 1); // send the height_To to RANK z with TAG 1
            }
            for (int x = 0; x < width; x++) // x = x co-ordinate of pixel
            {
                for (int y = 0; y < height / 2; y++) // y = y co-ordinate of pixel
                {
                    complexReal = MIN_REAL + x * (MAX_REAL - MIN_REAL) / width;
                    complexImag = MIN_IMAG + y * (MAX_IMAG - MIN_IMAG) / height;
                    int iteration = cal_pixel(complexReal, complexImag);
                    if (iteration == max_iteration)
                        bitmap1.SetPixel(x, y, Color.BlueViolet);
                    else bitmap1.SetPixel(x, y, Color.Black);
                }
            }
            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_IMAG + y * (MAX_IMAG - MIN_IMAG) / height;
                    int iteration = cal_pixel(complexReal, complexImag);
                    if (iteration == max_iteration)
                        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 < max_iteration));
    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 primarily 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.

| Number of iterations | 30000 |
|-----------------------|-------|
| Sequential execution time | 10s 437ms |
| Parallel execution time  | 3s 875ms |
| Speedup factor  | $t_s / t_p = 10437/3875 = 2.7$ |

B. Source Code
```csharp
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)
{
    ArrayList[] buckets = new ArrayList[8]; // 8 buckets -> requires 3-bits
    for (int i = 0; i < buckets.Length; i++)
        buckets[i] = new ArrayList(); // create object
    DateTime t1 = DateTime.Now; // Start Time
    for (int i = 0; i < list.Length; i++)
    {
        string number = ConvertToBinary(list[i]); // taking the 3 MSBs
        string MSB = number.Substring(0, 3); // taking the 3 MSBs
        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[6].Count - 1; i++)
        label7.Text = label7.Text + buckets[6][i].ToString() + ", ";
    for (int i = 0; i < buckets[7].Count - 1; i++)
        label8.Text = label8.Text + buckets[7][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
            for (int i = 0; i < 4; ++i)
                comm.Send(buckets[i], 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 = 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 + "s " + duration.Milliseconds + "ms";
        }
        else // SLAVE
        {
            this.Text = "SLAVE"; // Set TitleBar
            string sortedList = "";
            ArrayList[] buckets_SLAVE = new ArrayList[4];
            for (int i = 0; i < buckets_SLAVE.Length; i++)
            {
                buckets_SLAVE[i] = comm.Receive<ArrayList>(0, i);
                sortedList = BubbleSort(buckets_SLAVE[i]);
                comm.Send(sortedList, 0, 5);
            }
        }
    }
}
```

private string BubbleSort(ArrayList bucket)
{
    // Bubble Sort
    // converting ArrayList 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;
            }
        }
    }
}
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 the 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 3: Bucket Sort Testing Results

| Number of iterations | 50000000 |
|----------------------|----------|
| Sequential execution time | 7s 359ms |
| Parallel execution time | 3s 890ms |
| Speedup factor | $t_s / t_p = 7359 / 3890 = 1.9$ |

B. Source Code

```csharp
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 = max_iterations / 2; i < max_iterations; 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;
            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
        }
    }
}
```

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

| Image size | 698x475 pixels |
|------------|----------------|
| Sequential execution time | 0s 953ms |
| Parallel execution time | 0s 718ms |
| Speedup factor | \( \frac{t_s}{t_p} = 953/718 = 1.3 \) |

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

**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, 1, 0); // send to RANK 1 with TAG 0
            for (int y = 0; y < bitmap1.Height; y++)
            {
                for (int x = 0; x < bitmap1.Width / 2; x++)
                {
                    Color c = bitmap1.GetPixel(x, 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);
                    bitmap1.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++)
            {
                for (int x = width_Rec; x < bitmap2.Width; x++)
                {
                    Color c = bitmap2.GetPixel(x, 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));
                }
            }
            pictureBox1.Image = (Image)bitmap2;
        }
    }
}
```

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 5: Pixel Summation Testing Results

| Number to add | 5000000 |
|---------------|---------|
| Sequential execution time | 1s 798ms |
| Parallel execution time | 0s 323ms |
| Speedup factor | \( \frac{t_s}{t_p} = 1798/323 = 5.56 \) |

**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_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] = ;
    }
    else // SLAVE
    {
        for (int i = 0; i < partition_size; i++)
        {
            MPI_Send(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        }
        MPI_Status status;
        MPI_Recv(&data[i], 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        int sum = 0;
        for (int i = 0; i < partition_size; i++)
        {
            sum += data[i];
        }
        MPI_Reduce(&sum, &data[0], 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    }
    if (my_rank == 0)
    {
        MPI_Reduce(&sum, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    }
}
```
for(int i=0 ; i<50000 ; i++)
data[i] = i;
clock_t t1 = clock();
MPI_Bcast(data , 50000 , MPI_INT , 0 ,
MPI_COMM_WORLD);
int sum=0 , partial_sum=0 , sumREC=0;
for(int k=0 ; k<partition_size ; k++)
    partial_sum = partial_sum + data[k];
for(int i=1 ; i<num_proc ; i++)
{
    MPI_Recv(&sumREC , 1 , MPI_INT , i , 0 ,
MPI_COMM_WORLD , &status);
    sum = partial_sum + sumREC;
}
clock_t t2 = clock();
cout<"Sum = "<<sum<<"\n";
cout<"\nTime elapsed: "<<(double)t2 - t1<" ms";
}
else // SLAVE
{
    int data[5000000] ;
    MPI_Bcast(data , 5000000 , MPI_INT , 0 ,
MPI_COMM_WORLD);
    int partial_sum ;
    for(int i=partition_size ; i<5000000 ; i++)
        partial_sum = partial_sum + data[i];
    MPI_Send(&partial_sum , 1 , MPI_INT , my_rank , 0 ,
MPI_COMM_WORLD);
}

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.

Table 6: Parallel Insertion Sort Testing Results

| Number to sort | 500 |
|----------------|-----|
| Sequential execution time | 2s 203ms |
| Parallel execution time | 1s 102ms |
| Speedup factor | \( t_s / t_p = 2203/1102 = 1.99\) |

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_Status status;
    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] = ; // List to sort
        for(int j=0 ; j<500 ; j++)
            data[j] = (int)rand();
        clock_t t1 = clock();
        for(int i=0 ; i<500 ; i++)
            MPI_Send(&data[i] , 1 , MPI_INT , my_rank+1 , 0);
        clock_t t2 = clock();
        cout<"\nTime elapsed: "<<(double)t2 - t1<" ms";
    }
    else // SLAVE
    {
        int number;
        MPI_Recv(&number , 1 , MPI_INT , my_rank-1 , 0,
MPI_COMM_WORLD);
        if(max==-1) // 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;
                
            
        
    
} 
```
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

[1] Mitsuhiro Shishikura, "The Hausdorff dimension of the boundary of the Mandelbrot set and Julia sets", Annals of Mathematics, Second Series, vol. 147, no. 2, pp.225–267, 1998

[2] Charles Petzold, "Programming Microsoft Windows with C#", Microsoft Press, 2002

[3] Microsoft MPI, Microsoft implementation of the Message Passing Interface, URL: https://www.microsoft.com/en-us/download/details.aspx?id=57467, Retrieved on March 2019.

[4] Corwin, E., Logar, A. "Sorting in linear time - variations on the bucket sort", Journal of Computing Sciences in Colleges, vol. 20, no. 1, pp.197–202, 2004.

[5] Karger, David R., Stein, Clifford, "A New Approach to the Minimum Cut Problem", Journal ACM, vol. 43, no. 4, pp.601–640, 1996

[6] Rafael C. Gonzalez, Richard E. Woods, "Digital Image Processing", 3rd Edition, Prentice Hall, 2007.

[7] Maria Petrou, Costas Petrou, "Image Processing: The Fundamentals", 2nd edition, Wiley, 2010

[8] David Kruglinski, George Shepherd, Scot Wingo, "Programming Microsoft Visual C++", Microsoft Press, 5th edition, ISBN-13: 9781572318571, 1998

[9] DeinoMPI, Implementation of MPI-2 for Microsoft Windows, URL: http://mpi.deino.net/, Retrieved on March 2019

[10] Donald Knuth, "5.2.1: Sorting by Insertion, The Art of Computer Programming, Sorting and Searching", 2nd edition, Addison-Wesley, ISBN: 0201896850, 1998