Finite difference as technique of numerical derivation

Midterm - Parallel Programming

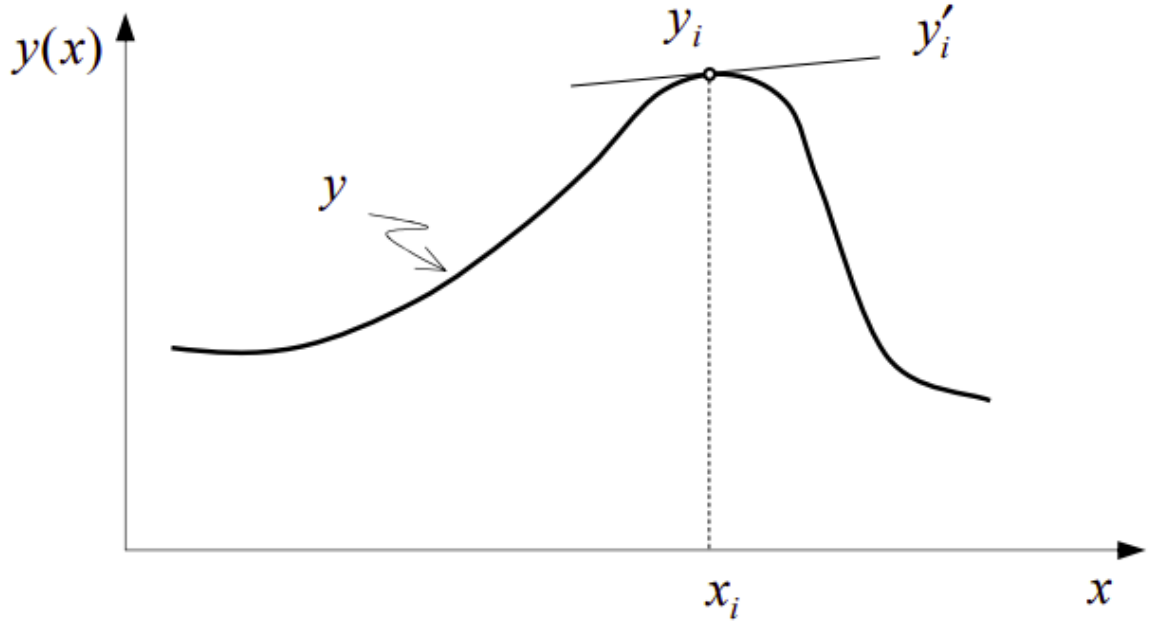
Katherine Camacho Calderón - Juan José Hoyos Urcué

This method is used to approximate derivatives of smooth functions, which must be analytically known or accurately evaluated by a given argument.

If we consider y = y (x), and we define the derivative of this function with respect to x like this:

$$y' = \frac{dy}{dx} = \lim_{\Delta x \to 0} \frac{\Delta y}{\Delta x} = \lim_{\Delta x \to 0} \frac{y(x + \Delta x) - y(x)}{\Delta x}$$

Graphically it can be seen as follows:



When Δx no longer tends to zero and tends to a finite value, the derivative at a point xi can be defined in several ways:

- if the point to the left is used it is called **regressive difference** , **backward** or **previous** and is defined by the following expression:

$$\nabla_h[f](x) = f(x) - f(x-h).$$

- If the point to the right is used, it is called **progressive difference** , **forward** or **posterior** and is defined by the following expression:

$$\Delta_h[f](x) = f(x+h) - f(x).$$

- If you use the point on the right and the one on the left it is called **center difference** and is defined by the following expression:

$$\delta_h[f](x) = f(x+\tfrac{1}{2}h) - f(x-\tfrac{1}{2}h).$$

This technique is widely used in numerical analysis, specifically in differential equations, ordinary numerical equations, difference equations, and partial derivative equation.

For this laboratory we decided to find the derivative of a function evaluated at a certain number of points, using the progressive, advanced or posterior finite difference.

The array x will contain all the points which will be evaluated in the function from which the derivation will be obtained, so the program aims to show n derivatives as a result, since n is the number of points in which the function will be evaluated.

This process was carried out sequentially first:

```
#include <bits/stdc++.h>
using namespace std;
typedef unsigned long long int ll;

double f(double x){
    return 3*(x*x)+2*x;
}
double derivar(double x,double h){
    return (f(x+h)-f(x))/h;

}
int main(){
    auto start = std::chrono::system_clock::now();
    ll n = 400000;
    ll x[n];
    for(ll h = 0;h<n;h++){
        x[h] = h;
    }

    for (int i=0;i<n;i++){
        double tmp = derivar(x[i],0.0001);
        cout<<tmp<<endl;
    }
```

```
    auto end = std::chrono::system_clock::now();
    chrono::duration<double> elapsed = end - start;
    printf("Ejecutar los  algoritmos de manera secuencial: %.2lf segundos\n",
elapsed.count());
}
```

In this part, the array is traversed linearly and one element of x is taken at a time to evaluate the function and then find the derivative. The time it took for the algorithm to find 400,000 derivatives for the previous function was: **2.00 seconds.**

Later the process was carried out in parallel:

```
#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#define n 400000
typedef unsigned long long int ll;
ll x[n];


double f(double y){
    return 3*(y*y)+2*y;
}
double derivar(double y,double h){
    return (f(y+h)-f(y))/h;
}

int main(int argc, char *argv[]){
    int rank,size;
    for(ll h = 0;h<n;h++){
        x[h] = h;
    }

    double t1,t2;
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    MPI_Barrier (MPI_COMM_WORLD);
    t1 = MPI_Wtime();


    int i;
    for (i=rank+1;i<=n;i+=size){
        printf("%f\n",derivar(x[i],0.0001));
    }

    MPI_Barrier (MPI_COMM_WORLD);
    t2 = MPI_Wtime() - t1;


    if (rank==0){
        printf("El tiempo que se demoraron los procesadores fue:  %f\n", t2);
    }
}
```

In this part, the values that are in array x were divided among all the available nodes of the processor, so each node can concurrently access a different value of x, to evaluate the function and make its proper derivation. The time it took for the algorithm to find 400000 derivatives for the previous function was:
1.405858 seconds.

The **speedup** was: **1.422618785**

Results:

Developing the parallel optimization of this algorithm is very important, since it helps people who normally work with derivatives on a long data set to do it in a much more efficient way, it is as important as the usefulness of the derivatives themselves, since the data with which scientist normally works is large, and a speedup of 1.4 would be very useful for any scientist or academic, since it saves time that can be used in a better way.