PARALLEL IMPLEMENTATION OF FAST MARCHING METHOD 18.337 FINAL REPORT

LEONARDO ANDRÉS ZEPEDA NÚÑEZ

1. Introduction. Seismic imaging has gained a great importance in the last years due to the high price of oil, a increasing demand from emergent economies, and the the constant explorations for new oil-fields. In fact, the traditional way of exploration based in geological patterns and well drilling had became too expensive. The oil-fields are located far beneath the ground or under some kilometers of water. Drilling a well is costly and needs a great amount of human and technical expertise. In order to overpass these difficulties and reduce cost, a seismic prospection is performed beforehand, in order to increase the probability of hitting an oil-field. This seismic prospection consists in sending seismic waves, using modified trucks or air cannons, and then recording the reflected waves using especial arrays of receivers. Once the data collection is completed, the imaging part starts, inverting the data to form the slowness model.

It exists an extensive list of different methods for seismic imaging; however, for the sake of simplicity we focus in the simplest model, which consist in the method of rays. In fact, if we suppose infinite frequency we wan reduce all the imaging processes which consist in solving the wave equation, to solving a transport equation. In fact, under the hypothesis of infinite frequency, a further reduction can be done, using an asymptotic expansion which result in solving the eikonal equation several times until the rays and the time arrivals given by the velocity model, match the data given by the on field receptors.

For the sake of solving the eikonal equation, some methods have been proposed. The most popular among them is the fast marching method introduced by J. Sethian and S. Osher [1] [2]. This method is first order and it has the property of visiting each node only few times, and combined with a heap sort strategy its complexity becomes $\mathcal{O}(n \log n)$. However, the amount of data used in real problem is gargantuan, then in order to perform the imaging process in a reasonable amount of time, parallelisation is indeed needed.

2. Modeling. The fundamental hypothesis in the seismic imaging is the fact that the soil and the rock are elastic mediums. Which is considered to be isotropic. These hypothesis plus some classic mechanics formalism give us that the equation which explains the propagation of a seismic wave is a wave equation given by (for further details [5] [6]).

$$\frac{\partial^2 u(x,t)}{\partial t^2} = \frac{1}{c^2(x)} \triangle u(x,t) \tag{2.1}$$

Where u is the displacement vector and c is the speed of the wave in the material given by the Lamé coefficients.

It exists a considerable amount of different methods in the literature to solve this equation. One of them takes advantage of the high frequency approximation. In fact we can propose an Anzat for this equation. Let u has the form:

$$u(x,t) = f(x) \cdot U(t - T(x))$$

where U is an impulse and T is the time which the waves takes to arrive to x. In other words, f control the amplitude of the wave, and U controls the phase. In this frame we suppose that U varies much faster than f.

Then putting this anzat inside 2.1 we have:

$$\begin{aligned}\frac{\partial^2 u(x,t)}{\partial t^2} = fU'' &= \frac{1}{c^2(x)} \triangle u(x,t) \\ &= \frac{1}{c^2} \triangle fU \\ &= \frac{1}{c^2}\left[U\triangle f - 2U''\nabla T \cdot \nabla f - fU'\triangle T + fU''\nabla T \cdot \nabla T\right]\end{aligned}$$

Given the fact that U varies much faster than f, it is possible to reduce this equation to

$$fU'' = \frac{1}{c^2} fU''\nabla T \cdot \nabla T$$

or (if f and U'' are different of zero)

$$\frac{1}{c^2}\|\nabla T\|^2 = 1$$

It is worth of remark that the travel time function only depends (in the high frequency approximation) of the velocity of the material.

3. Serial Implementation.

3.1. Algorithm. Seismic Imaging community defines the slowness as a quantity which is the inverse of the speed. We have then:

$$F = \frac{1}{c}$$

In order to get used to the Fast Marching Method a serial code was implemented.

The Eikonal Equation where F is positive Lipschitz function is given by:

$$\|\nabla T\| = \frac{1}{F} \tag{3.1}$$

plus boundary conditions. This will be s approximated by a finite differences scheme.

In order to define the scheme which will be used, let consider without any loss of generality the domain $\Omega = [0,1] \times [0,1]$ and a regular grid of space step Δx in the x direction and Δy in the y direction, such that every point in the grid can be written by $(x_i, y_j) = (i\Delta x, j\Delta y)$ for $i = 1, ..., N$ and $i = 1, ..., M$.

Let the finite difference operators be defined as follow

$$
\begin{aligned}
D^{x}_{i,j}T &= \frac{T(x_{i+1}, y_j) - T(x_i, y_j)}{\Delta x} \\
D^{-x}_{i,j}T &= \frac{T(x_i, y_j) - T(x_{i-1}, y_j)}{\Delta x} \\
D^{y}_{i,j}T &= \frac{T(x_i, y_{j+1}) - T(x_i, y_j)}{\Delta y} \\
D^{-y}_{i,j}T &= \frac{T(x_i, y_j) - T(x_i, y_{j-1})}{\Delta y}
\end{aligned}
$$

Then we can define the local upwind approximation proposed by Sethian and Osher using this compact notation as :

$$\left(\max(D^{-x}_{i,j}T, 0)^2 + \min(D^{x}_{i,j}T, 0)^2 + \max(D^{-y}_{i,j}T, 0)^2 + \min(D^{y}_{i,j}T, 0)^2\right)^{\frac{1}{2}} = \frac{1}{F(x_i, y_j)}$$

This scheme gives an approximation of the viscosity solution to the Eikonal Equation, and it can be solved using iterative methods like the Jacobi iteration or Rouy-Tourin iteration in $\mathcal{O}(N^4)$ operations (if $N = M$) however given the fact we are using a upwind method, it is possible to construct a data dependency between the elements of the grid.

In fact, the Fast Marching Method takes advantage of this special feature to decrease the order order of the operations to $\mathcal{O}(N^3 ln(N))$.

However, in this particular case a less dissipative scheme was used, given by

$$\left(\max(D^{-x}_{i,j}T, -D^{x}_{i,j}T, 0)^2 + \max(D^{-y}_{i,j}T, -D^{y}_{i,j}T, 0)^2\right)^{\frac{1}{2}} = \frac{1}{F(x_i, y_j)}$$

This scheme implies a cleaner and faster way to implement the local update.

The Fast Marching Method classifies the Nodes of the grid in three different categories or flags :

- **A** : accepted Nodes, thanks to the feature of the Upwind scheme, the solution does not change in these nodes, so once they are accepted their value remain constant.
- **NB** : Narrow Band Nodes, these nodes are between the accepted ones, which values are constant and the ones which haven't been visited yet, then their value can change.
- **FA** : Far Away Nodes, this nodes haven't been visited by the the method yet, so the value of the solution is unknown.

The main idea is of the Method is updating the Narrow Band Nodes, freezing one of them, and tagging it as Accepted. Then advance the Narrow Band by adding new nodes from Far Away to the Narrow Band, using the upwind finite difference operator to update their values.

The Fast Marching method can be summarized as follows :

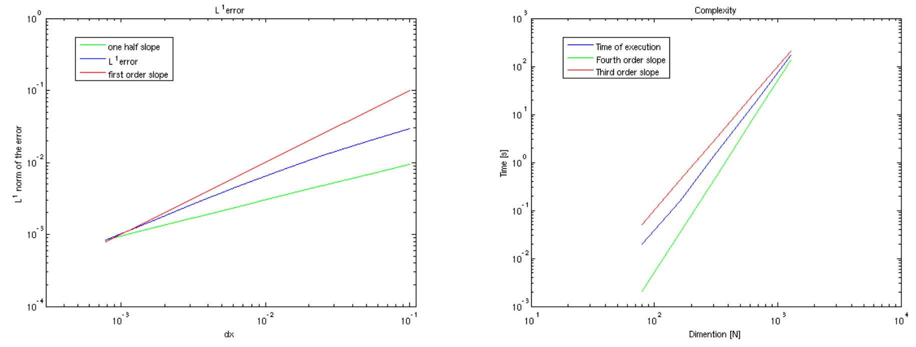

FIG. 3.1. *left The converge rate of T in the $L^1([0,1]^2)$ norm. right The time used to perform the computation v/s N the square root of the number of unknows.*

1. Initialize the whole domain as FA Nodes with a huge value on them
2. Set the boundary conditions, and set the Narrow Band
3. Find the node of the Narrow Band with the smallest value and tag it like Accepted, update its neighbors nodes, using the upwind scheme, and add them to the NB set.
4. repeat the last all nodes are Accepted.

The step of searching for the NB node with the smallest value, can be done in difference ways, in the original paper of Sethian and Osher it was done by a min heap data structure; however, given the fact that the aim here is made this code parallel, which means that the domain will be split in several smaller subdomains, then a heap data structure doesn't represent a significative advantage against a double linked chain. Then for the sake of simplicity the last was implemented with a bubble sort algorithm to find the minimum.

It is worth remarking that this method, is completely sequential in order to preserve the viscosity of the solution.

3.2. Results. The Algorithm was implemented following [3]. In fact a preliminary version was implemented in Matlab. The final version was coded in C++ and compiled with icpc Intel compiler. The benchmarks were run in an AMD 6600 machine running at 2.0 GigaHertz with 16 gigabytes of RAM.

The serial algorithm was tested agaisnt the Benchmark given by the known non smooth solution.

$$T(x,y) = \min\left(\sqrt{x^2+y^2}, \sqrt{(x-1)^2+(y-1)^2}\right) \tag{3.2}$$

We present the convergence rate in Fig 3.1 *left*, and we present time used to complete the computation in Fig. 3.1 *right.*

We can observe that the solution converges to the viscocity solution, at the good rate (the scheme is only first order and we have a discontinuity of the first derivative). This shows that the algorithm does what it is supposed to do.

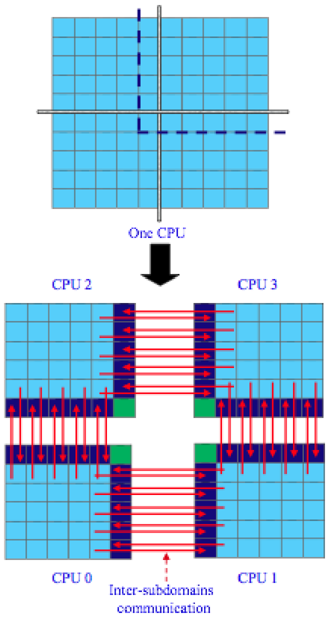

FIG. 4.1. *Interconnection between subdomains using the ghost cells.*

We have an asymptotic complexity which grows as $C(N) = N^{\alpha}$ where the empirical α from the data is given by least square fitting. The result is given by $\alpha = 3.3462$.

4. Parallel.

4.1. Architecture. A big domain is divided in several smaller subdomains. Is this case, the algorithm was performed to be used in squares. Each subdomain has a set of ghost cells which are linked to the boundary cell of its neighbors subdomains as shown in Fig. 4.1. The subdomain grid is defined by two number, n_x and n_y which are the dimentions of the subdomain matrix. Then the number of processors used is given by $n_p = n_x \cdot n_y$

An iteration of the parallel algorithm is given by a complete update of the local solution at each subdomain. After each iteration a transfer of information is performed. This information transfer is performed in 8 steps which are depicted in (Fig 4.2). This transfer is an synchronous transfer. Then at each boundary a fast sweep is performed using the new information from the ghost points.

This fast sweep is the key to maintain the viscosity of the solution computed.

A fast sweep consist in solving a 1 diminutional Eikonal Equation along the boundary. In order to do this, the same algorithm was used; however, it was only applied in the boundary cells (for further details see [3] pag. 91).

Once the Fast sweep was performed, all the nodes which value was decreased after the sweep are tagged as NB and a ad-hoc flag update is performed.

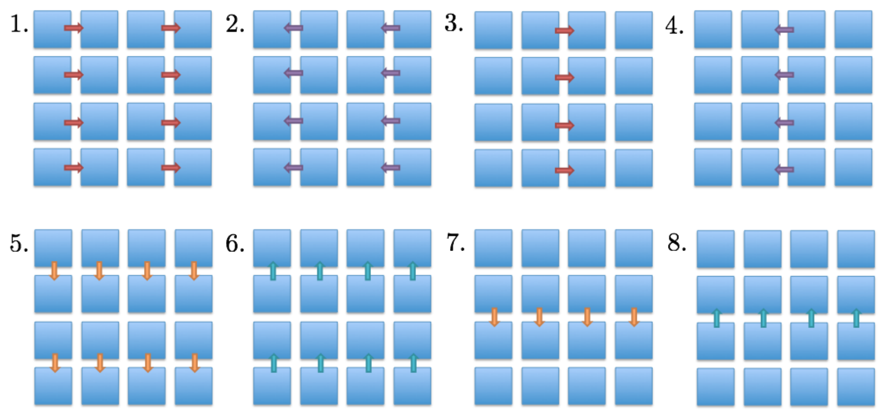

FIG. 4.2. *The steps to perform the transfer of information from one subdomain to its neighbors.*

The flag update consists in, compute the maximum (max) and the minimum (min) values which where modified. Then the tagging is performed as follows :

- Set to NB all the nodes with value bigger than min and less or equals than max
- Set to FA all the nodes with value bigger than max
- Set to A all the nodes with value strictly smaller than min

This new tagging and the update of the boundary cells gives us a new set of initial conditions to restart the Fast Marching method at each subdomain.

This iteration are performed until all the information from one subdomains arrived to all others. In this case this should be $n_x + n_y - 1$.

4.2. Scalability. From the empirical complexity we have that for a square array of size n, that the time spent to compute the solution of a $N \times N$ grid will be

$$T(N, n) = (2n-1)\left(\frac{N}{n}\right)^{\alpha}$$

where alpha given by $\alpha = 3,3462$, this implies that the speed up is given by the scalability factor

$$sc = \frac{n^{\alpha}}{2n-1} \approx \mathcal{O}\left(\frac{n^{\alpha-1}}{2}\right) \tag{4.1}$$

Given the fact that in this case $n_x = n_y = n$ we have that the number of processors is $n_p = n^2$.

The scalability factor, which is defined by the time of the serial implementation time divided by the parallel implementation time is given by

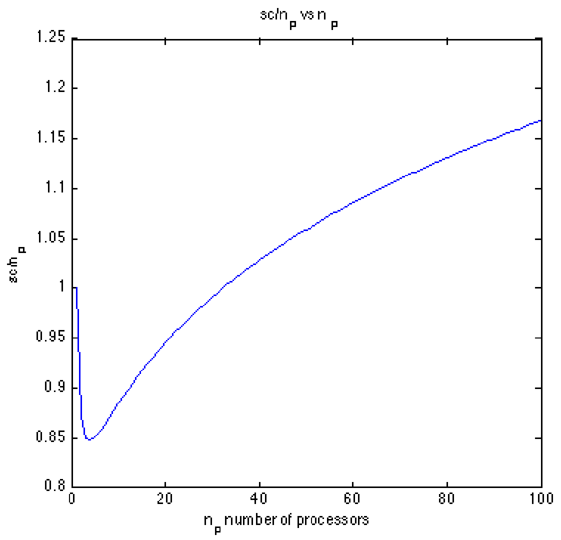

FIG. 4.3. *Plot of the scability divided by the number of the processors given by Eq 4.1 .*

$$sc = \frac{n_p^{\alpha/2}}{2\sqrt{n_p}-1} \approx \mathcal{O}\left(\frac{n_p^{\frac{\alpha-1}{2}}}{2}\right) = \mathcal{O}\left(\frac{n_p^{1.1731}}{2}\right)$$

Using the value of the empirical complexity we have that in this case the case is slightly super linear; however, there is a one half factor which reduces the efficiency to a half of the ideal one. Nevertheless the fact that the scalabilty is super linear implies that we can get have a speed up proportional to the number of processors, as n_p grows big enough. In fact for $n_p = 55$, $sc = 55^{1.1731}/2 = 55.02872 \approx n_p$.

However, this analysis was only based in the asymptotic behavior, the actual behavior is better as shown in the Fig. 4.3. We can observe that the actual ratio $sc(n_p)/n_p$ has a minimum value set to 0.85. and it becomes bigger than one with $n_p > 35$ which is smaller than what it was predicted by the asymptotic behavior.

4.3. Implementation and Results. The implementation of this algorithm was done for arrays of 2×1, 2×2, 4×4 and 6×6 subdomains. This was coded using C++ and the Intel OpenMPI libraries.

The algorithm was run the the wave2 cluster of the Wave and Imaging group at the MIT Mathematics Department, which consist in AMD 6600 machine with 48 cores running at 2.0 Gigahertz and 256 gigabytes of shared RAM. An Example of the solution given by this algorithm is given in Fig. 4.4.

The results of the time used to complete the computation are given in Fig. 4.5. It is possible to appreciate than for n small enough the serial algorithm is faster than the parallel ones,. This is given by the latency of the communications. However, once the number of unknowns starts to grow, there is a significant speed up given the

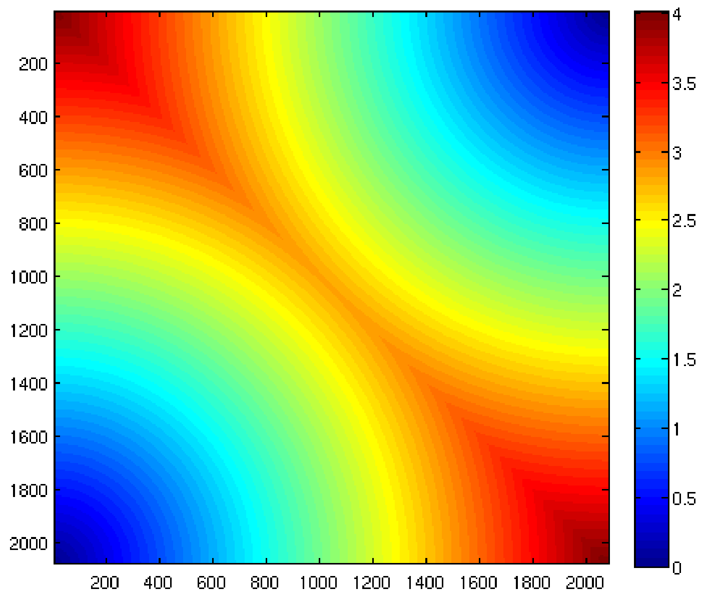

Fig. 4.4. *Solution of the Eikonal Equation benchmarking for Eq 3.2 in an* 4×4 *array with* 2100^2 *unknowns* .

Theoretical scalability.

The results are satisfactory and they fit well with the theoretical computations of the algorithm, which are presented in Table 4.1

Table 4.1
Speed up of the Parallel Implementation

n_p	Theoretical speed up Eq. 4.1	Empirical speed up
16	14.774	17.910
36	36.51379	35.6662

5. Conclusion.

This project was highly instructive, coding everything from ashes was a real challenge.

The algorithm was coded in a serial frame to get used to C++, and to have a reference in order to compare the reduction in the computational time. This serial algorithm gives the correct viscosity solution and is an fundamental part in the parallel algorithm.

The parallel algorithm described in this report was implemented in C++ using OpenMPI libraries. It does converge as expected to the viscosity solution and it presents the speed up which was predicted by straight forward theoretical computation.

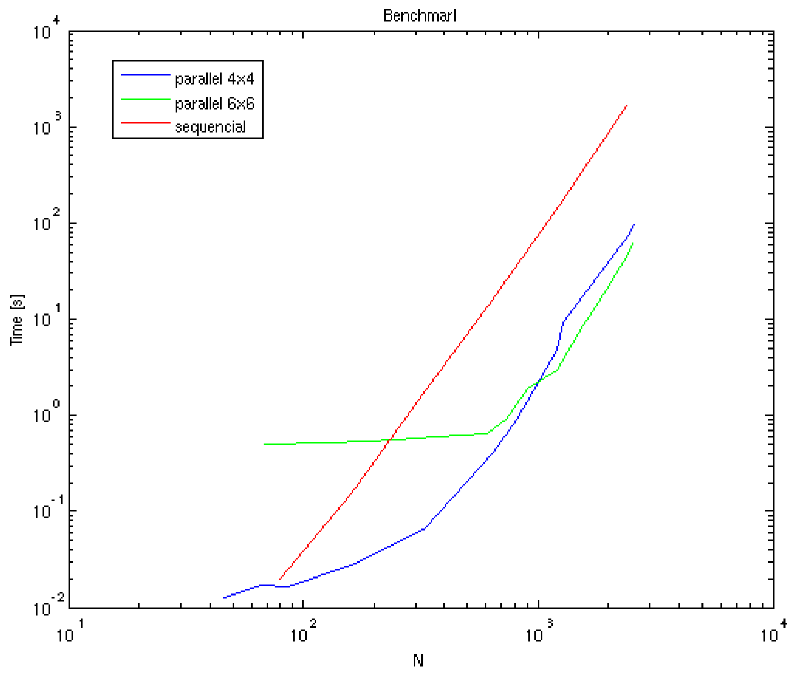

FIG. 4.5. *The steps to perform the transfer of information from one subdomain to its neighbors.*

The Eikonal Equation can be solved in parallel reducing the time of computation, and it presents a super linear behavior (in the best array configuration) which implies that increasing the number of processors the ratio $sc(n_p)/n_p$ should increase, giving better performance.

This case, was the good example of how to transform a pure serial algorithm, in to a parallel one. Even though the parallel one will recompute the same unknown several times, the fact of working with smaller subdomains and the fact that the complexity is super linear, allowing us for trading off, which can be used to obtain an speed up bigger than the number of processors.

References.

1. James A. Sethian. A fast marching level set method for monotonically advancing fronts. Proc. Nat. Acad. Sci. U.S.A., 93(4):15911595, 1996.
2. James A. Sethian. Level set methods and fast marching methods, volume 3 of Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, Cambridge, second edition, 1999
3. Maria Cristina Tugurlan, FAST MARCHING METHODS - PARALLEL IMPLEMENTATION AND ANALYSIS, PhD thesis 2008 Louisiana State University and Agricultural and Mechanical College
4. J. Andreas Brentzen ON THE IMPLEMENTATION OF FAST MARCHING METHODS FOR 3D LATTICES, DTU TECHNICAL REPORT IMM-REP-2001-13

5. Keiiti Aki, Paul G.Richards. Quantitative Seismology. University Sciences Book. Sausalito California.
6. Seth Stein, Michael Wysession. An Introduction to Seismology, Earthquakes and Earth Structure. 2003 Blackwell Publishing

6. Appendix.

1. list.h This file is the implementation of the double linked list. With ad-hoc append and removing functions

```
class Node
 {
  public:
    int indi, indj; //positions
    int** flag;   // the flag for this node
    Node* child;
    Node* father;
    double** T;  // a pointer to the values
    double** F;  // a pointer to the matrix of F_{i,j}
    Node(){
     indi = 0;
      indj = 0;
      child = NULL;
      father = NULL;
      T = NULL;
      F = NULL;
    }

    Node(int i, int j, Node* next, Node* before, double**
 t, double** f, int** drapeau  ){
      indi = i;
      indj = j;
      father = next;
      child = before;
      T = t;
      F = f;
      flag = drapeau;
    }
};

// method for insertion of nodes
// we suppose that the list is ordered, then we should see where the new node has to be.

void insert(Node* &biggest, Node* &smallest, Node* add)
{
  if(biggest != NULL) {  //the tree has at least one element
    //  cout << "biggest is not null" << endl;
    int i = (biggest)->indi;
    int j = (biggest)->indj;
    if((*biggest).T[i][j] < add->T[add->indi][add->indj]){
      add->child = biggest;    //     cout << "add to last bigger" << endl;
      add->father = biggest->father;     // cout << " connecting with the other part of the chai
      biggest->father = add; // cout << "the father of the child of the biggest" << endl;
      biggest = add;        // cout << "updating the biggest" << endl;
    }
  else {
    insert(biggest->child, smallest, add);
  }
}
  else{  // tree is empty
    //  cout << "the tree is empty" << endl;
    biggest = add; // cout << "including the head" << endl;
if(smallest == NULL){
  add->father = NULL;
}
else{
  add->father = smallest;
}
//    cout << "including the father" << endl;
add->child  = NULL; // cout << "including the child" << endl;
smallest  = add; // cout << "including the tail" << endl;
  }
}

void fast_reorder(Node* &biggest, Node* &smallest){ // bubble sort
    if(biggest!= NULL){
        Node* aux;
        aux = biggest->child;
        if (aux!= NULL){
            int i,j, ia, ja;
            i = biggest->indi;
            j = biggest->indj;
            ia = aux->indi;
            ja = aux->indj;
            if (biggest->T[i][j] < aux->T[ia][ja]){ //wrong order
                if(aux->child != NULL){ // aux is not the smallest
                    (aux->child)->father = biggest;
                }else{ // aux is the smallest
                    smallest = biggest;
                }
                if(biggest->father != NULL){ // bisggest is no the biggest
                     (biggest->father)->child = aux;
                }
                aux->father = biggest->father;
                biggest->child = aux->child;
                biggest->father= aux;
                aux->child = biggest;

            }
            fast_reorder(aux, smallest);
        }
    }
}

// removing the node from the list
```

```
void remove(Node* &biggest, Node* &smallest, Node* add){
  if(biggest == add && smallest == add){ //cout << "removing head and tail";
    biggest = NULL;
    smallest = NULL;
  }
  else{
    if (biggest == add){  // cout << "we have to remove the head" << endl;
      add->child->father = NULL;
      biggest = add->child;
    }
    else{
      if (smallest == add){ // cout<< "we have to remove the tail" << endl;
        smallest->father->child = NULL;
        smallest = add->father;
      }
      else{
        add->father->child = add->child;
        add->child->father = add->father;
      }
    }
  }
}

void print_values(Node* biggest)
{
Node* aux = biggest;
int i = 0;
while (aux!=NULL){
  if (i != 0){
    cout << "-->" ;
  }
  cout << aux->T[aux->indi][aux->indj] ;
  aux = aux->child;
  i++;
}
cout << "\nthe number of nodes is " << i << endl;
}
```

2. mesh.h. This File defines the slowness model and give some printing functions. The outputs will be read in Matlab for plotting and for convergence estimates.

```
double** speedgeneration (double a,double b,double c,double d,int n,int m){
  double** F;
  F = new double*[n];
  for(int j = 0; j<n ; j++ )
    F[j] = new double[m];

  for(int i = 0; i<n; i++){
    for(int j = 0; j < m; j++){
      F[i][j] = speed(a + (b-a)/(n-1)*i, c + (d-c)/(m-1)*j);
    }
  }
  return F;
}

void print_file(int n, int m, double** T){
  ofstream myfile;
  myfile.open("solution.txt");
   for (int j = m-1; j>=0 ; j--){
    for(int i = 0 ; i<n ; i++){
      myfile << "\t" << T[i][j];
    }
    myfile <<endl;
  }
  myfile.close();
}
```

3. quadratic.h. This class implements all the functions to update the nodes and the Fast sweep strategy

```
  double quadratic(double a, double b, double c){
    // a and b are the values of the neighbors and c is just \delta x / F square
    double delta;
    double d;
    delta = (2*c - (a-b)*(a-b) );
    if (delta >= 0)
      d = (a + b + sqrt(delta))/2;
    else
      d = 1000000000; //just a huge number
    return d;
  }

  void solve(Node* kk,double dx){
    Node k = *kk;
    int i = k.indi;
    int j = k.indj;
    if(k.flag[i][j] != 0){
      double Tij = k.T[i][j];
      double Tiij= k.T[i+1][j];
      double Tijj= k.T[i][j+1];
      double Tj  = k.T[i-1][j];
      double Ti  = k.T[i][j-1];
      double F   = k.F[i][j];
      double s   = dx*dx/(F*F);
      double d1, d2, d3, d4 ,d5;

      d1  = 100000000;
      d2 = d1;
      d3  = d2;
      d4  = d3;
      d5  = d4;
      d1 = dx/F + min(min(Ti,Tijj), min(Tiij,Tj));
      if( Ti < Tij && Tj < Tij){
       d2 = quadratic(Ti,Tj,s);
      }
      if( Tj < Tij && Tijj < Tij){
       d3 = quadratic(Tj,Tijj,s);
      }
      if( Tijj < Tij && Tiij < Tij){
       d4 = quadratic(Tijj,Tiij,s);
      }
      if( Tiij < Tij && Ti < Tij){
       d5 = quadratic(Tiij,Ti,s);
      }
      k.T[i][j] = min( min(d1,min(d2,d3)) ,min(d4,d5));
    }
  }

  void  Update(Node* &biggest,Node* &smallest,Node** &Grid,int n,int m,int** &Fl, double dx) {
      int curri = smallest->indi;
      int currj = smallest->indj;

      // updating the values of the neighbors
      if (curri < n-2 )
          solve(&Grid[curri+1][currj], dx);
      if (curri > 1 )
          solve(&Grid[curri-1][currj], dx);
      if (currj > 1)
          solve(&Grid[curri][currj-1], dx);
      if(currj < m-2 )
          solve(&Grid[curri][currj+1], dx);

      //setting the flag of the smallest one to zero
      Fl[smallest->indi][smallest->indj] = 0;
      remove(biggest, smallest, &Grid[curri][currj]);

  for(int i = curri - 1; i <= curri+1; i++){
        for(int j = currj-1; j <= currj+1; j++){
            if(i !=curri || j != currj){
  if (Fl[i][j] == 1) {
      remove(biggest,smallest, &Grid[i][j]);
    }
        }
      }
  }

      for(int i = curri - 1; i <= curri+1; i++){
        for(int j = currj-1; j <= currj+1; j++){
            if(i != curri || j != currj){
                  if(i>0 && i<n-1 && j>0 && j<m-1){
                      if(Fl[i+1][j]!=0 && Fl[i-1][j]!=0 &&  Fl[i][j+1]!=0 && Fl[i][j-1]!=0 ){
                          Fl[i][j] = 2;
                      }
                      else{
                          if (Fl[i][j] == 1) {
                              insert(biggest, smallest, &Grid[i][j]);
                          }
                          if (Fl[i][j] != 0 && Fl[i][j] != 1){
                              Fl[i][j] = 1;
                              insert(biggest, smallest, &Grid[i][j]);
                          }

                      }
                  }
              }
          }
      }


  }

  void Fastsweep(double** data, double* speed, int n, double* &sol,double dx){
    double first[n];
    double second[n];
    //copy the initial data
    for (int i = 0; i < n ; i++){
      sol[i] = data[1][i];
    }
    // the first sweep
    int i = 1;
    double Tij,Tiij, Tijj,Tj,Ti,F,s;
    double d1, d2, d3, d4, d5;

    for(int j = 1; j < n-1; j++){
      Tij = data[i][j];
      Tiij= data[i+1][j];
      Tijj= data[i][j+1];
      Tj  = data[i-1][j];
      Ti  = data[i][j-1];
      F   = speed[i];
      s   = dx*dx/(F*F);
      d1  = 100000000;
      d2 = d1;
      d3  = d2;
      d4  = d3;
      d5  = d4;

      d1  = dx/F + min(min(Ti,Tijj), min(Tiij,Tj));
      if( Ti < Tij && Tj < Tij){
       d2 = quadratic(Ti,Tj,s);
      }
      if( Tj < Tij && Tijj < Tij){
       d3 = quadratic(Tj,Tijj,s);
      }
      if( Tijj < Tij && Tiij < Tij){
       d4 = quadratic(Tijj,Tiij,s);
      }
      if( Tiij < Tij && Ti < Tij){
       d5 = quadratic(Tiij,Ti,s);
      }

      data[i][j] = min(min(min(d1,min(d2,d3)),min(d4,d5)), sol[j]);

      first[j] = data[i][j];
    }
   for(int j = n-2; j >=0; j--){
      Tij = data[i][j];
      Tiij= data[i+1][j];
      Tijj= data[i][j+1];
      Tj  = data[i-1][j];
      Ti  = data[i][j-1];
      F   = speed[i];
      s   = dx*dx/(F*F);

      d1  = 100000000;
      d2 = d1;
      d3  = d2;
      d4  = d3;
      d5  = d4;

      d1  = dx/F + min(min(Ti,Tijj), min(Tiij,Tj));
      if( Ti < Tij && Tj < Tij){
       d2 = quadratic(Ti,Tj,s);
      }
      if( Tj < Tij && Tijj < Tij){
       d3 = quadratic(Tj,Tijj,s);
      }
      if( Tijj < Tij && Tiij < Tij){
       d4 = quadratic(Tijj,Tiij,s);
      }
      if( Tiij < Tij && Ti < Tij){
       d5 = quadratic(Tiij,Ti,s);
      }

      data[i][j] = min(min(min(d1,min(d2,d3)),min(d4,d5)), sol[j]);
      second[j] = data[i][j];
    }
   for(int ii = 1; ii<n-1 ; ii++){
     sol[ii] = min(min(sol[ii],first[ii]), second[ii]);
   }
}

// function to update flags after a double fast sweep in the boundary
void  Update_flags(double** &T,int n,int m,int** &Fl){
  double minT = 100000000;
  double maxT = -100000000;
  for (int i = 1; i < n-1 ; i++ ){
    if(Fl[i][1] == 1){
      minT = min(minT,T[i][1]);
      maxT = max(maxT,T[i][1]);
    }
    if(Fl[i][m-2] == 1){
      minT = min(minT,T[i][m-2]);
      maxT = max(maxT,T[i][m-2]);
    }
  }
  for (int j = 1; j < m-1; j++ ){
    if(Fl[1][j] == 1){
      minT = min(minT,T[1][j]);
      maxT = max(maxT,T[1][j]);
    }
    if(Fl[n-2][j] == 1){
      minT = min(minT,T[n-2][j]);
      maxT = max(maxT,T[n-2][j]);
    }
  }
  if(minT != 100000000 && maxT != -100000000){
    for(int i = 1; i<n-1 ; ++i){
      for(int j = 1; j < m-1; j++){
if(Fl[i][j] !=1){
  if (T[i][j] < minT){
    Fl[i][j] = 0 ;
  }
  else{
    if(T[i][j] >= maxT){
      Fl[i][j] = 2;
    }
    else{
      Fl[i][j] = 1;
    }
  }
       }
      }
    }
  }
```

```
}
```

4. main.cpp. This is the example of the parallel implementation, in this case for a 4×4 grid.

```
// program to compute the distance from two oposites points
// in parallel with sixteen cores, in a sixteen 18x18 grids optimized version

#include <iostream>
#include <sstream>
#include <fstream>
#include <cstdlib>
#include <iomanip>
#include <ctime>

using namespace std;

#include "mpi.h"
#include "math.h"
#include "fonctions.h"
#include "list.h"
#include "mesh.h"
#include "quadratic.h"

int main ( int argc, char *argv[] );

int main ( int argc, char *argv[] ) {

  //-------------------------------MPI -----------------------------
  // initialize  the MPI variables

  int id;
  int np;
  double wtime;
  int tag;
  //typical stuff for MPI
  MPI_Status Stat;
  // data type for sending vectors
  MPI_Datatype columntype;
  MPI_Datatype rowtype;

//  Initialize MPI.
  MPI::Init ( argc, argv );
//  Get the number of processes.
  np = MPI::COMM_WORLD.Get_size ( );
//  Get the individual process ID.
  id = MPI::COMM_WORLD.Get_rank ( );

//  Process 0 prints an introductory message.

  double t0;
```

```
if ( id == 0 )
{
  cout << "Running Eikonal Equation Solver based in Fast Marching Method"<< endl;
  t0 = MPI_Wtime();
}

// initialization of the local variables
int n = 18;   //size of the grid
int m = 18;
double a = 0;  // box [a,b]x[c,d]
double b = 1;
double c = 0;
double d = 1;
double dx = (b-a)/(n-3);
// number of subdomains
int ax = 4; // in the x coordinate
int ay = 4; // in the y coordinate

//cout << "defining the input output strutures" << endl;
MPI_Type_contiguous(m, MPI_DOUBLE, &columntype);
MPI_Type_commit(&columntype);

MPI_Type_contiguous(n, MPI_DOUBLE, &rowtype);
MPI_Type_commit(&rowtype);

double** T;   // the solution
T = new double*[n];

int** Fl;     // Flags
Fl = new int*[n];

Node** Grid; //pointer from the grid to the list
Grid = new Node*[n];

//initialization of the subarrays
for(int i = 0; i<n; i++){
  T[i] = new double[m];
  Fl[i] = new int[m];
  Grid[i] = new Node[m];
}

double** G;  //defining the speed map
G = speedgeneration(a,b,c,d,n,m);  //building the speed map, have
// initialiate the node reference table and the distance map
for(int i = 0; i<n ; i++){
  for (int j = 0; j<m ; j++){
    Grid[i][j] = Node(i,j,NULL,NULL,T,G,Fl);
    T[i][j]    = 100000000;
  }
  }

  Node* biggest = NULL;
  Node* smallest = NULL;

  // cout << "we need to set all the boundarie terms as far awai i.e. set Fl = 2 " << endl;
  for(int i = 0; i<n; i++){
    for(int j = 0; j<m; j++){
      Fl[i][j] = 2;
    }
  }

// setting the initial conditions in the central core must be done for each core
//and starting point in the first quadrant
 if(id  == 0){
 T[1][1] = 0;
 Fl[1][1] = 1;
 }
 //another starting point in the last quadrant
 if(id  == 15){
 T[n-2][m-2] = 0;
 Fl[n-2][m-2] = 1;
 }

 // cout << "adding all the nodes with narrow flag to biggest" << endl;
  for(int i = 0 ; i<n; i++){
    for(int j = 0; j<m; j++){
      if(Fl[i][j]==1){
// cout << i << "," << j << endl;
insert(biggest, smallest, &Grid[i][j]);
      }
    }
  }

//-----------solving the local problem------------------------

 //cout << "performing the iteration" << endl;

  while(biggest!=NULL){
    Update( biggest, smallest, Grid, n,m, Fl,dx)    ;
  }

  //++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

   // scheme of the array
   //  ----------------------
   //  | id + ax  | id +ax+1 |...
   //  --------------------
   //  | id       | id+1     |...
   //------------------------

  for (int kk = 0; kk<4; kk++){

  double North[n];
  double South[n];
  double East[m];
  double West[m];

  for (int i = 0 ; i<n; i++){
    North[i] = 100000000;
    South[i] = 100000000;
  }
  for (int j = 0 ; j<m; j++){
    East[j] = 100000000;
    West[j] = 100000000;
  }
  //cout << "sending information" << endl;

   if(id % 2 == 0 && ax != 1 ) {
    // we are going to transfer information to the right
    // cout << "process " << id << " is sending information to " << id+1<< endl;

      double send_buff[m];
      for(int j = 0 ; j<m ; j++){
  send_buff[j]= T[n-2][j];
  if((id )%ax == 0){
  West[j] = 100000000; // we wont modify the West values
  }
}
      MPI_Send (&send_buff, 1 ,columntype, id+1 , 2, MPI_COMM_WORLD );
      //cout<< "information sent" << endl;

     //cout << "receiving ghost cells from the right then East" << endl;
     MPI_Recv (East,m, MPI_DOUBLE, id+1 ,0 , MPI_COMM_WORLD  ,&Stat);
   }

   if((id-1) % 2 == 0){
     //cout << "receiving ghost cells" << endl;
     MPI_Recv (West,m, MPI_DOUBLE, id-1 ,2 , MPI_COMM_WORLD  ,&Stat);

     // cout << "sending the ghost sell back to the left process" << endl;
     double send_buff[m];
      for(int j = 0 ; j<m ; j++){
  send_buff[j]= T[1][j];
  if((id +1)%ax == 0){ //i.e. we are at the right boundary of the subdomains
    East[j] = 100000000; // to change afterwards
  }
}
```

```
      MPI_Send (&send_buff, 1 ,columntype, id-1 , 0, MPI_COMM_WORLD );
      //cout<< "information sent" << endl;
   }
   //=========================================================================

   if((id-1) % 2 == 0 && (id+1) % ax != 0 ) {
    // we are going to transfer information to the right
     //cout << "process " << id << " is sending information to " << id+1<< endl;

      double send_buff[m];
      for(int j = 0 ; j<m ; j++){
  send_buff[j]= T[n-2][j];
}
      MPI_Send (&send_buff, 1 ,columntype, id+1 , 2, MPI_COMM_WORLD );
      //cout<< "information sent" << endl;

     //cout << "receiving ghost cells from the right then East" << endl;
     MPI_Recv (East,m, MPI_DOUBLE, id+1 ,0 , MPI_COMM_WORLD  ,&Stat);
   }

   if(  id % 2 == 0 && id%ax != 0  ){
     //cout << "receiving ghost cells" << endl;
     MPI_Recv (West,m, MPI_DOUBLE, id-1 ,2 , MPI_COMM_WORLD  ,&Stat);

     // cout << "sending the ghost sell back to the left process" << endl;
     double send_buff[m];
      for(int j = 0 ; j<m ; j++){
  send_buff[j]= T[1][j];
  if((id +1)%ax == 0){ //i.e. we are at the right boundary of the subdomains
  }
}
      MPI_Send (&send_buff, 1 ,columntype, id-1 , 0, MPI_COMM_WORLD );
      //cout<< "information sent" << endl;
   }

   //*********************** information transfer in the vertical direction

   if(id < ax || (id >= 2*ax && id < 3*ax ) ) {
    // we are going to transfer information to the upside
     //  cout << "process " << id << " is sending information to " << id+ay<< endl;
      double send_buff[n];
      for(int i = 0 ;i<n ; i++){
  send_buff[i]= T[i][m-2];
  if(id < ax){
    South[i] = 100000000;
   }
}
      MPI_Send (&send_buff, 1 ,rowtype, id+ax , 3, MPI_COMM_WORLD );
      //cout<< "information sent" << endl;
      //cout << "receiving ghost cells from the the up the North" << endl;
      MPI_Recv (North,n, MPI_DOUBLE, id+ax ,4, MPI_COMM_WORLD  ,&Stat);
   }

    if( (id >= ax && id < 2*ax) || (id >= 3*ax)  ) {
  // cout << "receving the ghost points" <<<

      //cout << "receiving ghost cells" << endl;
      MPI_Recv (South,n, MPI_DOUBLE, id-ax ,3 , MPI_COMM_WORLD  ,&Stat);

      // cout << "sending the ghost sell back to the left process" << endl;
      double send_buff[n];
       for(int i = 0 ; i<n ; i++){
   send_buff[i]= T[i][1];
   if( id+ax  >=  np){ //i.e. we are at the upper boundary of the subdomains
     North[i] = 100000000;
   }
 }
       MPI_Send (&send_buff, 1 ,rowtype, id-ax , 4, MPI_COMM_WORLD );
       //cout<< "information sent" << endl;
   }

    // ==================================================================

    if(id >= ax && id < 2*ax ) {
     // we are going to transfer information to the upside
      //cout << "process " << id << " is sending information to " << id+ay<< endl;
       double send_buff[n];
       for(int i = 0 ;i<n ; i++){
   send_buff[i]= T[i][m-2];
    if(id < ax){
     South[i] = 100000000;
    }
 }
       MPI_Send (&send_buff, 1 ,rowtype, id+ax , 3, MPI_COMM_WORLD );
       //cout<< "information sent" << endl;

      //cout << "receiving ghost cells from the he up the North" << endl;
      MPI_Recv (North,n, MPI_DOUBLE, id+ax ,4, MPI_COMM_WORLD  ,&Stat);
   }

    if( id >= 2*ax && id < 3*ax ) {
      //cout << "receiving ghost cells" << endl;
      MPI_Recv (South,n, MPI_DOUBLE, id-ax ,3 , MPI_COMM_WORLD  ,&Stat);

      // cout << "sending the ghost sell back to the left process" << endl;
      double send_buff[n];
       for(int i = 0 ; i<n ; i++){
```

```
    send_buff[i]= T[i][1];
    if( id+ax  >=  np){ //i.e. we are at the upper boundary of the subdomains
      North[i] = 100000000;
    }
  }
        MPI_Send (&send_buff, 1 ,rowtype, id-ax , 4, MPI_COMM_WORLD );
        //cout<< "information sent" << endl;
    }

    //+++++++++++++++++++++++information sent, now update+++++++++++++++++++++++++++++++
       //----------------------------------------------Horizontal Direction----------------------

       double** dataH; dataH = new double*[3];
       dataH[0] = new double[m]; dataH[1] = new double[m]; dataH[2] = new double[m];
       double* answerH; answerH = new double[m];
       double columnspeed[m];

       for(int j = 0; j<m; j++){
         dataH[0][j]  = East[j];
         dataH[1][j]  = T[n-2][j];
         dataH[2][j]  = T[n-3][j];
         columnspeed[j]  = G[n-2][j];
       }

       // cout << "performing fast sweep East in core " << id << endl;
       Fastsweep(dataH, columnspeed, m, answerH, dx);
       for(int j = 0; j < m ; j++){
         if (  T[n-2][j] > answerH[j] ){
   Fl[n-2][j] = 1;
         }
         T[n-2][j] = answerH[j];
         T[n-1][j] = East[j];
       }

       for(int j = 0; j<m; j++){
         dataH[0][j] = West[j];
         dataH[1][j] = T[1][j];
         dataH[2][j] = T[2][j];
         columnspeed[j] = G[1][j];
       }

       // cout << "performing fast sweep West in core " << id << endl;
       Fastsweep(dataH, columnspeed, m, answerH, dx);
       for(int j = 0; j<m ; j++){
         if (  T[1][j] > answerH[j] ){
    Fl[1][j] = 1;
         }
         T[1][j] = answerH[j];
         T[0][j] = West[j];
       }

       //---------------------------------------------------Vertical Direction------------------

       double** dataV; dataV = new double*[3];
       dataV[0] = new double[n]; dataV[1] = new double[n]; dataV[2] = new double[n];
       double* answerV; answerV = new double[n];
       double rowspeed[n];

       for(int i = 0; i<n; i++){
         dataV[2][i]  = North[i];
         dataV[1][i]  = T[i][m-2];
         dataV[0][i]  = T[i][m-3];
         rowspeed[i]  = G[i][m-2];
       }

       //cout << "performing fast sweep North in core " << id << endl;
       Fastsweep(dataV, rowspeed, n, answerV, dx);
       for(int i = 0; i<n ; i++){
         if (  T[i][m-2] > answerV[i] ){
   Fl[i][m-2] = 1;
         }
         T[i][m-2] = answerV[i];
         T[i][m-1] = North[i];
       }

       for(int i = 0; i<n; i++){
         dataV[0][i] = South[i];
         dataV[1][i] = T[i][1];
         dataV[2][i] = T[i][2];
         rowspeed[i] = G[i][1];
       }

       //cout << "performing fast sweep South in core " << id << endl;
       Fastsweep(dataV, rowspeed, n, answerV, dx);
       for(int i = 0; i<n ; i++){
         if (  T[i][1] > answerV[i] ){
   Fl[i][1] = 1;
         }
         T[i][1] = answerV[i];
         T[i][0] = South[i];
       }
  // ______________Boundary updated, updating flags____________

       Update_flags(T, n,m,Fl);

       for(int i = 0 ; i<n; i++){
          for(int j = 0; j<m; j++){
    if(Fl[i][j]==1){
     insert(biggest, smallest, &Grid[i][j]);
    }
          }
       }

  //++++++++++++++++++ Solving Locally +++++++++++++++++
      // cout << "performing the fast sweeping locally in " << id << endl;
      while(biggest!=NULL){
        Update( biggest, smallest, Grid, n,m, Fl,dx);
      }
    }

  //-----------------giving back all the information to the first node------

     if (id > 0  )
      {
        double send_buff[m];
        tag = 1;
        for(int i = 0; i<n ; i++){
  // cout << "sending the " << i<< "th column of the array" << endl;
  for(int j =0; j<m; j++){
    send_buff[j]= T[i][j];
  }
  MPI_Send (&send_buff, 1,columntype, 0 , 1, MPI_COMM_WORLD );
          }
      }

      if(id == 0){
        double t1;
        t1 = MPI_Wtime();
        tag = 1;
        double*** Global;
        Global = new double**[np];
        cout <<"number of processors are " << np << endl;
        Global[0] = T;
        for(int i = 1; i < np ; i++ ){
  Global[i] = new double*[n];
  for (int j = 0; j<n; j++ ) {
    Global[i][j] = new double[m];
    double b[m];
    MPI_Recv (b,m, MPI_DOUBLE, i ,1 , MPI_COMM_WORLD  ,&Stat) ;
    for(int kk = 0; kk < m; kk++){
      Global[i][j][kk] = b[kk];
    }
  }
        }

        cout << "total time of computation  " << t1-t0 << "seconds" << endl;
        double** TT = blend(np, n, m, 4, 4, Global);
        print_file(4*n-6,4*m-6,TT);
      }

    MPI_Type_free(&columntype);
    MPI_Type_free(&rowtype);
    MPI::Finalize ( );

    return 0;

  }
```