Parallelization of 4T-LE Refinement

Bhargav Sriram Siddani*, Christian Porrello†, Aakash Patel‡, and Vinay Kumar§
*College of Mechanical and Aerospace Engineering
University of Florida, Gainesville, Florida 32611
Email: siddanib@ufl.edu
†College of Mechanical and Aerospace Engineering
University of Florida, Gainesville, Florida 32611
Email: cporrello@ufl.edu
‡Department of Mechanical Engineering
Indian Institute of Technology
Bombay, Mumbai 400076, India
Email: apatel.ittbombay@gmail.com
§Department of Aerospace Engineering
Indian Institute of Technology
Bombay, Mumbai 400076, India
Email: vkbommoju@gmail.com

***Abstract*—This works deals with the Parallelization of a 2D Triangular mesh using the 4T-LE Refinement method. The parallelization was done using MPI. The 4T-LE Refinement algorithm and the methodology used to parallelize this algorithm are discussed. The parallel method is analysed for different meshes sizes and the corresponding results have been presented. Ultimately, while this work's parallel implementation yielded substantial runtime improvements over serial code, the scaling efficiency can be improved. Lastly, suggestions are made for future works.**

***Index Terms*—Adaptive Mesh Refinement, 4T-LE, OpenMP, OpenMPI**

I. Introduction

In the study of partial differential equations using computational grids, it is common practice to refine the grid to obtain more accurate results. It is very much essential to refine in regions of high solution activity in order to capture the involved physics and also to obtain more accurate results. However, in the study of problems which display highly localized or volatile phenomena, it is often difficult to predetermine where high solution activity will occur and efficiently allocate computational resources accordingly. Consequently, adaptive mesh refinement (AMR) techniques have been developed to refine the grid as the solution progresses. AMR techniques take advantage of error estimates to selectively refine the grid in regions of high error, thus more efficiently allocating computational resources than a uniform mesh refinement. Since their introduction, AMR techniques have been commonly used in numerical analyses.

However, as the scale of problems increase and more codes take advantage of parallel cluster computing, new and efficient parallel AMR algorithms must be developed to take advantage of parallel computing. Due to the complexity of communication in parallel cluster computers, AMR techniques are susceptible to race conditions and synchronization issues when implemented in parallel [1]. Futhermore, since AMR modifies the grid in a nonuniform manner, data structures must be devised that can be updated quickly [2].

Subsequently, this work presents the implementation of the Four Triangles Longest Edge (4T-LE) partition method in parallel. Here, an object-oriented programming approach was used to develop code for mesh refinement. For implementation in a parallel processing environment, the OpenMP and OpenMPI communication protocols were used to implement the method separately within a multi-core node and between nodes, respectively. The results were validated with serial version. A speed-up study was performed on mesh refinement on a 2D grid to determine the scaling efficiency of both implementations, and suggestions are made for future improvements.

II. 4T-LE Refinement in Serial Computing

The 4T-LE mesh refinement method, also known as Rivara refinement [3], is primarily used for the refinement of unstructured and structured conformal triangular meshes. This method has also been extended to 3D geometries. [4].

A. 4T-LE Refinement of Individual Elements

Fig. 1 roughly outlines the 4T-LE procedure for refinement of a unstructured conformal grid. Triangles are marked for refinement based on error indicators of the solution. It is common practice to refine the grid in regions of high solution gradients, which is the convention adopted in this work.

B. Propagation to Adjacent Elements

Observe in Fig. 1 that as a particular element gets refined, new nodes are introduced at the sides of the element. These midpoint nodes are not necessarily conformal, and subsequently adjacent elements must be refined to maintain conformity. The succession of refinement of adjacent triangular

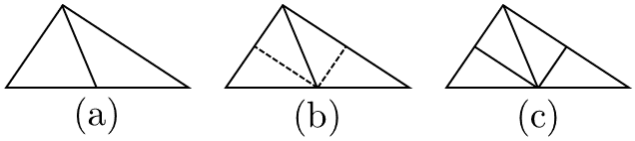

Fig. 1. 4T-LE refinement of a triangular element: (a) the triangle is bisected by the line drawn between the midpoint of the longest edge to the opposite vertex, (b) either of the two consequent triangles may be bisected accordingly to preserve conformity, (c) or both.

elements is known as propagation, and the order in which adjacent triangles get refined is known as the propagation path. Fig. 3 depicts the propagation path for an arbitrarily drawn structured mesh, demonstrating that although one element may be originally marked for refinement, several may be ultimately redrawn. In a study of properties of the propagation path for 2D unstructured meshes, Suárez *et al.* provide a proof that 4T-LE refinement asymptotically covers the entire mesh when the method is recursively applied [5].

C. Serial Implementation

For this work, the 4T-LE method was first implemented in serial following the procedure described by Plaza *et al.* [6]. Fig. 2 depicts the pseudocode used in this work's implementation.

D. Object Representation of Elements

In this work, an object-oriented approach was used to implement the 4T-LE method. Classes were established to represent vertices (Vertex class), edges (Edge class), and triangles (Triangle class). First, the Vertex class primarily contains the coordinate positions of the node it represents. Second, the Edge class primarily keeps track of triangles that share a particular edge. It stores information of the two nodes which define the edge and also the information about nodes that are required to form triangles sharing this edge. The Edge class also contains information of the type of the edge, that is whether the edge is on the boundary or interior of the grid. Lastly, The Triangle class primarily stores information of the three nodes that form the element and a flag which indicates if the triangle is to be refined or not. If an edge of a triangle is bisected, the triangle object also stores the information of the new point which has bisected its edge.

Thus, the implemented codes basically work with dynamic arrays of types Edge and Triangle which respectively contain the edges and triangles of the entire domain. The concept of std::map was used to do an efficient retrieval of information at the desired situation in the codes.

```
1:  for each triangle t to be refined do
2:      all edges e of t are bisected by adding points p
3:      for every p do
4:          let t* denote the adjacent triangle to t that shares
            p
5:          let q be the longest edge midpoint of t*
6:          while p ≠ q do
7:              add q to t*
8:              p ← q
9:              t ← t*
10:         end while
11:     end for
12: end for
13: for each t with bisected edge(s) do
14:     perform subdivision of t
15: end for
16: generate refined mesh
```

Fig. 2. pseudocode of serial 4T-LE

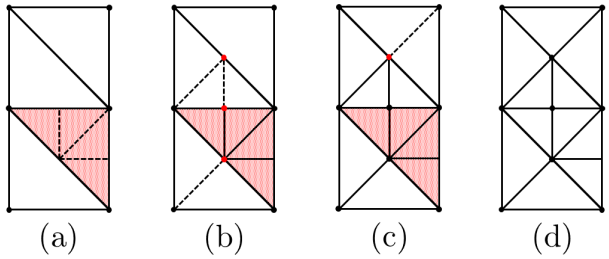

Fig. 3. propagation of 4T-LE refinement: (a) an element is marked (pattern fill) for refinement and bisected, then (b) nonconforming nodes are generated and adjacent triangles must be bisected to maintain conformity, and (c) refinement must occur further in the mesh as nonconforming nodes are generated until (d) a conforming mesh is generated.

III. 4T-LE Refinement in Parallel Computing

Because of the irregular nature of 4T-LE refinement, implementation of the method in parallel is challenging. Synchronization issues will occur if not properly accounted for, and efficient data structures must be developed that can update quickly as new elements get created in the mesh.

A. Addressing Conflicting Propagation Paths

The propagation path in 4T-LE refinement presents the primary challenges of implementing the method in parallel. To maintain conformity in the grid, triangular elements must be refined past those originally marked for refinement, and each element may have its own propagation path. Consequently, as independent threads and processors refine the propagation pass, synchronization issues arise. Rivara *et al.* [7] report two primary synchronization issues: (i) collisions due to overlapping propagation paths and (ii) inconsistencies in data structures due conflicting grids from the independent processors.

Although difficult, overcoming these two synchronization issues is tractable. In one previous work, a parallel implementation of the 4T-LE method focused on predetermining the propagation path of each element, predicting synchronization issues, and addressing any issues prior to refinement of the grid [2]. In another implementation by Castaños and Savage, refinement and synchronization issues were addressed simultaneously [1]. The parallel implementation of the 4T-LE method in this work adopts a similar methodology to the latter.

In the work by Castaños and Savage, a global termination criterion was established for the communication between processors to avoid synchronization issues during refinement. The termination criterion is based upon message acknowledgements for each processor, and is considered when every processor has: (i) acknowledged all received messages from adjacent processors, (ii) received acknowledgements for each message that it has sent, and (iii) each individual processor has finished refining the propagation paths it is responsible for. Once the termination criterion is met for an individual processor, any inconsistencies in the grid can be resolved and the processor is considered inactive. Once all processors are inactive, the mesh refinement is complete.

B. Parallelization of 4T-LE Refinement

In this work, the 4T-LE method is implemented in parallel (distributed memory framework) by distributing equal regions of the mesh to each processor prior to refinement of the mesh. Then, each processor is responsible for refinement of the mesh within the regions it has been allocated. In this implementation, only edges that are shared among processors need to communicate information about the shared edge. Consequently, the amount of information that has to be shared among the processors is reduced. Fig. 4 depicts pseudocode for the MPI implement ion in this work.

C. OpenMPI

To communicate information between processors, the Open-MPI message passing interface (MPI) library was used. Open-MPI is an open-source implementation of the MPI library which conforms to full MPI standards. In the MPI library specification [8], data is shared between independent processors. In a distributed memory environment, each processor runs its own copy of the code and MPI serves to communicate data in the memory of one processor to the memory of another. Further functions and subroutines are provided in the specification to support a processor hierarchy and efficient communication.

1: let P denote the process rank and P_0 denote the main rank
2: Every P gets a sub domain of mesh
3: Every P does bisection of edges of every t to be refined similar to serial method
4: Every P detects the edges it shares with other $P(s)$
5: let q denote a shared edge in P and q^* be a bisected q
6: **while** there is q^* in "ANY" P that has not communicated with its pair in another P **do**
7: every $P \neq P_0$ sends its list of q to P_0
8: communication of bisection state between corresponding q in P_0
9: P_0 sends respective q list to each P
10: every P does bisection process for every q^* in its q list
11: **end while**
12: every P subdivides its set of t similar to serial method
13: Every $P \neq P_0$ transfers its list of e and t info to P_0
14: P_0 generates refined mesh

Fig. 4. pseudocode of parallel 4T-LE in distributed memory framework

IV. Results

A. Validation of the Code

To validate the refinement code, a simple test case of a square was considered. The square was divided into 4 triangles as the initial mesh, and different elements were flagged to be refined. Fig. 5 depicts one such square in which the bottom triangle was flagged for refinement. Fig. 6 displays that 4T-LE bisection of the bottom element has occurred and refinement has propagated to the left and right elements to maintain conformity. The top triangle remains unrefined. Thus, Fig. 6 depicts agreement with the procedure outlined for 4T-LE refinement.

B. Applications of the Work

To demonstrate that the 4T-LE bisection code can adaptively refine a mesh, the mesh refinement code was used in conjunction with a solver for Poisson's equation which was also created as part of this work. The solver numerically approximates the solution to a 2D linear heat transfer problem with a heat source. The solver can take in boundary conditions (BC) like fixed temperature, convection heat transfer, and constant heat flux BC which are specified in a text file named `BoundaryConditions.txt`.

The governing equation is solved using the Continuous Galerkin approach to a 3 node triangular system. To refine the mesh, the solver outputs a file which is read in by the refinement code with indicators to triangles which have to be refined. The decision to refine a triangle is based on the gradients of temperature along the x and y directions. As the performance of the solver is outside the scope of this work, the solver was only implemented in a serial computing environment.

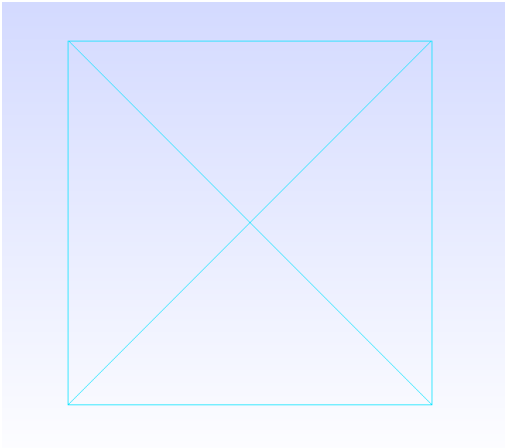

Fig. 5. example validation mesh before refinement. The mesh was generated and visualized using the GMSH software package [9].

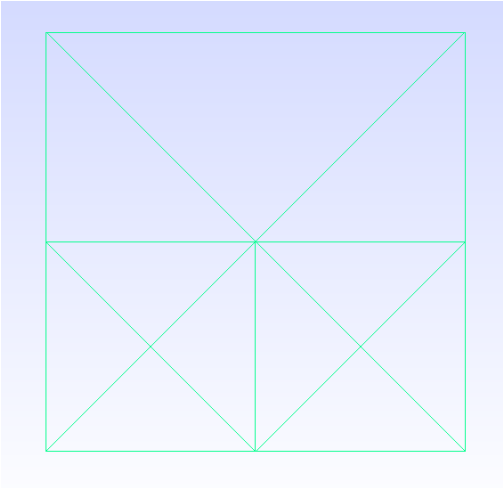

Fig. 6. example validation mesh after refinement. The mesh was generated and visualized using the GMSH software package [9].

In the MPI version of the code, the triangles which are to be refined are equally distributed among the processes. Fig. 7 displays the output mesh from the MPI version of refinement code which shows refinement applied only to triangles with high temperature gradients. Conformity has been preserved in the final mesh, indicating the 4T-LE bisection has successfully occurred to adaptively refine the mesh.

C. OpenMPI Results

The results presented in this subsection were run in a parallel computing cluster with 32 Intel® Xeon® CPU E5-2698 v3 @ 2.30GHz processors with a clock speed of 2799.896 MHz and cache size of 40960 kB.

Initially, the phenomenon of superlinear speedup was observed. Superlinear speedup is marked by a speedup greater than the number of processors. Superlinear speedup typically suggests poor serial computing implementation, the usual causes of which are large memory usage which increases memory access times and inefficient serial algorithm implementation. To reduce the effects of these causes, the O1 optimization flag was used which is included as part of the GCC compiler suite [10]. Table I demonstrates that using the O1 optimization flag dramatically reduced runtimes of the serial code. The O1 optimization was found to have a negative impact on the runtime of the MPI parallel code, and thus the flag was not used for those results.

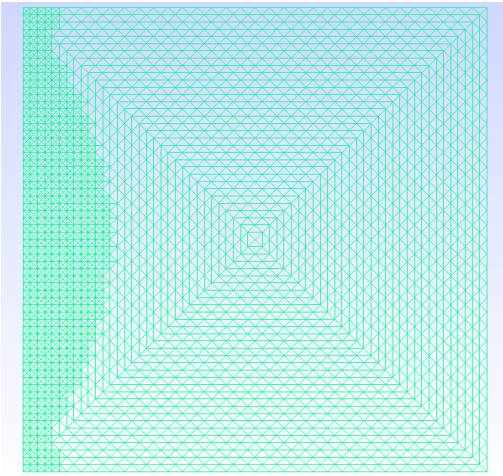

Fig. 7. 4T-LE adaptive refinement of a grid using high temperature gradients as a refinement flag

TABLE I
EFFECT OF O1 OPTIMIZATION FLAG ON SERIAL CODE RUNTIME

No. of Triangles	Unoptimized Runtime [s]	Optimized Runtime [s]
14490	445.33	26.11
20882	872.05	61.12

Subsequently, serial and parallel runtimes were measured for two initial mesh sizes, one of approximately 31K triangular elements and another of approximately 41K elements. Due to time limitations, larger initial meshes could not be tested. To measure runtimes, every element in the initial mesh was refined per the 4T-LE bisection method and the runtimes are recorded in Table II. It can be observed that as the number of triangles in the domain increases, the serial fraction at a given number of processors decreases. This indicates that the parallel version of the code is more efficient for larger number of triangles.

Subsequently, the experimentally determined serial fraction e was computed, which takes the parallel overhead into consideration. Fig. 8 shows the relationship between e and the number of processor for the OpenMPI implementation of the 4T-LE bisection method.

Lastly, the speedup was determined for the OpenMPI implementation of the 4T-LE method. The speedup is a measure of

TABLE II
AVERAGE RUNTIMES FOR UNIFORM MESH REFINEMENT

Initial Mesh Size	No. of Processes	Average Runtimes [s]
31K	1 (serial)	210.51
	2	181.00
	4	78.20
	8	72.93
	16	81.39
	24	100.67
	32	121.79
41K	1 (serial)	360.78
	2	306.06
	4	130.24
	8	121.61
	16	136.52
	24	177.12
	32	203.03

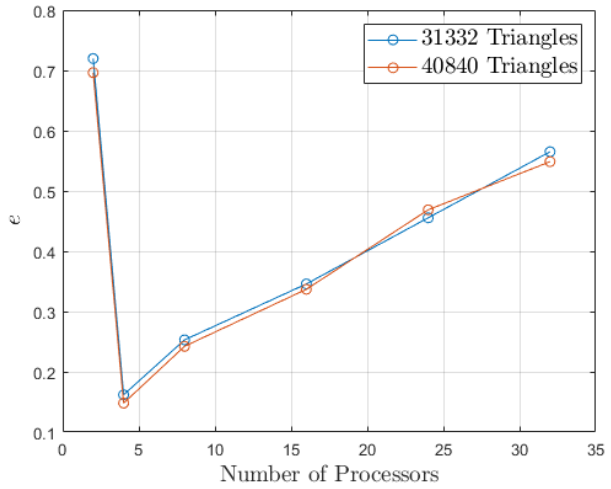

Fig. 8. e versus the number of processors.

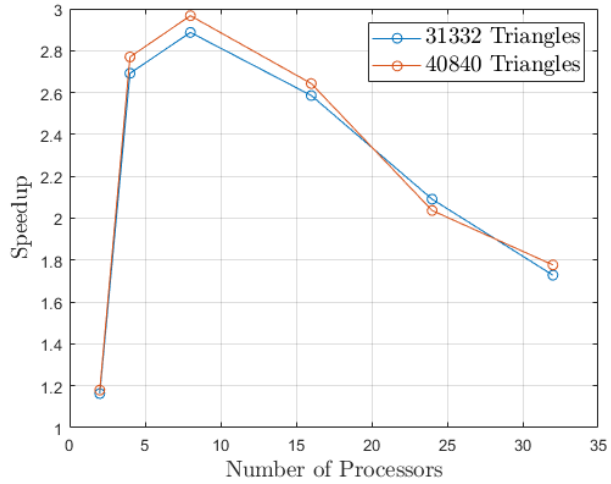

Fig. 9. Speedup versus the number of processors.

the performance improvement from parallelizing a serial code. Relative to other parallel implementations, the speedup may also suggest the efficiency of one parallel implementation over another, with higher speedups preferable. Fig 9 demonstrates that for this implementation, the highest speedup was observed with 8 processors.

V. Discussion

Fig. 9 demonstrates that for both initial meshes tested, the OpenMPI implementation is most efficient with 8 processors. Although theory predicts asymptotic behavior for the speedup of a code as more processors are utilized, the theory does not take into account practical considerations such as the cost of increasing communication between more processors. As expected, past the optimum of 8 processors, runtimes increase and speedup decreases as more processors must communicate information among themselves. With this implementation of the 4T-LE method, distributing the mesh to a larger number of processors also increases the number of edges that must be communicated. Therefore, although an optimum was observed rather than an asymptote, the observation agrees with expectations.

Fig. 8 demonstrates that e is at a minimum with 4 processors. The serial fraction for 2 processors is very high as it involves sending a single message of very large data of triangles. The serial fraction at 8 processors is more than that at 4 processors because of the "Blocking type" MPI Send and Receive functions involved in the code. Hence, the sweet spot for this code is at 4 processors. However, the implementation of "Non-Blocking type" MPI Send and Receive might actually move the sweet spot to a higher number of processors and this might also help in increasing the speedup. The scalability of this work in these conditions is only till 8 processors, however, as mentioned previously, changes in the type of information communication can actually increase the scalability to a much higher value.

For a future study, several modifications should be considered to improve the performance of this work's parallel 4T-LE implementation. First, an efficient geometrical decomposition of the domain which minimizes the amount of data transfer required should be considered. For this study, the mesh was read directly from the outputted GMSH mesh file and and the created array of data was equally distributed among the processes. Secondly, more efficient data structures should be considered as other data structures were outside the scope of this study. In this work's implementation, an object representation of elements was considered with the C++ maps container used for quick retrieval of information.

VI. Conclusion

The objective of this work was to implement parallel version(s) of the 4T-LE Refinement method, which was achieved using the MPI framework. An Object-Oriented Programming approach was used in the implementation of this method using C++. A serial version of 4T-LE method was first implemented to understand the algorithm and also to use it for the parallel version's analysis. The challenges in implementing a parallel version of this problem were discussed and addressed using a parallel algorithm. The effect of optimization flags on both serial and parallel versions were studied. Speedup, serial fraction, and scalability of the parallel version was studied in two different sizes of meshes. Ultimately, it was found that although the parallel implementation demonstrated substantial improvements over the serial implementation, the parallel version implemented in this work can be significantly improved using Non-Blocking Type of communication between processes. It has been observed that keen attention has to be paid to the type of communication and the amount of data that is being communicated. Also, geometrical decomposition of the entire domain can significantly reduce the required amount of data transfer, although this is outside the scope of the study.

Distribution of Work

- B. Siddani
  - Serial 4T-LE Code
  - OpenMPI 4T-LE Code and Results
  - Mesh Reader and Output Generator for Poisson Solver
- C. Porrello
  - Possion Solver
  - Mesh Output
  - Background Theory
  - Report/Presentation Formatting
- A. Patel
  - OpenMP Code
- V. Kumar
  - OpenMP Code

GitHub Repository

https://github.com/siddanib/4T-LE-Refinement

Acknowledgment

The authors would like to thank Dr. Roy, Professor, Department of Mechanical and Aerospace Engineering, University of Florida, and Dr. Gopalakrishnan, Assistant Professor, Department of Mechanical Engineering, Indian Institute of Technology-Bombay for providing an opportunity and resources to work on a topic of current interest and use. The authors would also like to thank University of Florida for providing access to HiPerGator, platform which was used to carry out this work.

References

[1] J. G. Castaños and J. E. Savage, "Parallel refinement of unstructured meshes", in Proc. IASTED Int. Conf. Parallel and Distributed Computing and Systems, Boston, Mass., Nov. 1999, pg. 1-11.

[2] B. Hatipoglu and C. Ozturan, "Parallel triangular mesh refinement by longest edge bisection", *Soc. for Ind. and Appl. Math. J. on Scientific Computing*, vol. 37, no. 5., pp. C574-88, Oct. 2015

[3] M. C. Rivara, "Mesh refinement processes based on the generalized bisection of simplicies", *SIAM J. on Numerical Anal.*, vol. 21, no. 3, pp. 604-13, Jun. 1984

[4] A. Plaza and G. F. Carey, "Local refinement of simplicial grids based on the skeleton", *App. Numerical Math.*, vol. 23, no. 2, pp. 195-218, Feb. 2000.

[5] J. P. Suárez *et al.*, "The propagation problem in longest-edge refinement", *Finite Elements in Anal. and Design*, vol. 42, no. 2, pp. 130-151, Nov. 2005.

[6] A. Plaza, J. P. Suárez, M. A. Padrón, "Fracticality of refined triangular grids and space-filling curves", *Eng. with Comput.*, vol. 20, no. 4, pp. 323-32, Oct. 2004.

[7] M. C. Rivara, P. Rodriguez, R. Montengegro, and G. Jorquera, "Multithread parallelization of Lepp-bisection algorithms", *Appl. Numerical Math.*, vol. 62, no. 4, pp. 473-88, Apr. 2012.

[8] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard, Version 3.1*, University of Knoxville, Tennessee, 2015.

[9] C. Geuzaine and J. F. Remacle, "GMSH: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities", *Int. J. Numer. Methods in Eng.*, vol. 79, no. 11, pp. 1309-1331, 2009.

[10] *Using the GCC Compiler for GCC versio 8.3.0*, GNU Press, Boston, MA., USA, 2019, pp. 115.