Eikonal equation: implementation and analysis of multiple numerical solvers

Giorgio Negro, Enrico Tirri, Dario Santoro

June 2024

Contents

Chapter 1

Introduction

1.1 Introduction and motivation

Application of the Eikonal equation are numerous. It's used, for example, in geometric optics, or in cardiac electrophysiology to describe the propagation of some action, like electrical impulse, to the heart. Given the importance of an efficient way to calculate the solution, many numerical methods have been proposed, each trying to speed up the process to get the solution and dealing with always larger meshes, either triangulated or tetrahedral. Each method is based on a local solver, able to compute, usually using a jacobi method, the solution on a single element, and an algorithm describing the way to traversing the mesh. In this project we briefly explore the theory of the equation, we describe the data structures used, we describe the local solvers and the global solvers, and we analyze the performance of each method on different testing meshes.

1.2 Theory

The Eikonal equation is a special case of the Hamilton-Jacobi partial differential equations, encountered in problems of wave propagation

$$\begin{cases} |\nabla u(x)| = f(x), & x \in \Omega \\ u(x) = 0, & x \in \partial\Omega \end{cases} \tag{1.1}$$

where Ω is an open set in R^n and F(x) is a function with positive values. The module is the Euclidean norm taking into account the anisotropic symmetric velocity information M as a property of the domain omega, i.e.

$$|v(x)|^2 = ||v(x)||_M^2 = \langle v(x), v(x)\rangle_M = v(x)^T M(X) v(x) \tag{1.2}$$

Physically, the solution $u(x)$ is the shortest time needed to travel from the boundary to x inside Ω, with $F(x)$ being the time cost at x. The solution in the special case $F(x) = 1$ expresses the signed distance from the boundary. Reformulating (1) with the relation above results in the variational formulation of the Eikonal equation

$$\sqrt{(\nabla u(x))^T M(x) \nabla u(x)} = 1, x \in \omega \tag{1.3}$$

describing a traveling wave through the domain ω with given heterogeneous, anisotropic velocity information M. The solution $u(x)$ denotes the time when the wave arrives at point x. The computational domain ω for solving the Eikonal equation is discretized by planar-sided elements whose vertices are the discretization points storing the discrete solution. The continuous solution is represented by a linear interpolation within each element.

Chapter 2

Data structures and external libraries

2.1 Vtk File

As input and output we used the .vtk file format, which is a file format for 3D computer graphics and is used in computational fluid dynamics, and similar. The file format is composed of a header and a data section. The header contains the metadata of the file, like the type of data, the number of points, the number of cells, etc. The data section contains the actual data, like the coordinates of the points, the connectivity of the cells, etc. The file format is very flexible and can be used to store different types of data, like scalars, vectors, tensors, etc. We used the vtk file format to store the mesh data, like the coordinates of the points, the connectivity of the cells, etc. We also used the vtk file format to store the solution data, like the solution values at the points, etc. For simplicity, we used the ASCII version of the vtk format, and we support only uniform meshes (i.e., all the cells must have the same number of vertices).
Official reference for the .VTK file format

2.2 Parsing data structures

These data structures are used to represent the structure of a parsed .vtk file. The main purpose of these data structures is to be a connection between the data structure used as computational reference and the file representation of data in the .vtk file.

2.2.1 VtkPoint

Represents a point in 3D space. Contains coordinates x, y, and z. Can optionally store additional data in a vector.

```
struct VtkPoint {
    double x, y, z;
    std::vector<double> data;

    VtkPoint(double x, double y, double z,
            const std::vector<double>& data = {})
            : x(x), y(y), z(z), data(data) {}
};
```

2.2.2 VtkCell

Represents a cell in the unstructured grid. Contains a vector of point IDs that define the cell. Stores the cell type. Can optionally store additional data in a vector.

```
struct VtkCell {
    int type;
    std::vector<int> point_ids;
    std::vector<double> data;

    VtkCell(int type = 0, const std::vector<int>& point_ids = {},
            const std::vector<double>& data = {})
        : type(type), point_ids(point_ids), data(data) {}
};
```

2.3 Eigen

Eigen libraries were used to dispatch the computation of matrix multiplications that occurs in the implementation, The mostly (and uniquely) data structure defined in Eigen libraries and used in the implementation is:

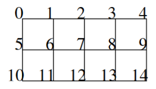
```
Matrix <typename Scalar,
        int RowsAtCompileTime,
        int ColsAtCompileTime>
```

For all the details of this data structure we refer to the Reference documentation

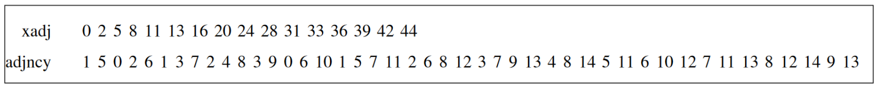
2.4 Computation data structures

For all computational purposed we made use of CSR-like format data structures. This data structure is composed by two vectors:

- An adjacency list
- An id-pointer list, where each values refers to the starting position of a ”zone of interest” into the adjacency list



(a) A sample graph



```
xadj     0 2 5 8 11 13 16 20 24 28 31 33 36 39 42 44
adjncy   1 5 0 2 6 1 3 7 2 4 8 3 9 0 6 10 1 5 7 11 2 6 8 12 3 7 9 13 4 8 14 5 11 6 10 12 7 11 13 8 12 14 9 13
```

Figure 2.1: Adjacency vector pair for a graph, from metis documentation

To represent the mesh, we used:

- A CSR where pointers refers to elements and adjacency list refers to all points each element is composed by
- A CSR where pointers refers to points and adjacency list refers to all elements each point is part of
- A vector that contains for each point the final value (that in our case is the time-to-reach of wave front)

In all the global solver implementations that will make use of patch-subdivided mesh 4.4, CSR will be used also to represent:

- For each patch the patches it has a border in common
- For each patch the element it is composed by
- For each patch the point it is composed by

To divide the mesh into patches, we used the software library Metis to produce the subdivision, ad-hoc algorithm have been implemented to transform patch-subdivision into adjacency lists.

Chapter 3

Local Solvers

3.1 Local solvers

We will refer to local problem as the computation of the wave travel time into a single element. Our implementation of local solvers is able to interact with triangular or tetrahedral elements, since triangular case can be trivially obtained by tetrahedral one we will focus only on the analysis and implementation details of the case of tetrahedral elements. It's also important to specify that for numerical consistency it's fundamental that all the elements the local solver is acting on are **acute tetrahedron**.

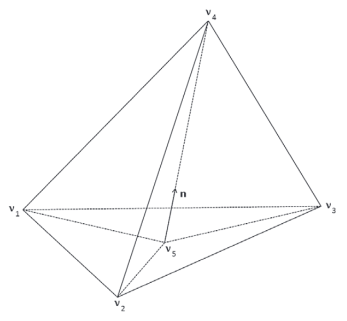


Figure 3.1: Diagram representing components of local solver

Be:

- **Point**: tuple $x_i = (x, y, z)^T$
- **Edge**: segment connecting two points $e_{ij} = x_j - x_i$
- **Time-value**: the wave front arrival time at a point $\Phi_i = \Phi(x_i)$
- **Velocity matrix**: 3×3 symmetric positive-definite matrix encoding the speed information M

Know:

- All x_i position of tetrahedron vertexes
- Time-values Φ_i of vertexes 1,2,3
- The velocity information M

The target is to find the Time-value $\mathbf{\Phi_4}$.
Since velocity is constant we can define the time travel from a point to another as:

$$\Phi_{ij} = \Phi_j - \Phi_i = |e_{ij}|_M = \sqrt{e_{ij} M e_{ij}} \tag{3.1}$$

In order to find the value Φ_4 we will use an upwind scheme that takes in account a fifth point x_5 placed on the planar face defined by points x_1, x_2, x_3 whose position and time-value are defined by a weighted average as follows:

$$x_5 = \lambda_1 x_1 + \lambda_2 x_2 + (1 - \lambda_1 - \lambda_2) x_3 \tag{3.2}$$
$$\Phi_5 = \lambda_1 \Phi_1 + \lambda_2 \Phi_2 + (1 - \lambda_1 - \lambda_2) \Phi_3 \tag{3.3}$$
$$\text{with} \quad \lambda_1 + \lambda_2 <= 1 \tag{3.4}$$

Now we can define Φ_4 as:

$$\Phi_4 = \Phi_5 + \Phi_{45} \tag{3.5}$$
$$= \lambda_1 \Phi_1 + \lambda_2 \Phi_2 + (1 - \lambda_1 - \lambda_2) \Phi_3 + \sqrt{e_{45} M e_{45}} \tag{3.6}$$

Local problem is now transformed into find λ_1 and λ_2 both $0 \leq \lambda_i \leq 1$ such that Φ_4 is minimum (Constrained optimization problem).
In order to get a better problem representation we trivially derive:

$$e_{45} = x_5 - x_4 = [e_{13}, e_{23}, e34]\lambda \tag{3.7}$$
$$M' = [e_{13}^T, e_{23}^T, e_{34}^T]^T M [e_{13}, e_{23}, e_{34}] \tag{3.8}$$
$$t = [\Phi_{13}, \Phi_{23}, \Phi_3] \tag{3.9}$$

To get:

$$\Phi_4 = t\lambda + \sqrt{\lambda^T M' \lambda} \tag{3.10}$$

Since (3.9) is a parabolic equation in λ_1, λ_2, we are sure that a global minimum exists, we can then use a descendent direction method in order to approach the solution (we still have to deal with constraints, but this will be solved in implementations choices).
We need also to introduce the following definitions to better understand the implementation of local solver:

$$M' = [\alpha, \beta, \gamma] \tag{3.11}$$
$$= \begin{cases} \alpha = [e_{13}^T M e_{13} \;\; e_{23}^T M e_{13} \;\; e_{34}^T M e_{13}] \\ \beta = [e_{13}^T M e_{23} \;\; e_{23}^T M e_{23} \;\; e_{34}^T M e_{23}] \\ \gamma = [e_{13}^T M e_{34} \;\; e_{23}^T M e_{34} \;\; e_{34}^T M e_{34}] \end{cases} \tag{3.12}$$

So that the partial derivative of (3.9) with respect to λ_1, λ_2 can be expressed as:

$$D(\Phi_4, \lambda_1, \lambda_2) = \begin{cases} \partial_{\lambda_1}\Phi_4 = -\Phi_{13} + \lambda^T\alpha/\sqrt{\lambda^T M'\lambda} \\ \partial_{\lambda_2}\Phi_4 = -\Phi_{23} + \lambda^T\beta/\sqrt{\lambda^T M'\lambda} \end{cases} \tag{3.13}$$

3.1.1 Local solver

Local solver aims to found the solution of the equation:

$$D(\Phi_4, \lambda_1, \lambda_2) = 0 \qquad 0 \le \lambda_i \le 1 \tag{3.14}$$

But the solution does not always exists due to constraints.
What the local solver will really do is use a quasi-newton method in order to approach the solution of the two variable problem $D(.) = 0$ and in case solution falls of constraints it will correct the solution accordingly, falling on a single variable problem $D(0, .) = 0$ if needed.
In order to use a two-variable quasi-newton method we also need to compute Jacobian matrix of $D(.)$:

$$J(\Phi_4, \lambda_1, \lambda_2) = \begin{bmatrix} \alpha_1 - \Phi_{13} * \frac{\lambda^T\alpha}{\sqrt{\lambda^T M'\lambda}} & \beta_1 - \Phi_{13} * \frac{\lambda^T\beta}{\sqrt{\lambda^T M'\lambda}} \\ \alpha_2 - \Phi_{23} * \frac{\lambda^T\alpha}{\sqrt{\lambda^T M'\lambda}} & \beta_2 - \Phi_{23} * \frac{\lambda^T\beta}{\sqrt{\lambda^T M'\lambda}} \end{bmatrix} \tag{3.15}$$

For the single-variable problem solver method we opted for a bisection method, which have also to take account for constraint checks.
Starting point iterative methods has been chosen as $\lambda_1, \lambda_2 = (0, 0)$ in order to make constraints checks and fall on bisection works.

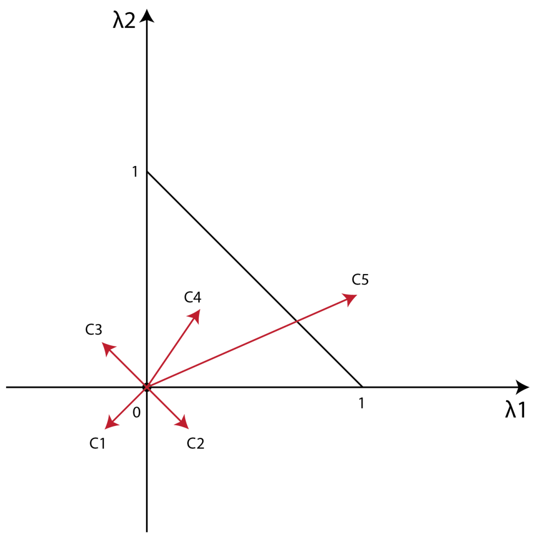


Figure 3.2: Case scheme of method execution

Follows definition of LOCAL SOLVER:

```
M’ = [Alpha, Beta, Gamma] // M-prime PRECOMPUTED MATRIX
t = [t13, t23, t3] //TIME DIFFERENCE VECTOR
l1 = 0; //LAMBDA 1
l2 = 0; //LAMBDA 2
do {
   // LAMBDA-ALPHA, LAMBDA-BETA
   La = [l1 l2 1] * Alpha
   Lb = [l1 l2 1] * Beta
   // SQRT(Lambda_T * M’ * Lambda)
   Dist = M-distance(l1,l2)
   //FIND RESIDUE OF SOLUTION
   Res = [ -t13 * Dist + La, -t23 * Dist + Lb]
   //BUILD JACOBIAN
   J = [Alpha[0] - t13 * La / Dist, Beta[0] - t13 * Lb / Dist;
        Alpha[1] - t23 * La / dist, Beta[1] - t23 * Lb / Dist]
   //GET DESCENDENT DIRECTION
   Dir = - J.inverse() * Res
   //UPDATE SOLUTION
   l1 += Dir[0]; l2 += Dir[1]
   //CHECK CONSTRAINTS
   if(l1 <= 0){
      l1 = 0
      if(l2 <= 0){
         l2 = 0 // CASE 1
      } else{
         // IF ONLY LAMBDA1 FALL OFF CONSTRAINT USE
         // BISECTION TO FIND SOLUTION OF LAMBDA2
         l2 = Bisection(0, l2, t, M’) // CASE 3
      }
      // SOLUTION HAS BEEN FOUND NO NEED TO DO MORE ITERATION
      break;
   } else{
      if(l2 <= 0){
         // IF ONLY LAMBDA2 FALL OFF CONSTRAINT USE
         // BISECTION TO FIND SOLUTION OF LAMBDA1
         l2 = 0
         l1 = BISECTION(l1, 0, t, M’) // CASE 2
         break;
      }else if(l1 + l2 >= 1){
         // IF BOTH FALL OFF CONSTRAINTS USE BISECTION
         // TAKING ACCOUNT OF RELATION l1 = 1 - l2
         l1 = BISECTION(l1, 1-l1, t, M’) // CASE 5
         l2 = 1-l1
         break;
      }
   }
   //IF SOLUTION FALLS INTO CONTRAINT THEN CHECK IF
   //THE DIRECTION HAS NORM SMALLER THAN TOLLERANCE AND
   //DO ANORTHER CYCLE OF QUASI-NEWTON METHOD
   ITER-- //CASE 4
} while (Dir.norm() > TOLLERANCE && ITER > 0)
```

Follows definition of BISECTION:

```
BISECTION(l1, l2, t, M’) {
   a = 0;
   b = 1;
   f = GET_BORDER_FUNCTION_DERIVATIVE(l1,l2,t,M’)
   fa = f(a);
   fb = f(b);
   // IF THE FUNCTION DOES NOT INTERSECT THE ZERO-AXIS THEN
   // TAKE THE EXTREMIS THAT HAS LOWER VALUE.
   // THIS MEANS THAT DERIVATIVE HAS LOWER VALUE SO THE
   // SOLUTION IS CLOSER TO THE OPTIMAL ONE
   if (sameSign(fa, fb)) {
     if (abs(fa) < abs(fb))
         return 0;
     else
         return 1;
     }
   }
   // OTHERWISE DO SOME BISECTION ITERATIONS UNTILL WE
   // HAVE APPROACHED ENOUGH THE ZERO OF FUNCTION
   do {
     k = (a + b) 2;
     res = f(k);
     if (sameSign(fa, res)) {
         a = k;
         fa = res;
     } else {
         b = k;
     }
     ITER--;
   } while (abs(res) > TOLLERANCE && ITER > 0);
   return k;
}
```

3.1.2 Optimized local solver

The local solver described in section 3.1.1 is limited on a general assumption: the spatial configuration of points x_1, x_2, x_3, x_4 is fixed and so it's the direction of the upwind scheme we follow to find the solution to the problem, this is reflected on the configuration of M' matrix and t vector that are used all along local solver method, making inefficient it's use if we need to find the solution for multiple vertex of the same element one after another (since we have to compute multiple time all the values of M' one for each configuration). Moreover we can notice that the matrix M' is symmetric, so there is no need to compute and store all the elements of the matrix.
We will now introduce some optimizations that act on M' matrix and that can increase the performance of local solver for some application (mostly those where data locality and parallelism are very important).
All following optimization are based on the [(2018) 7], although some corrections have been done in order to fix discrepancies in M' optimized configurations with reference to standard ones, that used to brake local solver method. For convenience we will only describe the main idea behind this optimization and the correction we made, for all the mathematical background and algorithm description we link to [(2018) 7].

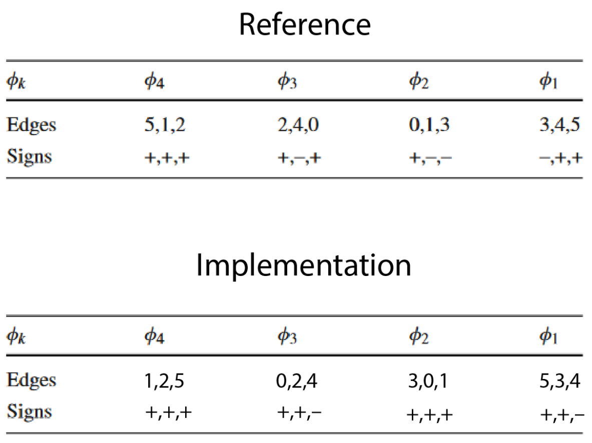
- **Precompute inner products**
  Since different M' configuration of the same element share entries values, we can precompute all the possible M' entries for each element.
  Specific algorithms have been designed in order to retrieve only the entries needed for each specific configuration, see [(2018) Section 3.2 7].

- **Inner product compression**
  The number of precomputed entries we have to store is 18, but we can derive most of entries from a restricted selection which number equals to the number of edges of the element (6 for tetrahedral, 3 for triangular). For all details see [(2018) Section 3.3 7]

Two main changes have been applied to the reference algorithms:

- **Rotation of edges (gcodes)**
  With reference to [(2018) Section 3.2.2 Table 3 7] all edges groups have been left rotated by 1.

- **Merging of signs**
  With reference to [(2018) Section 3.2.2 Table 3 7] signs for the first and second edge of each group have been fixed to +, the third edge of each group is the sign-product of all signs of the group.



Reference

ϕ_k	ϕ_4	ϕ_3	ϕ_2	ϕ_1
Edges	5,1,2	2,4,0	0,1,3	3,4,5
Signs	+,+,+	+,–,+	+,–,–	–,+,+

Implementation

ϕ_k	ϕ_4	ϕ_3	ϕ_2	ϕ_1
Edges	1,2,5	0,2,4	3,0,1	5,3,4
Signs	+,+,+	+,+,–	+,+,+	+,+,–

Figure 3.3: Correction to reference scheme

The (Figure 3.3) shows how reference scheme has been changed in implementation, but notice that implementation still uses most of algorithms shown in [(2018) 7]. For convenience the non-trivial configuration of edges and signs for triangular elements is left at code implementation overview.

Chapter 4

Global solvers

4.1 Global solvers

From now on as "Global solver" we will refer to the algorithms that schedule the order of elements exploration in order to find the solution of the Eikonal problem for a mesh composed of multiple elements.
We implemented three different methods:

- Fast Marching Method (FMM)
- Fast Iterative Method (FIM)
- Patch - Fast Iterative Method (PFIM)

Other variant has been implemented in order to test the behaviour some methods with the application of optimization as parallelization and use of Optimized Local Solver.
For completeness we will list them, but detailed information will be left at code implementation overview and references overview.

- Fast Marching Method - Optimized (FMMO)
- Fast Iterative Method - Parallelized (FIMP)
- Patch - Fast Iterative Method - Cuda Implemented (PFIMC)

All methods works on both triangular and tetrahedral element composed meshes. All method takes as input the Mesh data and returns a vector which contains mesh-points-referenced time-solution values.

4.2 Fast Marching Method

The main idea for the Fast Marching Method is to iterates on all the points of the mesh only once. In order to keep consistency on the global solution the point-exploration has to be done following the time evolution of the wavefront, this

is achieved by keeping a list of "active points" (that represents the wavefront) ordered by increasing solution value and picking always the first one, removing it from the list and calculating the solution values for all the points witch it's connected to (called "neighbours"). Also a list of "explored points" is kept.
The procedure is the following:

```
ActiveList << Starting_points
while( not ActiveList empty ){
    active_point = least_value(ActiveList)
    explored[active_point] = true

    foreach (neighbour of active_point){
        solution = LOCAL_SOLVE(neighbour)
        if (solution < global_solution[neighbour]){
            global_solution[neighbour] = solution

            if (explored[neighbour] == false){
                ActiveList << neighbour
            }
        }
    }
}
```

In order to achieve a fast access to the element of active list with least value we have used a so called "MinHeap" data structure. Since we have to deal with only one point at a time to maintain the consistency of solution, this method cannot be applied to parallel execution implementations.
Notice that as "LOCAL_SOLVE(neighbour)" we refers to finding the solution of neighbour applying the Local Solver to all the elements the Active Point and the Neighbour are part of and keeping only the lower solution.

4.3 Fast Iterative Method

The main idea for the Fast Iterative Method is to extend the concept of FMM introducing a broader definition of wavefront, calling "Agglomerate" associated to a point P the One-Ring collection of points connected to P, we will call "Active List" the collection of points that are part of agglomerates that has changed solution between two consecutive iterations.

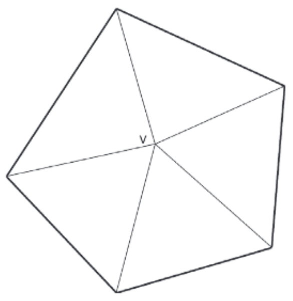


Figure 4.1: Agglomerate representation for point V

The procedure is the following:

```
foreach( point in StartingPoint){
    foreach ( neighbour in agglomerate(point)){
        ActiveList << neighbour
    }
}
while( not ActiveList empty ){
    foreach( point in ActiveList ){
        solution = LOCAL_SOLVE(point)

        if ( solution < global_solution(point) ){
            global_solution(point) = solution

            foreach ( neighbour in agglomerate(point) ){
                ActiveList << neighbour
            }
        }
    }
}
```

Notice that as "LOCAL_SOLVE(point)" we refers to finding the solution of "point" applying the Local Solver to all the elements "point" is part of and keeping only the lower one.
Since we are not dealing with constraint on solving one point at a time we can parallelize the iteration on all points of ActiveList in order to speedup the execution (FIMP).

4.4 Patch Fast Iterative Method

This method is an evolution of FIM, it has been introduced in order to deal with SIMD architectures that offers massive computational power but are less reactive to branching in execution. The concept of "agglomerate" is extended

to "patch" and the structure of the algorithm that manage active patches is the same of active points. The gain in performance is obtained thanks to the fact that PFIM can calculate "LOCAL_SOLVE(point)" (referring to the definition of FIM) for a large number of point all at once. Massive parallel local solution calculation for a large number of points have to be managed accordingly, in order to achieve that reduction lists have been introduces.
Follows the procedure for PFIM:

```
foreach( point in StartingPoint){
    ActiveList << Patch[point]
}
while( not ActiveList empty ){
    convergence_list = []

    foreach( patch in ActiveList ){
        foreach ( element in patch ){
            foreach ( point in element ){
                solution = LOCAL_SOLVE(point)
                reduction_list(point) << solution
            }
        }

        foreach ( point in patch ){
            best_solution = reduce(reduction_list(point))
            if ( best_solution < global_solution(point) ){
                global_solution(point) = best_solution
                convergence_list << patch
            }
        }
    }

    foreach ( patch in converged_list ){
        foreach ( neighbour of patch ){
            activeList << neighbour
        }
    }
}
```

Chapter 5

Analysis

We presents the results for the numerical tests in double precision performed on a workstation equipped with Ryzen 5800H (8-core 16-threads) CPU and Nvidia RTX 3070 mobile version. All of our tests are based on triangular and tetrahedral 3D mesh representing a toroid, created using gmesh using various element density values. Results and analysis will be shown for time dependency on number of nodes, and for the cpu parallel version also on number of threads used.

5.0.1 Parallel Methods

In this section we will show scalability of parallel methods (FIMP, PFIM, PFIMC) on two main dimensions: number of threads and size of patches subdivision, all test have been performed on big meshes:

- **Scalability on number of threads**

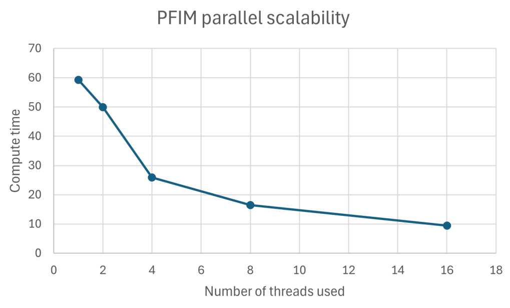


Figure 5.1: PFIM thread scalability triangular mesh

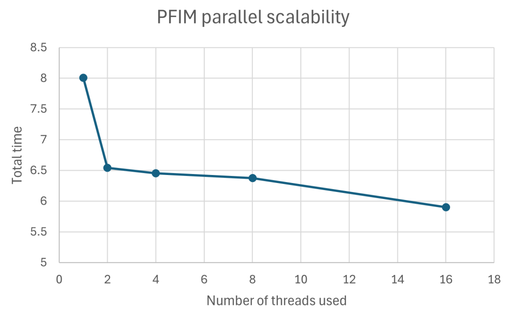


Figure 5.2: PFIM thread scalability tetrahedral mesh, note y axis start at 5

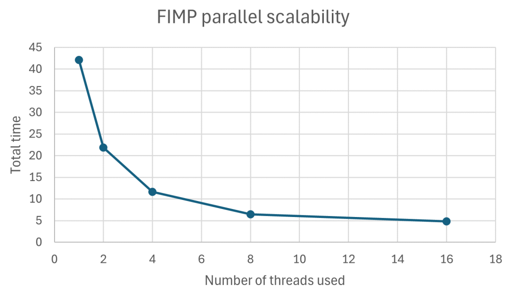


Figure 5.3: FIMP thread scalability triangular mesh

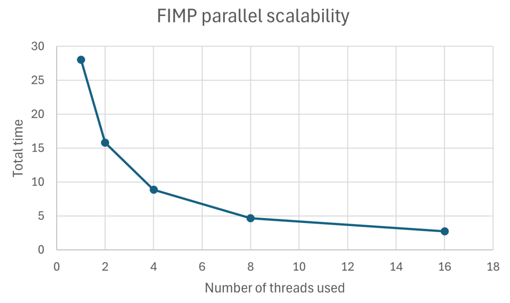


Figure 5.4: FIMP thread scalability tetrahedral mesh

- **Scalability on patch size**

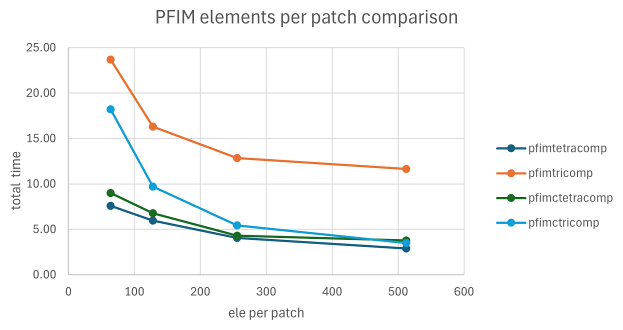


Figure 5.5: PFIM and PFIMC patch scalability

5.0.2 Methods comparison

In this section we will compare time execution of all methods. Notice that for parallel methods we used the best configuration for big meshes following the above results:

- FIMP - 16 threads
- PFIM - 16 threads - 512 elements / patch
- PFIMC - 512 elements / patch

- **Comparison total time on Mesh size**

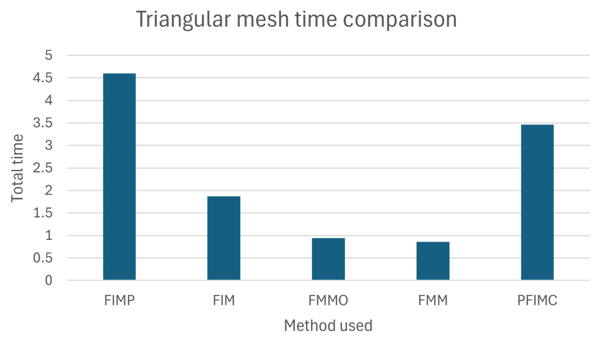


Figure 5.6: Triangular meshes total time

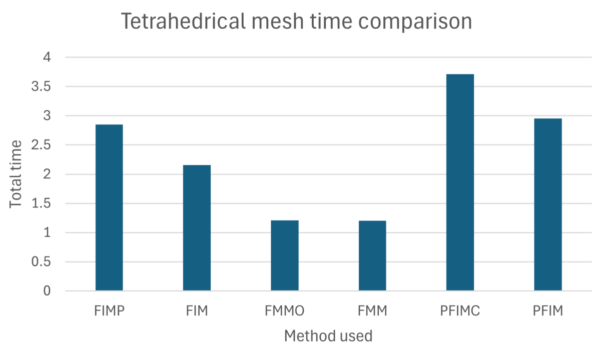


Figure 5.7: Tetrahedral meshes total time

- **Comparison computational time on Mesh size**

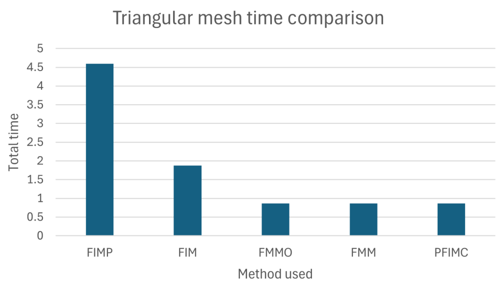


Figure 5.8: Triangular meshes compute time

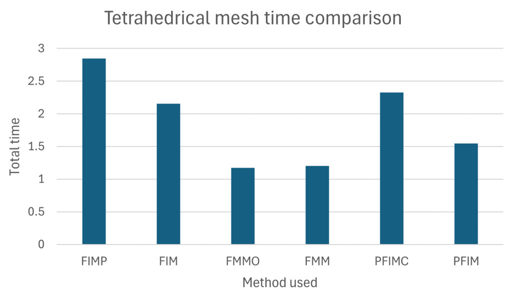


Figure 5.9: Tetrahedral meshes compute time

- **Comparison computational complexity**

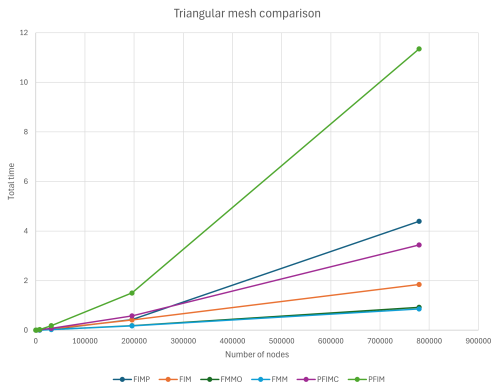


Figure 5.10: Triangular meshes total time

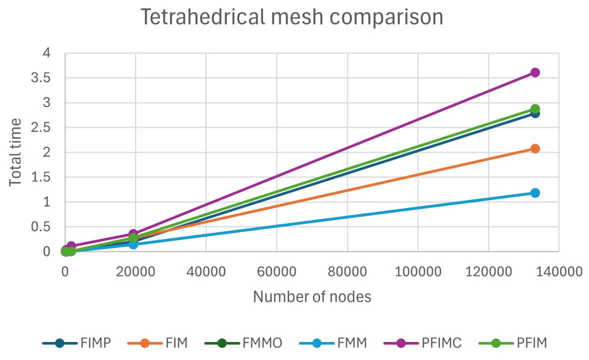


Figure 5.11: Tetrahedral meshes total time

5.0.3 Analysis conclusions

The thread scalability show sizable performance uplifts up the 8-threads mark, beyond this point, hardware limitations become evident- we run out of physical core, as indicated by the fattening of the performance curve. Regarding patch size scalability, the results are favorable up to a patch size of 512. However, beyond this size, there is an inefficient allocation of block within the GPU SMs (Streaming Multiprocessors). When the patch size exceeds 512, the distribution of work across the SMs become sub optimal, leading to under utilization of the GPU's capabilities.

Chapter 6

Conclusion

In our current implementations, we were unable to achieve the significant speedups described in the reference paper for the domain decomposition methods, specifically PFIM and PFIMC. One of the main challenges we encountered was that the partitioning process alone could take more time than the complete execution of FMM. Consequently, FMM remains the fastest option unless repeated calculations with the same mesh are necessary.
Despite this, there is potential for improvement. PFIM, when executed with only 8 physical cores, demonstrated compute times similar to FMM. This suggests that with more powerful CPUs, we could achieve better performance and possibly exceed FMM speeds. It is also worth noting that our CUDA version was executed on mobile hardware, which inherently has power and performance limitations. Therefore, running the CUDA implementation on desktop hardware, which is not constrained by such limitations, could yield significantly better results. Additionally, there is potential to use a parallelized version of the partitioning methods, even on GPUs. By leveraging the parallel processing capabilities of GPUs, we can significantly speed up the partitioning process, thereby enhancing the overall performance of the domain decomposition methods. This approach could further narrow the performance gap between PFIM, PFIMC, and FMM, especially for large-scale problems where partitioning becomes a major bottleneck.

Chapter 7

References

- W. K. Jeong and R. T. Whitaker, A fast iterative method for eikonal equations, SIAM J. Sci. Comput., 30 (2008)
- Z. Fu, W.-K. Jeong, Y. Pan, R. M. Kirby, and R. T. Whitaker, A fast iterative method for solving the eikonal equation on triangulated surfaces, SIAM J. Sci. Comput., 33 (2011)
- Z. Fu, Kirby, R.M., Whitaker, R.T.: Fast iterative method for solving the Eikonal equation on tetrahedral domains. SIAM J. Sci. Comput. 35(5), C473–C494 (2013)
- Ganellari, D., Haase, G. and Zumbusch, G. A massively parallel Eikonal solver on unstructured meshes. Comput. Visual Sci. 19, 3–18 (2018)