

Adaptive-Mesh-Refinement Pattern

I. Problem

Data-parallelism is exposed on a geometric mesh structure (either irregular or regular), where each point iteratively communicates with nearby neighboring points in computing a solution until a convergence has been reached. There is a system of formula that characterizes and governs the global and local behavior of the mesh structure, exposing each partition of mesh elements to different error and accuracy as the computation progresses in steps. Due to the varying error among the partitions, some mesh points provide sufficiently accurate results within a short number of steps, while others exhibit inaccurate results that may require “refined” computation at more fine-grained resolution. Efficiency is an important requirement of the process, thus it is necessary to adaptively refine meshes for selected regions, while leaving out uninteresting part of the domain at a lower resolution.

II. Driving Forces

1. **Performance** of the *adaptively* refined computation on the mesh structure must be higher than *uniformly* refined computation. In other words, the overhead of maintaining adaptive features must be relatively low.
2. To provide **accurate criterion** for further refinement, a good local error estimate must be obtained locally without consulting the global mesh structure. Since global knowledge is limited, useful heuristics must be employed to calculate the local error.
3. **Efficient data structure** needs to be used to support frequent structural resolution change and to preserve data locality across subsequent refinement.
4. Partition and **re-partition** of the mesh structure after each refinement stage must provide each processing unit balanced computational load and minimum communication overhead.
5. At each refinement stage, **data migration** and work stealing needs to be implemented for dynamically balancing the computational load.

III. Solution

1. Overview

The solution is an iterative process that consists of multiple components. First, we need an initial partition to divide the mesh points among the processing units. Second, an error indicator that will evaluate how close the locally computed results are to the real solution. Thirdly, when the error is above certain tolerance level, the partition needs to be “refined”, meaning that the mesh size will be reduced by a factor of a constant (usually by power of two). Fourth, as the

partition gets altered the mapping of the data elements to the processing units must also be adjusted for better load balance while keeping the data locality at the same time. All these components repeat themselves under efficient data structures designed for efficient access and locality preservation. The outline the algorithm for the Adaptive-Mesh-Refinement pattern, it looks as the following.

```
n = number of processors;
m = mesh structure;

Initially partition m over n processors;

while (not all partitions satisfy error tolerance) {
  compute locally value of partition p;
  // using the system of equations.

  for each mesh points mp in partition p {
    if errorEstimate(mp) > tol) {
      mark mp for refinement;
    }
  }
  refine mesh structure where marked;

  redistribute m OR
  migrate individual data between processors;
}
```

Algorithm 1. Adaptive-Mesh-Refinement Overview.

2. Error Estimate

To decide whether further refinement should take place, there must be decision criterion for each of the mesh points. Optimal criterion would be taking the difference between the exact solution and approximated solution at the given step and comparing it against a predefined tolerance value. However, in many cases the real solution is not provided for comparison with the approximated results. Also, since the criterion must be set for each individual mesh points, the error estimation has to be done locally (without global knowledge about the mesh structure). In practice several heuristics have been devised to estimate the local error for each of the mesh points. The estimate techniques should be chosen carefully depending on the accuracy and efficiency requirement of the implementation.

1) Gradient Based Estimate

In general, finer mesh resolution is required near discontinuities or steep curve of the solution. Using this observation, locating the positions that have large magnitudes of variance from the smooth curve serves as a good heuristic that indicates the regions for further refinement. At each of the steps, each mesh point can estimate a conjugate gradient using finite element approximation with the neighboring points. The magnitude of the gradient will be measured and compared against the tolerance value. This estimate effectively leaves out the smoother part of

the solution, and concentrate more computational effort on the steeper part of the problem domain.

2) Reconstruction Based Estimate

This estimation technique starts with a construction of an improved approximated solution with finer resolution meshes. The true error is then assumed to be proportional to the difference between the original and the improved solution. This estimate provides reliable error for the smooth-curved solution. However, the cost of reconstructing the improved approximated solution is usually prohibitively high. As an alternative to this solution, the difference is sometimes taken from the previous iteration, where the resolution is actually higher.

3) Residual Based Estimate

This technique simply computes the mean or norm from a cluster of meshes to take the residual of a given mesh point from the mean. The strength of this technique is in the simplicity and cheap computational cost, which a lot of adaptive mesh refinement computations do require, but its accuracy in estimating the error is inferior to the gradient or reconstruction based technique.

3. Refinement

Using the carefully chosen error indicator, a mesh point that does not meet the error tolerance value can be flagged for further refinement at each iteration. Generally, all the flagging is done first, and then refinement takes place. This way, the abstraction between error indicator and the refinement stage is clearly separated. Refinement of a mesh point is simply done by evenly dividing the point with a predefined constant, and updating the data structure appropriately. A simple illustration of this can be shown in figure 1.

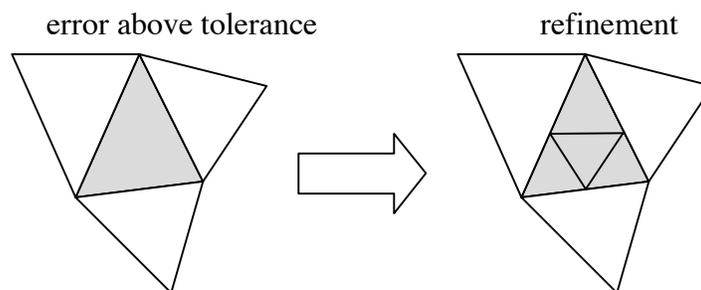


Figure 1. simple refinement strategy for triangular mesh by a factor of four.

The mesh points that do not go through the refinement is considered finished and left untouched in the subsequent iterations. Some processing units as a result could run out of useful computation in this many cases where all of the mesh points in the local partitions are below the error tolerance thus considered

completed. It is necessary to even out the computational load after each refinement stage. There are two ways of achieving the balanced load.

1) Repartition

Repartition involves re-evaluation of the whole mesh structure to partition the mesh points among the processing units. The goal of the partitioning is to minimize the inter-process communication (cut the mesh with less intersecting edges) and also improve the physical locality of the data for efficient memory operations. For detailed partitioning strategies, consult Dependency Graph Partition Pattern.

However, graph partitioning is known to be NP-Complete. That is, figuring out the optimal cut in the mesh to minimize communication over n processing elements is not known to be computable in polynomial time. Even if good heuristic information is used, it is likely that the overhead of computing the repartitioning is huge. Programmer must use caution and use this load balancing technique under desirable circumstances. Despite expensive cost, repartitioning ensure data locality is preserved at each partition, improving the performance of overall program.

2) Data Migration

Unlike repartitioning strategy where whole mesh structure has to be re-evaluated, a process can simply migrate the data directly to another after each refinement takes place. Distributed processes can communicate with each other in various ways (consult Work Stealing Pattern for further information) and distribute the load evenly among the units. Without the graph partitioning taking place, it will be generally more efficient to use this strategy. However, one disadvantage is that during the data migration locality is poorly preserved, which might become a factor for performance degradation.

4. Data Structure

Data structure in Adaptive Mesh Refinement Pattern must serve two purposes both regarding performance factors. It needs to provide small update cost as refinement takes place, and at the same time ensure each logically local data is also within close physical locality. There are two main data structures for maintaining mesh representation. Both have tradeoffs.

1) Quad or Oct-tree Representation

This is most widely used data representation among Adaptive Mesh Refinement implementations. Depending on the dimension of the data domain, a Quadtree (2D) or a Octtree (3D) is used where each child from a parent represents a partition generated by bisection at each dimension. The leaf nodes represent the mesh partition at most fine-grained level and coarse levels follow up to the root of the tree. Because of the tree structure, updating the refinement to the data structure can be done in a logarithmic complexity. Since each nodes are generated on fly and attached by pointers as refinement takes place, locality of the meshes

are relatively poor in terms of coarse partitions. Work distribution can be done easily by each processing unit being mapped to different subtrees, but it can lead to poor load balance if the refinement is unevenly clustered on the system. Example representation of quadtree representation is shown in Figure 2.

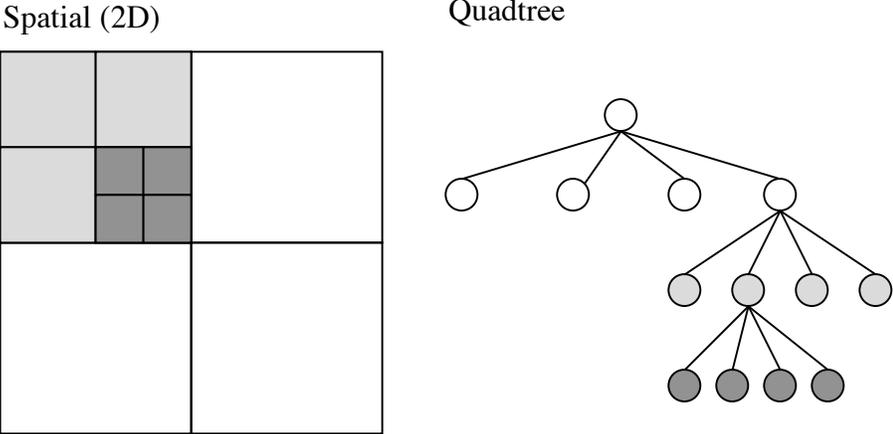


Figure 2. Quadtree representation for 2D spatial domain.

2) Linear Representation

Second option in representing spatial data is through simple linear array. N-dimensional space can be mapped to a 1-dimensional linear array by using a special walk order that packs the proximate data values into a physically local array. For a simple data domain with uniform mesh a linear mapping shown in Figure 3(a) can be used. As refinement takes place each of the refined mesh is replaced by newly generated child mesh. Even after the refinement update takes place, note that the locality is still preserved when the child data representation replaces the former data.

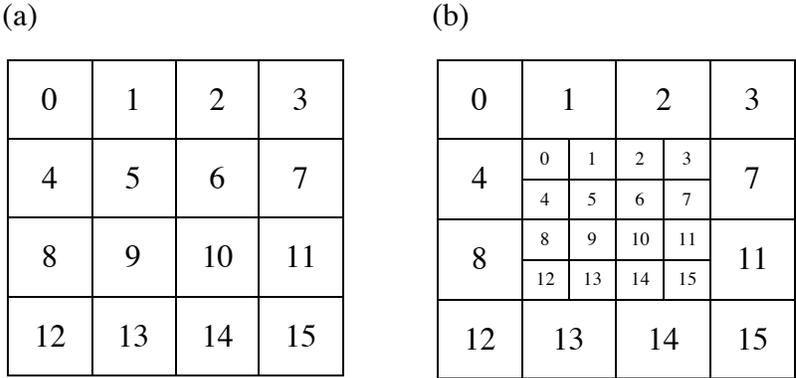


Figure 3 Mapping order from N-dimensional space to linear array.

- (a) {0 1 4 5 2 3 6 7 8 9 12 13 10 11 14 15}
- (b) {0 1 4 {0 1 4 5} 2 3 {2 3 6 7} 7 8 {8 9 12 13} 12 13 {10 11 14 15} 11 14 15}

Since logical locality and data locality are tightly coupled, mapping can be done simply by assigning each processing unit with a linear partition of data

elements. Despite excellent data locality, using a linear array must be compensated with extended search time. Hashing can be done to mitigate the search time, but complexity increases as a result. Programmer must use caution in discerning whether having data locality will outweigh the cost of complex update.

IV. Difficulty

1. Programmer must make a decision on which data structure to use, considering over factors such as data locality and update cost.
2. Designer must make a decision on which error estimate indicator to use, depending on the computational cost and accuracy of the method.
3. Designer must choose an appropriate load balancing strategy depending on the data locality and performance parameters.
4. If repartitioning is chosen, designer must make a decision on which graph-partitioning pattern to use.
5. If data migration is chosen, designer must take into consideration in which work-stealing pattern to use.
6. All the factors must orchestrate in order to provide adaptive solution with less performance overhead than uniformly refined solution.

V. Related-Pattern

Multilevel-Grid, Divide-and-Conquer, Partition and load balancing patterns

VI. Example

1. Finite Difference Method of Partial Differential Equation

Behavior of physical objects can be explained and predicted with system of differential equations. Some of the basic equations that model common natural phenomenon include Maxwell's equations for electro magnetic field, Navier-Stokes equations for fluid dynamics, Linear elasticity equations vibrations in an elastic solid, Schrodinger's equations for quantum mechanics, and Einstein's equations for general relativity. Partial differential equations(PDE) have partial derivatives of an unknown function with respect to more than one independent variables.

There are many methods in numerically achieving solution to PDE. Finite difference methods a widely used technique that discretize the continuum domain by overlaying a grid(mesh) over it. Algebraic equations are solved numerically by incrementally advancing from initial and boundary conditions of the equation to the solution space by taking difference and approximating various partial derivatives. In many PDE problems, solutions features which are of interest require high resolution and localized. Adaptive mesh refinement is necessary to efficiently refine and compute the domain of interest.

PDEs take place in the domain of time and space, forming a 2D or 3D space of data. Adaptive Mesh Refinement will start from a coarse grid with minimum acceptable resolution that covers the entire computation domain. As the solution progresses, regions in the domain requiring additional resolution are identified(flagged) and finer grids are overlaid on the regions of interest. The process is recursively continued with proper load balancing and re-partition until the desired accuracy is obtained.

2. Barnes-Hut N-Body Problem

Barnes-Hut algorithm is used in n-body particle simulation problem, to compute the force between each pair among n particles, and thereby updating their positions. The spatial domain is represented as Quad or Octal tree, and iteratively updated as each particle's position gets changed during the simulation. One special property that the algorithm exploits is the distance between the particles. If a cluster of particle is located sufficiently far away from another, its effective force can be approximated by using a center of mass of the cluster(without having to visit individual nodes). This effective force can be calculated recursively, thus reducing the number of traversals required to compute force interactions.

Although quite different from partial differential equations, Barnes-Hut also exhibit adaptive refinement aspect in a simply way: it adaptively updates the tree representation depending on the distance of the particles. Unlike PDEs that refines triggered by error condition, Barnes-Hut updates the representation depending on the distance of the particles. The "distance indicator" will act as a guideline in whether to refine or even retract a given data representation as the particle positions move throughout the iterations. As this happens proper load balancing and re-distribution must be followed to increase the performance of the program.