Chapter - 4

Complexity and Performance Analysis of Algorithms of Interpolation Methods

CHAPTER - 4

Complexity and Performance Analysis of Algorithms of Interpolation Methods

4.1 Introduction

The word “interpolation” refers to interpolating some unknown information from a given set of known information. The technique of interpolation is widely used as a valuable tool in science and engineering. The problem is a classical one and dates back to the time of Newton and Kepler, who needed to solve such a problem in analyzing data on the position of stars and planets. Mathematical applications of interpolation include derivation of computational techniques for numerical differentiation, numerical integration and numerical solutions of differential equations.

There is a rich history behind interpolation. It really begin with the early studies of astronomy when the motion of heavily bodies was determined from periodic observations. The name of many famous mathematician are associated with interpolation: Gauss, Newton, Besel and Striling. More than three hundred years have passed since a procedure for solving an algebraic equation was proposed by Newton (1669) and later by Raphson (1690) Johnson and Bicanic (1979). The method is now called Newton's method or the Newton Raphson method and is still a central technique for solving nonlinear equations. From very ancient time Interpolation is being used for various purposes. Whittaker (1913), observed the most common form of interpolation occurs when we seek data from a table which does not have the exact values we want. Jianping et al. (2006) used the equivalent of second order Gregory- Newton interpolation to construct an “Imperial Standard Calendar”. In 625 AD, Indian astronomer and mathematician Brahmagupta introduced a method for second order interpolation of the sine function and, later on, a method for interpolation of unequal-interval data. Numerous researchers study the possibility of Interpolation based on the Fourier transform, the Hartley transform and the discrete cosine transform. Parker et al. (1983) published a first comparison of Interpolation techniques in medical image processing. They failed, however to

implement cubic B-spline interpolation correctly and arrive at erroneous conclusions concerning this technique Meijering and Erik. (2002). In present days, several algorithms are used for image resizing Jianping et al. (2006) based on Newton's Interpolation Formula Steven and Raymond (2003). In this chapter we will study the complexity and performance analysis of the algorithms using OpenMP and found that scattered data can be interpolated to estimate values at uniformly positioned grid point

4.1.1. Programming in OpenMP

An OpenMP Application Programming Interface (API) was developed to enable shared memory parallel programming. OpenMP API is the set of compiler directives, library routines, and environment variables to specify shared-memory parallelism in FORTRAN and C/C++ programs Barbara et al (2008). It provides three kinds of directives: parallel work sharing, data environment and synchronization to exploit the multi-core, multithreaded processors. The OpenMP provides means for the programmer to: create teams of thread for parallel execution, specify how to share work among the member of team, declare both shared and private variables, and synchronize threads and enable them to perform certain operations exclusively OpenMP is based on the fork-and-join execution model, where a program is initialized as a single thread named master thread. This thread is executed sequentially until the first parallel construct is encountered. This construct defines a parallel section (a block which can be executed by a number of threads in parallel). Therfor the master thread creates a team of threads that executes the statements concurrently in parallel contained in the parallel section. There is an implicit synchronization at the end of the parallel region, after which only the master thread continues its execution.

4.1.2. Performance of Parallel Algorithm

The amount of performance benefit an application will realize by using OpenMP depends entirely on the extent to which it can be parallelized. Amdahl's law specifies the maximum speed-up that can be expected by parallelizing portions of a serial program Quinn (2004). Essentially, it states that the maximum speed up (S) of a program is

$$P_n(\alpha + nh) = f(\alpha + nh).....P_n(\alpha) = f(\alpha)$$

Where, F is the fraction of the total serial execution time taken by the portion of code that can be parallelized and N is the number of processors over which the parallel portion of the code runs. The metric that have been used to evaluate the performance of the parallel algorithm is the speedup Quinn (2004). It is defined as:

$$\text{Sp} = \frac{T_1}{T_p}$$

Where, T_1 denotes the execution time of the best known sequential algorithm on single processor machine, and T_p is the execution time of parallel algorithm .In other word, speedup refers to how much the parallel algorithm is faster than the sequential algorithm.

4.1.3. Proposed Methodology

Calculation: to find the f(x) (Interpolation value) for given value of x of various function using following methods:

(A) Newton-Gregory forward Interpolation formula

(B) Newton-Gregory formula for backward interpolation

(C) Divided difference

(D) Newton's formula for unequal intervals

(E) Lagrangian Interpolation formula for unequal intervals

4.2 Lagrangian Polynomial Interpolation

It is very significant approach. The Lagrangian polynomial is the simplest way to exhibit the existence of a polynomial with unevenly spaced data. Suppose we have a table of data with four pairs of x_i and $f(x_i)$ values with xi indexed by variable i:

i	x_i	$f(x_i)$
0	x_0	f_0
1	x_1	f_1
2	x_2	f_2
3	x_3	f_3

The x- value must all be distinct, however. The Lagrangian form for this is

$$P_3(x) = \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} f_0 + \frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)} f_1 +$$
$$\frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)} f_2 + \frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)} f_3$$

And the error form of $g(x)$ is now apparent:

$$g(x) = \frac{f(n+1)(^{\xi})}{(n+1)!} \quad \text{Where } \xi \text{ between } \alpha, \alpha+h, \alpha+2h.....\alpha+nh.$$

4.2.1 An Algorithm for Interpolation from Lagrange Polynomial

Given a set of $n+1$ points $[(x_i, y_i)\, i = 0,1.....n]$ and a value for x at which the polynomial is to be evaluated;

1. Set sum=0
2. For i=0 to n step 1 do
3. Set p=1
4. For j=0 to n step 1 do

   If i<> j then

   Set $p = \frac{p \cdot (x - x_j)}{(x_i - x_j)}$

   End if

   End do j
5. Set *sum = sum + p · f*

   End do i
6. Print interpolated value is sum

   End

4.2.2. Counting Primitive Operation

1. Assigning the value 0 to sum contributes one unit of count.
2. The body of loop executes n times with count:
   i. Assigning the value 1 to p contributes one unit of count.
   ii. The body of loop executes n times with count:

   To compare i with j contributes one unit

Set p contributes five unit to count.

=6n

iii. Set sum contributes three unit of operation

=n (1+6n+3)

= $4n + 6n^2$

3. Write sum contributes one unit

To summarize, the number of primitive operations t (n) executed by algorithm is at least.

$$t(n) = 6n^2 + 4n + 2$$

So the complexity of algorithm used for Langarian Interpolation formula is method is O (n^2).

```
// Lnagrange_inter.cpp: Defines the entry point for the console application.
Visual C++ program for Lagrangian Interpolation to find the value of f(3) for given set of values.
#include "stdafx.h"
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>
#include<conio.h>
#define MaxN 90
int _tmain(int argc, _TCHAR* argv[])
{
          float arr_x[MaxN+1], arr_y[MaxN+1], numerator, denominator, x, y=0;
 int i, j, n;
 double time_begin;
 double time_elapsed;
 double time_stop;
 printf("Enter the value of n: \n");
 scanf_s("%d", &n);
 printf("Enter the values of x and y: \n");
```

```
  printf ( "  Number of processors available = %d\n",       omp_get_num_procs ( ) );
  printf ( "  Number of threads = %d\n", omp_get_max_threads ( ) );
time_begin = omp_get_wtime ( );
//#pragma omp parallel for
  for(i=0; i<=n; i++)
  scanf_s("%f%f", &arr_x[i], &arr_y[i]);
  printf("Enter the value of x at which value of y is to be calculated: ");
  scanf_s("%f", &x);
for (i=0; i<=n; i++)
  {
            numerator=1;
            denominator=1;
            for (j=0; j<=n; j++)
                        if(j!=i)
                        {
numerator *= x-arr_x[j];
                                    denominator *= arr_x[i]-arr_x[j];
  }
            y+=(numerator/denominator)*arr_y[i];
  }
  printf("When x=%4.1f y=%7.1f\n",x,y);
  time_stop = omp_get_wtime ( );
time_elapsed = time_stop - time_begin;
printf ( "  Elapsed time dT = %lf\n", time_elapsed );
  return 0;
}
```

Example: given that f(0)=8, f(1)=68 and f(5)=123, determine f(x) for x=1.5,2, 2.5, 3., 3.5, .4. and 4.5 and measure execution time using sequential and parallel execution.

Input:

0 8

1 68

5 123

Output:

Table-4.1: Performance analysis in sequential and parallel algorithm for Langrangian Method

S. No	For given value of x	F(x)	Sequential Execution time Time in ms	Parallel Execution time Time in ms	Performance/ Speed Up(s) Difference
1.	1.5	91.06250	12.311247	9.991232	1.232205
2.	2	109.5	10.926843	8.004243	1.365131
3.	2.5	123.0	10.549568	9.053898	1.165196
4.	3	132.5	12.793176	8.686153	1.472824
5.	3.5	137.0625	13.414446	8.109880	1.654087
6.	4	137.0	12.556711	10.12	1.240782
7.	4.5	132.32500	12.929047	6.571	1.967592

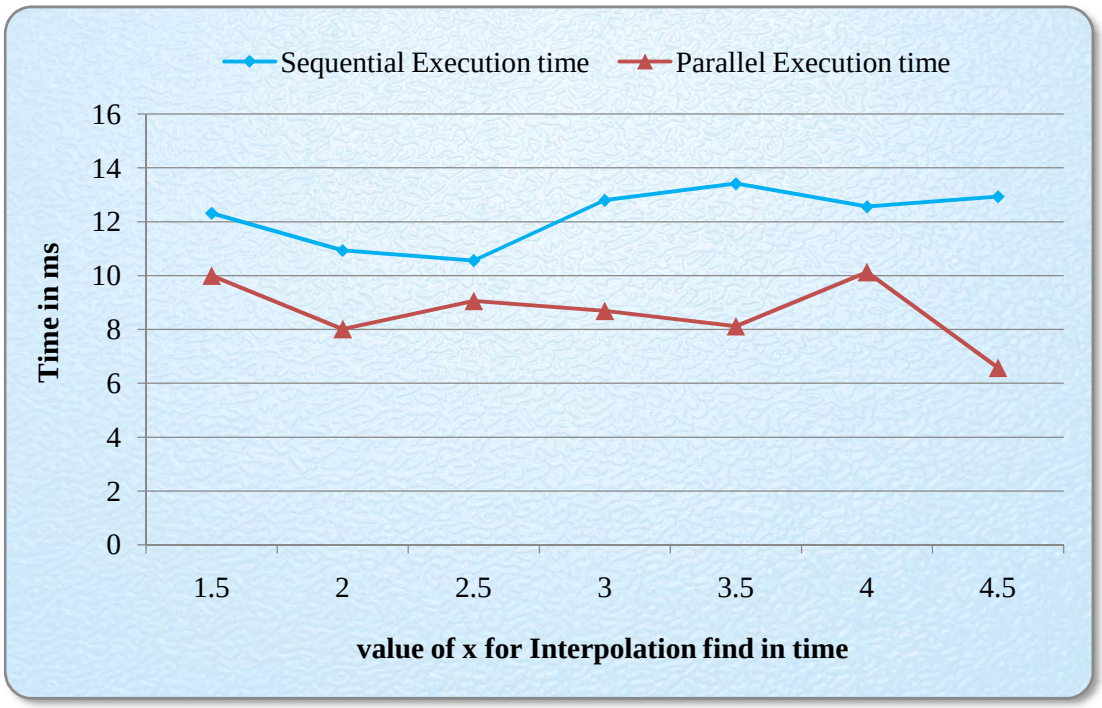

Fig 4.1: Performance analysis in sequential and parallel algorithm for Langrangian Method

4.3 Divided difference

Assume that the function, f(x), is known at several values for x:

x_0	f_0
x_1	f_1
x_2	f_2
x_3	f_3

Consider the nth degree polynomial written a special way:

$$P_n(x) = a_0 + (x - x_0)a_1 + (x - x_0)(x - x_1)a_2 + ... + (x - x_0)(x - x_{n-1})a_n .$$

If we chose a_i so that $P_n = f(x)$ at the n+1 known points $(x_i, f_i), i = 0,1.....n$ then $P_n(x)$ is an interpolating polynomial. A special standard notation for divided differences is $f[x_0, x_1] = \frac{f_1 - f_0}{x_1 - x_o}$ called the first divided differences between x_0 and x_1. The function $f[x_1, x_2] =$ is first divided difference between x_1 and x_2. In general $f[x_s, x_t] = \frac{ft - fs}{xt - xs}$

Second and higher order differences are defined in term of lower order differences. For example:

$$f[x_0, x_{1.....}x_n] = \frac{f[x_1, x_2...x_n] - f[x_0, x_1.....x_{n-1}]}{x_n - x_0}$$

4.3.1 An Algorithm for Interpolation from divided difference table

Give a set of n+1 points $[(x_i, f_i), i = 0,1.....n]$ and a value x=u at which interpolating polynomial is to be evaluated. We first find the coefficients of the interpolating polynomial. These are stored in vector dd.

1. For i=0 to n step 1

   Set $dd[i] = f[i]$.

   End for i

2. For j=1 to n step 1 do

   a. Set $temp1 = dd[j - 1]$.

b. For k=j to n step 1 do

i. Set $temp2 = dd[k]$.

ii. Set $dd[k] = \frac{(f[k] - temp1)}{(x[k] - x[k-j])}$

iii. Set Temp1=temp2.

End for k

End for j

3. Set sum=0

4. For i= n down to 1 step -1 do

(a) Set $sum = (sum - dd[i]) \cdot (u - x[i-1])$

(b) Set $sum = sum + dd[0]$.

End for i

5. Set $ddvalue = sum$.

6. Print $ddvalue$

7. End

4.3.2. Counting Primitive Operation

1. The body of loop executes n+1 times with count:

Set $dd[i] = f[i]]$ contributes one unit to count

= 1(n+1).

= n+1.

2. The body of loop executes n times with count:

i. Assigning the value to temp1 contributes one unit of count.

ii. The body of loop executes n times with count:

(a) Assigning value to temp2 contributes one unit

(b) Assigning the value to $dd[k]$ contributes four unit to count.

Total a+b= $5n$

Total i+ii= $n(1+5n)$

= $n + 5n^2$

iii. Set Temp1 = Temp2 contributes one operation

Total (i+ii+iii) $= n + 5n^2 + n$

$= 5n^2 + 2n$

3. Assign the value 0 to sum contributes one unit to count.
4. The body of loop executes n time
   i. Set sum contributes five units to count.
   ii. Set sum contributes two unit to count.

   Total (i+ii) = 7n
5. Write ddvalue contributes one unit to count.

To summarize, the number of primitive operations t (n) executed by algorithm is at least.

$$t(n) = n + 1 + 2n + 5n^2 + 1 + 7n + 1$$
$$= 5n^2 + 10n + 3$$

So the complexity of algorithm used for divided difference Interpolation from divided difference formula is method is $O(n^2)$

Visual C++ program for Divide Difference Formula for Interpolation to find the value of f(3) for given set of values.

```
#include "stdafx.h"
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>
int _tmain(int argc, _TCHAR* argv[])
{
float ddvalue,x[100],f[100],dd[100],temp1,temp2,sum=0,u;
int n,i,j,k;
double time_begin;
  double time_elapsed;
  double time_stop;
printf("Enter n");
scanf_s("%d",&n);
printf ( "  Number of processors available = %d\n", omp_get_num_procs ( ) );
  printf ( "  Number of threads =              %d\n", omp_get_max_threads ( ) );
time_begin = omp_get_wtime ( );
#pragma omp parallel for
```

```
for(i=0;i<=n;++i)
{
printf("enter x anf f\n");
scanf_s("%f %f",&x[i],&f[i]);
}
printf("Enter the value of x for which find the value");
scanf_s("%f",&u);
for(i=0;i<=n;++i)
{
dd[i]=f[i];
}
for(j=1;j<=n;++j)
{
temp1=dd[j-1];
for(k=j;k<=n;++k)
        {
        temp2=dd[k];
        dd[k]=(dd[k]-temp1)/(x[k]-x[k-j]);
        temp1=temp2;
        }
}
sum=0;
for(i=n-1;i>=1;--i)
sum=(sum+dd[i])*(u-x[i-1]);
sum=sum+dd[0];
ddvalue=sum;
printf("Interpolation is %f for %f",ddvalue,u);
time_stop = omp_get_wtime ( );
time_elapsed = time_stop - time_begin;
printf ( "  Elapsed time dT = %lf\n", time_elapsed );
return 0;
}
```

Example: Given the valuea and f(x) are

x	f(x)
3.2	22.0
2.7	17.8
1.0	14.2
4.8	38.3
5.6	51.7

Find the value of f(x) and execution in sequential and paprallel mode for following value of x: 2, 3, 4, and 5,6,7,8

OUTPUT: as per following table:

Table-4.2 Performance comparison of Serial and Parallel Algorithm for Interpolation for divided difference Method

S. No	For given value of x	F(x)	Sequential Execution time Time in ms	Parallel Execution Time in ms.	Performance/ Speed Up(s) Error/Difference
1.	2	13.875631	27.869010	26.728331	1.042677
2.	3	20.211960	27.661341	21.834969	1.266837
3.	4	30.044102	24.754409	21.630261	1.144434
4.	5	40.207169	22.076652	21.058515	1.048348

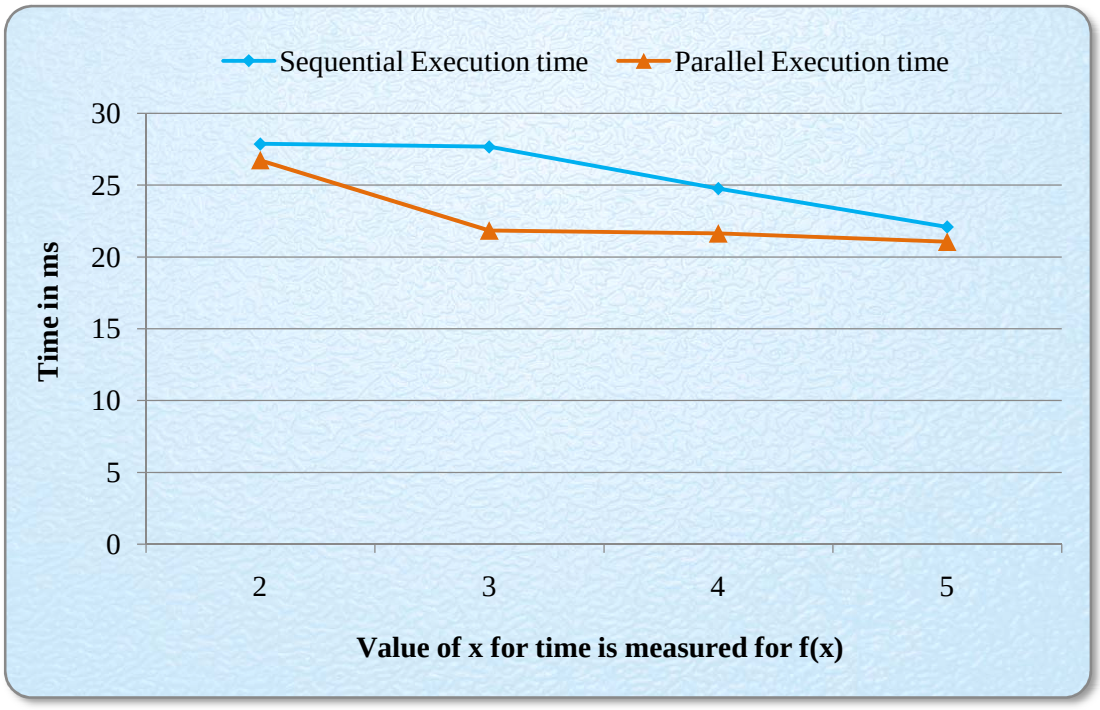

Fig 4.2: Performance analysis in sequential and parallel algorithm for divided difference Method

4.4 Newton-Gregory forward Interpolation formula

Let y= $f(x)$ represents a function which assumes the value $f(\alpha), f(\alpha+h), f(\alpha+2h).....f(\alpha+nh)$ for n+1 equidistant values α, α +h, α +2h…… $\alpha+nh$ of the independent variable x an let $P_n(x)$ be a polynomial in x of degree n. This polynomial may be written in the form:

$$P_n(x) = f(\alpha) + \frac{\Delta f(\alpha)}{h(x-\alpha)} + \frac{\Delta^2 f(\alpha)}{2!h^2(x-\alpha)(x-\alpha-h)} +$$

$$\frac{\Delta^3 f(\alpha)}{3!h^3(x-\alpha)(x-\alpha-h)(x-\alpha-2h)} + ... \frac{\Delta^n f(\alpha)}{n!h^n(x-\alpha)(x-\alpha-h).....(x-\alpha-nh)}$$

.....4.4.1

This is called Newton-Gregory formula for forward interpolation. This formula is used mainly for interpolating the values of y near of a set of tabulated values.

4.4.1 Algorithm for Newton-Gregory formula for forward interpolation

Using two dimension arrays named x and y each of size n are used to store the set of values for the function f(x) and two dimensional array named d of size (n-1)(n-1) is used to store the forward difference table

1. Read n, x
2. For i= 1 to n step 1

   Read x_i, y_i .

   End for i
3. If $(x < x_1)$) or $(x > x_0)$ then Write: "Value lies outside range and exit
4. Set i=2
5. While $(x < x_1)$ do

   Set i=i+1

   End while
6. Set k=i-1
7. Set $u = \frac{(x-x_k)}{(x_{k+1}-x_k)}$
8. For j= 1 to n-1

   For i=1 to n-j

If (j==1) then

Set $d_{ij} = y_{i+1} - y_i$.

Else

Set $d_{ij} = d_{(i+1)(j-1)} - d_{i(j-1)}$.

End if

End for i

End for j

9. Set $sum = y_k$.

10. For i=1 to n-k

Set prod=1

For i=1 to (n-k)

Set prod=prod · (u-j)

End for

Set $sum = \frac{sum + (d_{ki} \cdot prod)}{m}$

11. Print sum

12. End

4.4.2. Counting Primitive Operation

1. Read n, x contributes two unit of operation.
2. The body of loop executes n times with count:

   Read x_i, y_i contributes two unit of operations

   =2n
3. Comparison of x contributes three unit of operations.
4. Set i=2 contributes one unit of operation.
5. While loop contributes three unit of operations
6. Set k=i-1 contributes two unit of operations
7. Set $u = \frac{(x - x_k)}{(x_{k+1} - x_k)}$ contributes four unit of operations
8. The body of loop executes n-1 times with count:
   i. The body of loop executes n-j times with count:
      (a) Compare j and set d_{ij} contributes five operations

= (n-1) (n-j) 3.

$=3n^2-3nj-3-3j$.

9. Set $sum = y_k$ contributes one unit of operations.

10. The body of loop executes n-k times with count:

    i. set prod=1 contributes one unit of operations

    =1. (n-k)

    The body of loop executes n-k times with count:

    ii Set prod contributes three unit of operations.

    =3(n-k)

    Total (i+ii) $=3(n^2-2nk+k^2)$

    $=3n^2-6nk+3k^2$

11. Set sum contributes four unit of operations

12. Print sum contributes one unit of operations

To summarize, the number of primitive operations t (n) executed by algorithm is at least.

$t(n)=2+2n+3+1+3+2+4+3n^2-3nj-3j-3+1+3n^2-6nk+3k^2+4+1$

$=8n^2+n(2-5j-6k)+16$

So the complexity of algorithm used for Newton-Greogary forward interpolation is O (n^2).

Visual C++ Program for Newton-Greogary forward Interpolation formula:

```
// newtons Gragory forward interpolation for equal values
#include "stdafx.h"
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>
#include<conio.h>
int _tmain(int argc, _TCHAR* argv[])
{
 int i,j,k,n;
 float x[10],y[10],d[10][10],X,u,h,fact=1.0,sum=0.0,prod=0.0;
```

```
 double time_begin;
 double time_elapsed;
 double time_stop;
 printf("Enter the size:\t");
 scanf_s("%d",&n);
 printf("enter the value of X:\t");
 scanf_s("%f",&X);
 for(i=1;i<=n;i++)
   {
      printf("Enter x[%d]:\t",i);
     scanf_s("%f",&x[i]);
     printf("Enter y[%d]:\t",i);
     scanf_s("%f",&y[i]);
    }
i=2;
time_begin = omp_get_wtime ( );
#pragma omp parallel for
 while(X>x[i])
 i++;
 k=i-1;
u=(X-x[k])/(x[k+1]-x[k]);
 for(j=1;j<=(n-1);j++)
 {
                    for(i=1;i<=n-j;i++)
                    {
                     if(j==1)
                              d[i][j]=y[i+1]-y[i];
                    else
                              d[i][j]=d[i+1][j-1]-d[i][j-1];
                    }
}
printf("Backward difference Table:\n");
for(i=1;i<=n;i++)
{
                    for(j=1;j<n;j++)
                    {
```

```
                        if(i==1)
 printf(" \t");
 else if(j<i)
 printf("%f\t",d[i][j]);
 else
                        printf(" \t");
}
 printf("\n");
 }
sum=y[k];
for(i=1;i<=(n-k);i++)
 {
  prod=1.0;
  for(j=0;j<=(i-1);j++)
          prod=prod*(u-j);
fact=1;
 for(j=1;j<=i;j++)
 f       act=fact*j;
sum=sum+((prod*d[k][i])/fact);
  }
printf("Integral value=%f",sum);
time_stop = omp_get_wtime ( );
time_elapsed = time_stop - time_begin;
printf ( "  Elapsed time dT = %lf\n", time_elapsed );
        return 0;
}
```

Above program find the value of f(x) for unknown x and time execution for serial and parallel execution, where a set of values are given as:

x	f(x)
20	.01427
24	.01581
28	.01772
32	.01996

The result for following value of x are in given table:

Table 4.3: Performance analysis in sequential and parallel algorithm for Newton-Greogary forward Interpolation formula

S. No	For given value of x	F(x)	Sequential Execution time Time in ms	Parallel Execution time Time in ms	Performance/ Speed Up(s) Difference
1.	21	0.014618	.006119	.005597	1.093264
2.	22	0.014991	.005989	.005588	1.071761
3.	25	0.017510	.006155	.006153	1.000325
4.	26	0.016724	.006150	.005831	1.054708
5.	27	0.017212	.005656	.005178	1.092314
6.	29	0.016847	.005694	.005503	1.034708
7.	30	0.023840	.006146	.006109	1.006057

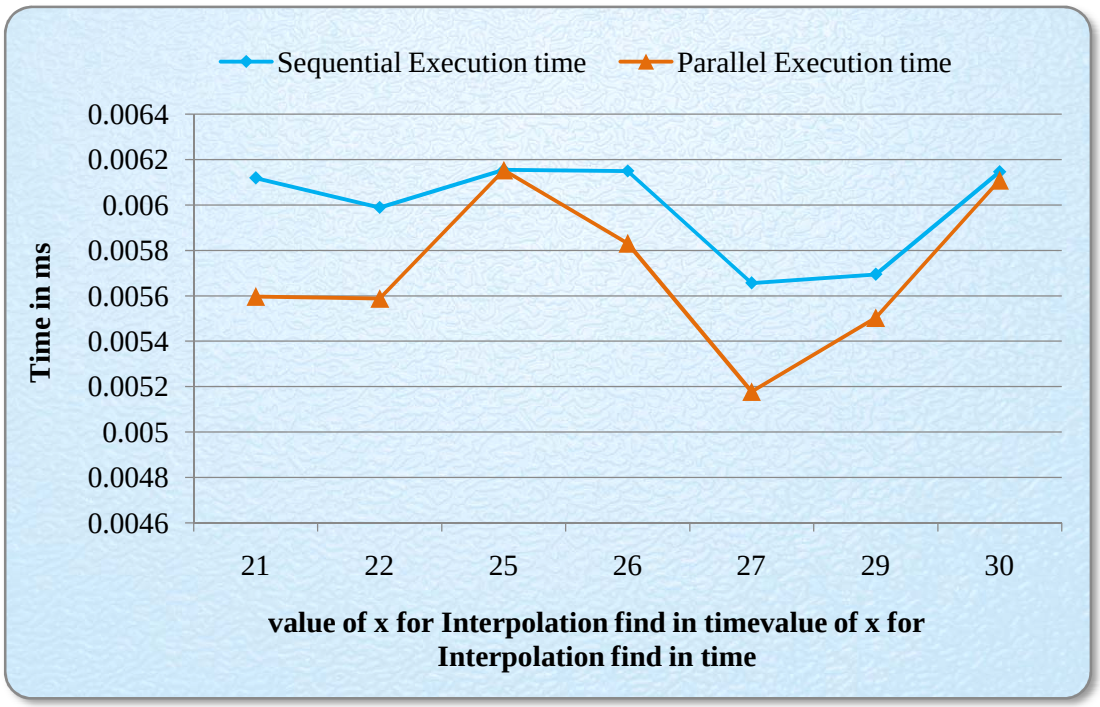

Fig4.3: Performance analysis in sequential and parallel algorithm for Newton-Greogary forward Interpolation formula

4.5 Newton-Gregory formula for backward interpolation

$$P_n(x) = f(\alpha + nh) + \frac{\Delta f(\alpha + nh)}{h \cdot (x - \alpha + nh)} ..... +$$

$$\frac{\Delta^2 f(\alpha + nh)}{n! h^n \cdot (x - \alpha + nh)(x - x - nh - h)...(x - x - h)}$$

is Newton-Gregory formula for backward interpolation.

4.5.1 Algorithm for Newton-Gregory formula for backward interpolation

Using two dimension arrays named x and y each of size n are used to store the set of values for the function $f(x)$ and two dimensional array named d of size (n-1) · (n-1) is used to store the forward difference table

1. Read n, x
2. For i= 1 to n step 1

   Read x_i, y_i

   End for i
3. If $(x < x_l)$ or $(x > x_0)$ then Write: "Value lies outside range and exit
4. Set i=2
5. While $(x < x_l)$ do

   Set i=i+1

   End while
6. Set k=i
7. Set $u = \frac{(x - x_k)}{(x_k - x_{k-1})}$
8. For j= 1 to n-1

   For i=1 to n-j

   If (j==1) then

   Set $d_{ij} = y_i - y_{i-1}$

   Else

   Set $d_{ij} = d_{i(j-1)} - d_{(i-1)(j-1)}$

   End if

End for i

End for j

9. Set $sum = y_k$

10. For i=1 to n-k

Set prod=1.0

For j=0 to (i-1)

Set $prod = prod \cdot (u + j)$

End for

Set $sum = \dfrac{sum + (d_{ki} \cdot prod)}{m}$

11. Write “Interpolation value” ,sum

4.5.2. Counting Primitive Operation

1. Read n, x contributes two unit of operation.

2. The body of loop executes n times with count:

Read x_i, y_i contributes two unit of operations

=2n

3. Comparison of x contributes three unit of operations.

4. Set i=2 contributes one unit of operation.

5. While loop contributes three unit of operations

6. Set k=i contributes one unit of operations

7. Set $u = \dfrac{(x - x_k)}{(x_k - x_{k-1})}$ contributes four unit of operations

8. The body of loop executes n-1 times with count:

   i. The body of loop executes n-j times with count:

      (a) Compare and set d_{ij} contributes three operations

      $= (n-1)(n-j) \cdot 3$

      $= 3n^2 - 3nj - 3j - 3$

9. Set sum= y_k contributes one unit of operations.

10. The body of loop executes n-k times with count:

   i. set prod=1 contributes one unit of operations

$= 1 \cdot (n-k)$

(a) The body of loop executes i times with count :

Set prod contributes three unit of operations.

$= 3 \cdot i.$

$= 3i \cdot n - 3i \cdot k$

11. Set sum contributes four unit of operations

12. Print sum contributes one unit of operations

To summarize, the number of primitive operations t (n) executed by algorithm is at least.

$$t(n) = 2 + 2n + 3 + 1 + 3 + 1 + 4 + 3n^2 - 3nj - 3j - 3 + 1 + 3in - 3ik + 4 + 1$$

$$t(n) = 5n^2 + n(2 - 5j - 3i) - 3ik + 17$$

So the complexity of algorithm used for Newton-Greogary Backward Interpolation from divided difference formula is method is O (n^2).

Program for Newton's Greogary Backward Interpolation formula

Visual C++ Program for Newton-Greogary Backward Interpolation formula:

```
#include "stdafx.h"
# include <stdlib.h>
# include <stdio.h>
# include <math.h>
# include <omp.h>
#include<conio.h>
int _tmain(int argc, _TCHAR* argv[])
{
{
 int i,j,k,n;
 float x[10],y[10],d[10][10],X,u,h,fact=1.0,sum=0.0,prod=0.0;
double time_begin;
 double time_elapsed;
 double time_stop
 clrscr();
 printf("Enter the size:\t");
 scanf("%d",&n);
```

```
 printf("enter the value of X:\t");
 scanf("%f",&X);
for(i=1;i<=n;i++)
 {
 printf("Enter x[%d]:\t",i);
 scanf("%f",&x[i]);
 printf("Enter y[%d]:\t",i);
 scanf("%f",&y[i]);
 }
i=2;
time_begin = omp_get_wtime ( );
#pragma omp parallel for
 while(X>x[i])
 i++;
 k=i;
u=(X-x[k])/(x[k]-x[k-1]);
 for(j=1;j<=(n-1);j++)
 {
          for(i=j+1;i<=n;i++)
           {
           if(j==1)
                    d[i][j]=y[i]-y[i-1];
          else
                    d[i][j]=d[i][j-1]-d[i-1][j-1];
          }
}
printf("Backward difference Table:\n");
for(i=1;i<=n;i++)
{
          for(j=1;j<n;j++)
          {
                    if(i==1)
 printf(" \t");
 else if(j<i)
```

```
 printf("%f\t",d[i][j]);
 else
 printf(" \t");
}
 printf("\n");
 }
sum=y[k];
for(i=1;i<=(n-1);i++)
 {
  prod=1.0;
  for(j=0;j<=(i-1);j++)
  prod=prod*(u+j);
fact=1;
 for(j=1;j<=i;j++)
 fact=fact*j;
sum=sum+((prod*d[k][i])/fact);
  }
printf("Integral value=%f",sum);
time_stop = omp_get_wtime ( );
time_elapsed = time_stop - time_begin;
printf ( "  Elapsed time dT = %lf\n", time_elapsed );
          return 0;
}
```

Above program find the value of f(x) for unknown x and time execution for serial and parallel execution, where a set of values are given as:

x	f(x)
20	.01427
24	.01581
28	.01772
32	.01996

The result for following value of x are in given table:

Table 4.4: Performance comparison of Serial and Parallel Algorithm for Newton-Greogary Backward Interpolation formula

S. No	For given value of x	F(x)	Sequential Execution time Time in ms	Parallel Execution time Time in ms	Performance/ Speed Up(s) Difference
1.	21	0.014655	.001584	.001236	1.281553
2.	22	0.015040	.001635	.001601	1.021237
3.	25	0.016254	.001678	.001501	1.117921
4.	26	0.015469	.001112	.001072	0.665072
5.	27	0.017208	.001608	.001604	1.002494
6.	29	0.018251	.002081	.001982	1.04995
7.	30	0.019371	.002396	.001240	1.932258

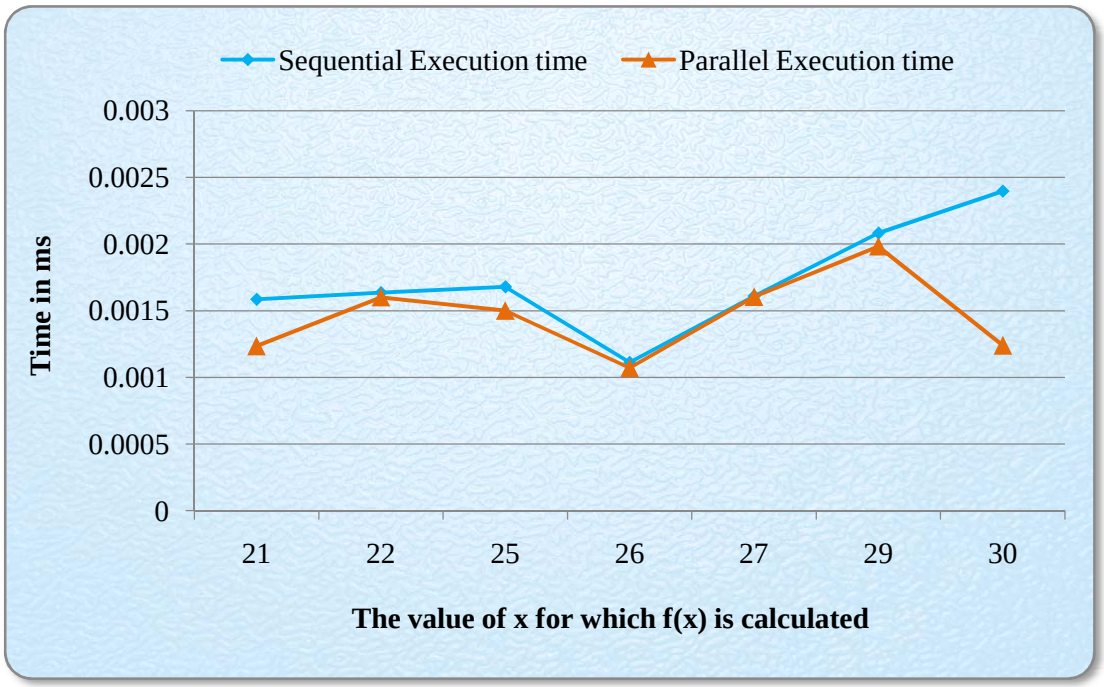

Fig 4.4: Performance analysis in sequential and parallel algorithm for Newton-Greogary Backward Interpolation formula

4.6 Interpolation with Unequal Intervals

(a) Newton's formula for unequal intervals

Let $f(x_o), f(x_1).....f(x_n)$ be the values of $f(x)$ corresponding to the arguments $x_0, x_1.....x_n$ not necessarily equally spaced. From the definition of divided differences

$$f(x, x_0) = \frac{f(x) - f(x_0)}{x - x_0} orf(x) = f(x_0) + (x - x_0) f(x, x_0).....(1)$$

Again

$$f(x, x_0, x_1) = \frac{f(x, x_0) - f(x_0, x_1)}{x - x_1} or \ f(x, x_0) = f(x_0, x_1) + (x - x_1) f(x, x_0, x_1).....(2)$$

Similarly

$$f(x, x_0, x_1) = f(x_0, x_1, x_2) + (x - x_2) f(x, x_0, x_1, x_2).....(3)$$

……. ……. …….

$$f(x, x_0, x_1.....x_{n-1}) = f(x_0, x_1, ...x_n) + (x - x_n) f(x, x_0, x_1, ..x_n). …….. (4)$$

Multiplying equation (2) by $(x - x_0)$, (3) by $(x - x_0)$ $(x - x_1)$ and so on finally equation (4) by $(x - x_0)$ $(x - x_1)$ ……. $(x - x_{n-1})$ and adding to equation (1) we obtain

$$f(x)=f(x_0)+(x-x_0)f(x_0,x_1)+(x-x_0)(x-x_1)f(x_0,x_1,x_2)+.....+(x-x_0)(x-x_1).....(x-x_{n-1})f(x_0,x_1,,,,,x_n)+R_n$$

Where the remainder R_n is given by

$$Rn = (x - x0)(x - x1)....(x - xn) f(x,, x0, x1.....xn)$$

If our function $f(x)$ is a polynomial of n^{th} degree, then $f(x,x_0,x_1.....x_n)$ vanishes, so that

$$f(x) = f(x_0) + (x - x_0) f(x_0, x_1) + (x - x_0)(x - x_1) f(x_0, x_1, x_2) + ... + (x - x_0)(x - x_1)....(x - x_{n-1}) f(x_0, x_1, ...x_n).....(5)$$

And

$$f(x) = f(x_0) + (x - x_0)\Delta f(x_0) + (x - x_0)(x - x_1)\Delta^2 f(x_0) + ... + (x - x_0)(x - x_1)....\Delta^n f(x_0)$$

$$f(x) = f(x_0) + \sum_{j=1}^{n} \prod_{i=1}^{j} (x - x_{i-1}) \underset{x1, x2, xn}{\overset{i}{\Delta}} f(x_0)$$

4.6.1 Algorithm for Newton's formula for unequal intervals

Using two dimension arrays named x and y each of size n are used to store the set of values for the function f(x) and two dimensional array named d of size (n-1)*(n-1) is used to store the divided difference table

1. Read n, x
2. For i= 1 to n step 1
   Read x_i, y_i
   End for i
3. If $(x < x_l)$ or $(x > x_0)$ then Write: "Value lies outside range and exit
4. Set i=2
5. While $(x < x_l)$ do
   Set i=i+1
   End while
6. Set k=i-1
7. For j= 1 to n-1
   For i=1 to n-j
   If (j==1) then
   Set $d_{ij} = \frac{(y_{i+1} - y_i)}{(x_{i+1} - x_i)}$
   Else
   Set $d_{ij} = \frac{d_{(i+1)(j+1)} - d_{i(j-1)}}{(x_{i+1} - x_i)}$
   End if
   End for i
   End for j
8. Set $sum = y_k$
9. For i=1 to n
   Set prod=1
   For i=0 to (i-1)
   Set $prod = prod \cdot (x - x_{k+1})$
   End for
   Set $sum = sum + (d_{kl} \cdot prod)$
10. Print sum
11. End

4.6.2. Counting Primitive Operation

1. Read n, x contributes two unit of operation.
2. The body of loop executes n times with count:

   Read x_i, y_i contributes two unit of operations

   =2n
3. Comparison of x contributes three unit of operations.
4. Set i=2 contributes one unit of operation.
5. While loop contributes three unit of operations
6. Set k=i-1 contributes two unit of operations
7. The body of loop executes n-1 times with count:
   i. The body of loop executes n-j times with count:
      (a) Compare and set d_{ij} contributes five operations

      $= (n-1)(n-j)\cdot 5$

      $= 5n^2 - 5nj - 5j - 5$
8. Set sum= y_k contributes one unit of operations.
9. The body of loop executes n times with count:
   i. set prod=1 contributes one unit of operations

      = n
10. The body of loop executes i times with count :

    Set prod contributes three unit of operations.

    = n+3i·n
11. Set sum contributes three unit of operations
12. Print sum contributes one unit of operations

To summarize, the number of primitive operations t (n) executed by algorithm is at least.

$$t(n) = 2 + 2n + 3 + 1 + 3 + 2 + 5n^2 - 5nj - 5j - 5 + 1 + n + 3in + 3 + 1$$
$$t(n) = 5n^2 + n(2 - 5j + 3i + 1) + 10$$

So the complexity of algorithm used for Newton-Greogary forward Interpolation from divided difference formula is method is O (n^2).

```
//Newton's divided Interpolation formula for unequal Intervals
#include <conio.h>
#include <stdio.h>
main()
{
int n,i,j,k;
float x, x1[20],y1[20],prod,sum,d[20][20];
clrscr();
printf("Enter n");
scanf("%d",&n);
printf("Enter x");
scanf("%f",&x);
for(i=1;i<=n;++i)
{
printf("Enter x and y\n");
scanf("%f %f",&x1[i],&y1[i]);
}
if((x<x1[1]) || (x>x1[n]))
{
printf("Out of range");
goto end;
}
i=2;
while(x<x1[i])
{
i=i+1;
}
k=i-1;
for(j=1;j<=(n-1);++j)
{
for(i=1;i<=(n-j);++i)
{
if(j==i)

d[i][j]=(y1[i+1]-y1[i])/(x1[i+1]-x1[i]);
else
d[i][j]=d[i+1][j+1]-d[i][j-1]/(x1[i+j]-x1[i]);
}
}
sum=y1[k];
for(i=1;i<=n;++i)
  {
```

```
 prod=1;
 for(j=0;j<=(i-1);++j)
 {
 prod=prod*(x-x1[k+j]);
 }
 sum=sum+d[k][i]*prod;
 }
 printf("Interpolation=%f",sum);
end:
getch();
}
```

Example f(0)=8, f(1)=68 and f(5)=123 find f(2)
Result=109.5

Conclusion:

There are two version of algorithm: sequential and parallel. In the experiments the execution times of both the sequential and parallel algorithms have been recorded to measure the performance (speedup) of parallel algorithm against sequential. The data presented in Table 4.1 to 4.4 represents the execution time taken by the sequential and parallel programs, We plot the graph (4.1-4.4) using the data in Table (4.1 to 4.4) to analyze the performance of parallel algorithm which is shown in figure 4.1 to 4.4. From the table 4.1 to 4.4 and figure 4.1 to 4.4 we observe that a vast difference in time required to execute the parallel algorithm and time taken by sequential algorithm. Based on our study we arrive at the following conclusions:

(1) Complexity of all interpolation formula calculated using Big Oh notation is O (n^2)

(2) We see that parallelizing serial algorithm using OpenMP has increased the performance.

(3) For multi-core system OpenMP provides a lot of performance increase and parallelization can be done with careful small changes.

(4) The parallel algorithm is approximately one and half faster than the sequential and the speedup is linear.