Computer Science 400
Parallel Processing
Siena College
Fall 2008

Topic Notes: Scientific Computing

Adaptivity and Linked Structures

So far, we have looked at distributed data structures that use only arrays. Arrays usually are the easiest to distribute; we simply assign a range of subscripts to each process. Quinn spends a lot of time discussing local vs. global indexing.

In your Jacobi solver, solution points were distributed evenly through the domain:

```
-------------------------------------
|                                   |
|   .   .   .   .   .   .   .   .   |
|                                   |
|   .   .   .   .   .   .   .   .   |
|                                   |
|   .   .   .   .   .   .   .   .   |
|                                   |
|   .   .   .   .   .   .   .   .   |
|                                   |
|   .   .   .   .   .   .   .   .   |
|                                   |
|   .   .   .   .   .   .   .   .   |
|                                   |
|   .   .   .   .   .   .   .   .   |
|                                   |
|   .   .   .   .   .   .   .   .   |
|                                   |
-------------------------------------
```

If you wanted a more accurate solution, you could add more solution points. But since your program used a uniform distribution of the points, better accuracy requires adding points everywhere.

```
-------------------------------------
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
| . . . . . . . . . . . . . . . . . |
-------------------------------------
```

While this is possible, it increases the total amount of work very quickly.

In some cases, we really only need more points in just parts of the domain. In a heat distribution problem, if there is a heat source in the northwest corner, we may only need more accuracy near that heat source. Fewer points may provide a sufficiently accurate solution further from the source.

```
-------------------------------------
| ..... . . . . . .   .   .   .   . |
| ..... . . . . . .   .   .   .   . |
| ..... . . . .   .   .   .   .   . |
| . . . . .   .   .   .   .   .   . |
| . . . . .   .   .   .   .   .   . |
| . . .   .   .   .   .   .   .   . |
| . . .   .   .   .   .   .   .   . |
| .   .   .   .   .   .   .   .   . |
| .   .   .   .   .   .   .   .   . |
| .   .   .   .   .   .   .       . |
| .   .   .   .   .   .   .         |
| .   .   .   .   .       .       . |
| .   .   .   .   .                 |
| .   .   .   .   .       .       . |
| .   .   .                         |
| .   .   .       .       .       . |
-------------------------------------
```

This is significantly more efficient in terms of the amount of computation we need to do to obtain a solution of acceptable accuracy.

If we know ahead of time where the extra work is needed, we could assign extra points there at the start of the computation. However, we often do not know this information. After all, we probably wouldn't be solving problems for which we already have a solution handy, so an *adaptive* approach is taken. Periodically, the accuracy of the solution can be checked, and extra points added as needed. In the context of the Jacobi solver, we may have a threshold for the greatest allowable difference in temperature between adjacent points. If the difference exceeds the threshold, points are added in that vicinity and the solution is recomputed.

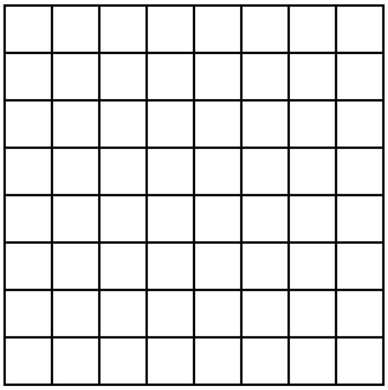
Adaptivity with arrays is difficult, as the completely regular structure is lost. In many cases, a *mesh* structure, often implemented as a linked data structure, is used instead of arrays.

Meshes come in a variety of types. Meshes consisting of quadrilaterals (hexahedra in three dimensions) are called *structured* meshes. Meshes constructed from triangles (tetrahedra in three dimensions) are called *unstructured* meshes. There are big differences in terms of what happens mathematically when you use them to solve a problem and how hard they are to generate, but for our purposes here, the issues are similar.

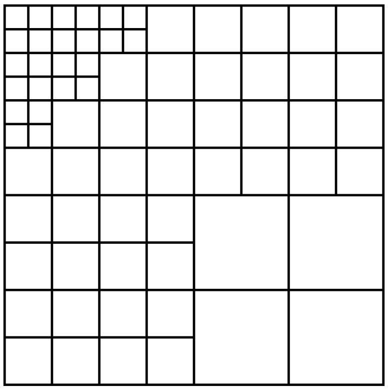
Some terminology: a typical mesh consists of three-dimensional *regions* or *volumes*, and their bounding *faces*, *edges*, and *vertices*. A data structure implementing the mesh will often allow queries such as "what faces bound this region" and "what edges are incident on this vertex" to be made efficiently.

The term "element" often is used to refer to the highest-dimension entity, and in many cases is the entity with which the solution is stored.

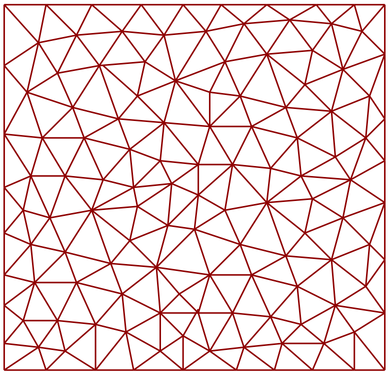
In fact, our Jacobi example is really just a computation on a uniform quadrilateral structured mesh. Here, the squares serve as the mesh elements and these contain the solution values.



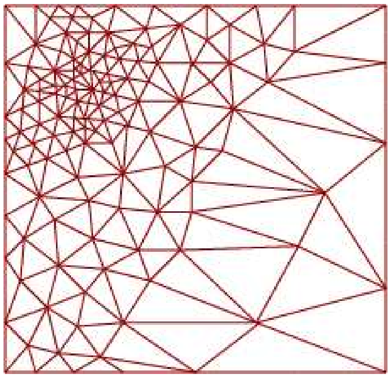
A (hypothetical) adaptive refinement of such a mesh might look like:



Here is a simple unstructured mesh:



The use of adaptivity (and some other considerations) necessitate linked data structures for the mesh. Here is the triangular mesh after a hypothetical refinement operation:



Such mesh structures may be stored in memory as arrays of entities or with a full topological hierarchy. We will soon consider an implementation of an adaptive structured mesh using a *quadtree* data structure.

These sample unstructured meshes are stored in memory using a hierarchical linked structure. These structures are implemented in a software library called the *SCOREC Mesh Database (MDB)*. We could write programs to operate entirely on these meshes (and many have been written) but it is not yet parallel.

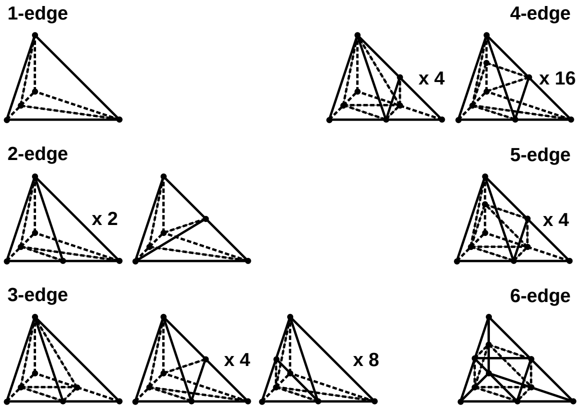
A second library, the *Parallel Mesh Database (PMDB)* is built on top of MDB and provides for distributed meshes.

We would like our distributed meshes to provide all the functionality and flexibility that is provided by the regular MDB. See the description and figures in the *Applied Numerical Mathematics* article. PMDB takes care of all of the message passing to migrate the mesh entities as appropriate and to update all of the partition boundary structures.

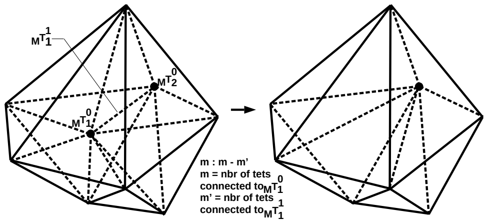
A variety of algorithms can be used to determine just how to distribute mesh elements among a set of cooperating processes and several of these *mesh partitioning* algorithms will be topics of study for us very soon.

Adaptive mesh refinement of PMDB meshes is provided by the (very scary) refdref library.

- Refinement of tetrahedra:



- Coarsening of tetrahedra:


${}_M T_1^1$
${}_M T_2^0$
${}_M T_1^0$
m : m - m'
m = nbr of tets connected to ${}_M T_1^0$
m' = nbr of tets connected to ${}_M T_1^1$

We can see some of this in action with some solution animations.

Computation on Quadtrees

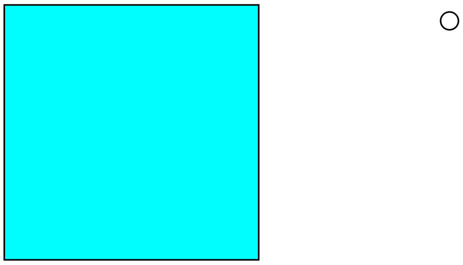
We will consider a relatively simple example of an adaptive computation. We return to our Jacobi Iteration heat solver.

We could approach this problem using an unstructured mesh as described above, but that can get complicated very quickly, even more so than the approach we will take. Instead, we will use a C program to solve Laplace's equation on a square domain using Jacobi iteration that operates on an adaptive *quadtree* structure. We will see how to parallelize this adaptive Jacobi solver, using the SPMD model with MPI for message passing and finally, how to implement a redistribution procedure to rebalance the load after imbalance is introduced by adaptivity.

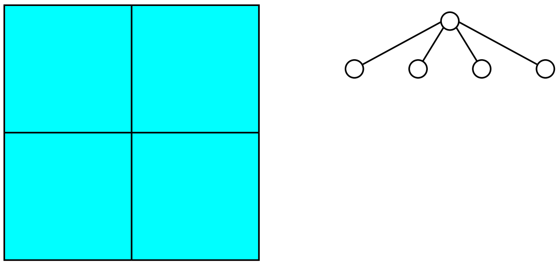
Quadtrees

A quadtree is a spatial tree data structure. Each node in the tree is called a *quadrant*. Leaf nodes are called *terminal quadrants*. These terminal quadrants will serve as the elements on which we will perform computation.

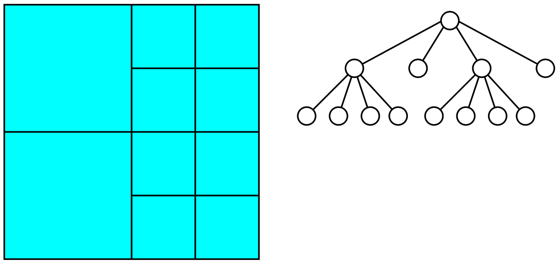
The tree's root represents the entire domain, which, in our case, is a square.

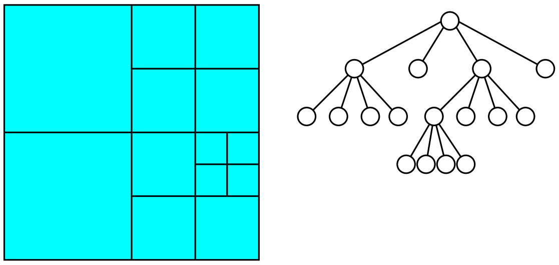


The four children of the root each represent one quarter of the space taken by the root.



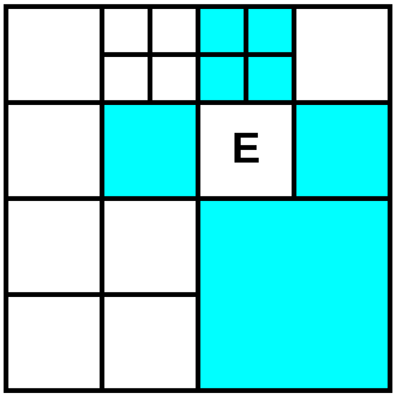
These children can then be divided in four, continuing down as many levels as desired. Different parts of the domain may be refined to different levels.





Computing Sequentially on a Quadtree

The Jacobi iteration on a given quadtree is similar to the iteration you've done in the array-based versions. However, since there may be a deeper quadtree in some parts of the domain, some leaf quadrants have neighbors more or less refined than themselves.



In the above figure, the element "E" should compute its value based on the values of the shaded neighbor quadrants. Note that E's north neighbor is actually refined one level further, so it has two immediate north neighbors. Ideally, the north neighbor value should be the average of the two immediate quadrant neighbors, but it is sufficient to use the value at the shaded quadrant (which is the average of the four leaf quadrants).

One obstacle to overcome is to locate neighboring quadrants efficiently. It is possible to maintain direct neighbor links within your quadtree structure, but you may find it more useful to search based on coordinate information. Each quadrant knows its own bounding box coordinates and from this can easily compute the coordinates of the adjacent quadrants at the same level. A nice feature of the quadtree structure is that the quadrants that contain any point in space can easily be found with a simple traversal from the root, determining the correct child at each step by comparing coordinates. If the neighbor point is outside the root quadrant, you know to apply a boundary condition.

Sequential Program Requirements

To make the program more interesting in the context of adaptivity, our program will allow a wider range of initial and boundary conditions than your previous implementations.

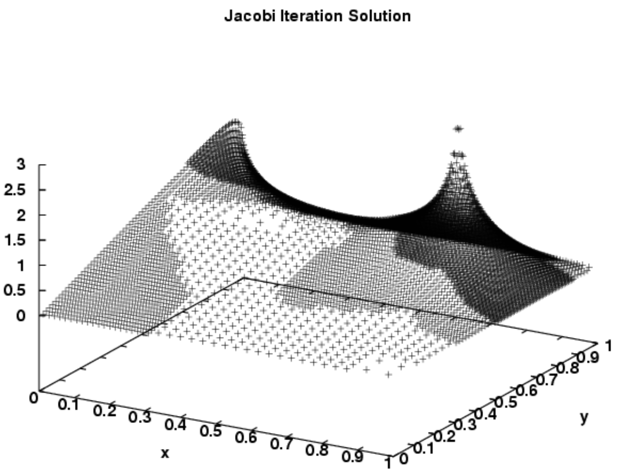
- Initial conditions are be specified by a C function that provides values given (x,y) coordinates of a point in the domain. Each leaf quadrant's solution value is initialized to the value obtained by passing the quadrant's centroid to this function.
- Boundary conditions are also specified through C functions. The left and right boundaries take the y coordinate as a parameter and the top and bottom boundaries take the x coordinate as a parameter, allowing boundary conditions to be functions in addition to simple constants. When a quadrant needs to query a boundary condition when computing its value, it should call the appropriate function.
- Special "internal boundary conditions" are specified by one more C function. The function takes a leaf quadrant and sets its value if the quadrant contains any points that have internal boundary values. For example, if there is a point heat source at (0.25,0.25) keeping the immediate area at a constant temperature of 2, this function will set the solution value of any quadrant containing (0.25,0.25) to 2. This function should be called on each leaf quadrant during each Jacobi iteration step. If the function sets the quadrant's solution value, that value should be used instead of the solution computed based on its neighbors.

These functions can be hard-coded into our program, but we could implement a system where initial and boundary conditions can be specified through a configuration file if time permits.

The program will take several parameters:

1. an initial global refinement level, which in turn determines the initial number of leaf quadrants
2. a Jacobi iteration tolerance, similar to the tolerance from the previous implementations
3. a limit on the number of Jacobi iteration steps
4. a refinement threshold, specifying the maximum difference in temperature allowed between adjacent leaf quadrants
5. a limit on the total number of refinement steps

Here is the result of the program when run on the unit square with the $x = 0$ boundary condition set to $2y$, the other side boundaries fixed at 0, and a special boundary condition setting the point at (0.75,0.75) to 3. The entire domain is initialized to 0. The initial quadtree is refined 3 levels (64 leaf quadrants), a Jacobi tolerance of 0.0001 with a maximum number of iterations of 2000, and a refinement threshold of 0.05 and a maximum number of refinement levels of 3.



Sequential Program Design and Implemenation

The structure of the program should look something like this:

```
create quadtree to initial refinement level
set initial solution values of leaf quadrants to initial conditions
do {
  do {
    foreach leaf quadrant {
      if (!special boundary condition applied)
        do Jacobi iteration step
    }
  } while (error > Jacobi tolerance && jac_steps < jac_max)
  refine quadrants based on refinement threshold
} while (any quadrants were refined && ref_steps < ref_max)
```

Even though our program is written in C instead of an object-oriented language like C++ or Java, we will follow good object-oriented design. One way to do this is to separate the data structures and functionality of the quadtree from the solution process.

We'll start by looking at the main program, then look at quadtree-related structures and functions as we encouter them.

Some things to notice at this point:

- The list of local variables in `main` is very short - just the pointer to the root of our quadtree and several solution parameters and miscellaneous variables like loop indices and file pointers.

- The program requires 6 command-line arguments. In addition to the solution parameters mentioned above, we have an "output level" that we can use to specify how frequently our program will print solution data.

- We create the initial quadtree structure: the parameters are the bounding box (top, left, bottom, right), the initial solution value, and the parent point (`NULL` for the root).

- Next, we have to do our initial global refinement.
  - We do this with a *visitor function* that will in turn call a *callback function* with each leaf quadrant as a parameter.
  - The function `visit_all_leaf_quadrants` is, as you might suspect, a recursive function. It takes as parameters the quadrant whose leaves are to be visited, a pointer to the function to call on each leaf, and a pointer to some caller-specified data that will also be passed along to the callback function. The function's two options are:
    - ∗ if we are already at a leaf, call the callback function
    - ∗ if we are an interior quadrant, make a recursive call to the visitor function on each child quadrant.
  - In this case, we use a callback function `do_refine`, which performs one level of refinement on each leaf quadrant.
  - Our `do_refine` function just calls a quadtree function `refine_leaf_quadrant`.
  - The `refine_leaf_quadrant` function:
    - ∗ Checks to make sure the quadrant is in fact a leaf. Note the use of the function `is_leaf_quadrant` to check and the use of the macro `ASSERT` to terminate our program with an error condition if this is not a leaf.
    - ∗ Creates 4 new leaf quadrants, each $\frac{1}{4}$ the size of the original leaf, and with the former leaf as their parent.
  - This happens as many times as we specified for the initial refinement level (`init_ref`), and results in a uniform quadtree with 4^{init_ref} leaves.

- Next, we need to set our initial conditions based on the function provided.
  - We again use our leaf quadrant visitor (leaves as the only quadrants that have a solution value in our implementation) this time with `set_init_cond` as the callback function.
  - The `set_init_cond` function calls our initial condition function `initial_cond`, with the coordinates of the centroid of the leaf to get the appropriate initial condition value for this leaf, then calls `quadrant_set_value` to assign it to the leaf.

- For the moment, we'll ignore the solution printing and look at the solution process.

- Our main loop is a nested `do/while`. The outer loop guides solutions on different refinements of the tree. The inner loop is the Jacobi iteration corresponding to what we had done previously. We'll consider that loop to start.
- The solution step is done with another visitor/callback.
  - Like you did in your implementations earlier, we always do two iterations in succession: one to compute a second solution from the current, then another to compute current again based on the second.

    The callback `do_jacobi_iter_phase1` computes what we call `previous` from `value`, and `do_jacobi_iter_phase1` computes `value` from `previous`.
  - Each of these follows the same general procedure:
    - Check to see if there is a special boundary condition that applies to this leaf. This is done with the `apply_other_bc` function. If a condition was applied, the function returns true and we're done with this leaf.
    - Otherwise, we need to find our 4 neighbor values that we'll be averaging to get our new value. This is not as simple as changing array indicies by one like we did when the computation was being done on a simple 2D array.
    - Neighbor-finding in a quadtree involves a simple search. In our quadtree code, it is done with the function `neighbor_quadrant`, which takes any quadrant and a direction and returns either a neighboring quadrant in the desired direction, or `NULL` if there is no neighbor (i.e., we are on a boundary).

      It is not even always clear what we mean by the "north neighbor" There is potentially a whole hierarchy of quadrants neighboring us in a given direction. What we need for our solution is the neighbor *at our own level* in the tree hierarchy. If our neighbor is not refined as far as us, we'll go ahead and use the leaf at a higher level. If it is refined more, we want to use the leaf quadrants adjacent to us, but only count them once, even if more than one is adjacent.

      We find the appropriate neighbor by finding a point in space that we know will be inside our neighbor in a given direction, then searching for that point in the tree.

      Something to think about: we could start this search at the root and work down, but can we do better by searching up the tree to find our nearest ancestor that contains the desired point, then back down to the appropriate leaf? This is faster when our neighbors are most likely our siblings or cousins. Only in those cases where our neighbor is in a different level 1 quadrant will we need to search all the way back to the root. This is the search we use in the `neighbor_quadrant` function.
    - Any neighbor search that returns `NULL` indicates that we've gone off the edge of the universe and should use a boundary condition instead. We do this with the `bc_*` functions.
    - With the 4 neighbor or boundary values to average ready, we compute the new value and store it in the quadrant.
    - In the phase 2 computation, we also check the error value, where the maximum encountered so far being passed in as `maxerr`. Note that this uses the extra

callback parameter to pass the pointer to `main`'s local variable `max_jac_diff` to the callbacks. At the end, we'll have the maximum error available in `max_jac_diff`.

  - After the 2 iterations, we check to see if we've reached our error tolerance or the maximum number of iterations.

We've ignored an important part here so far. The adaptivity. That's the outer loop.

- Recall that we want to find places where adjacent quadrants have solution values whose difference exceeds a given tolerance. When we find such situations, we want to refine the tree in that area to have a more accurate solution. That's what happening in the outer loop.
- The function `calc_error_and_refine` is the heart of the refinement functionality. It determines where refinement is needed, performs that refinement, then returns the number of refinements that were performed. (If there are 0, we can stop processing, since we'd only recompute the same solution we just computed on the previous grid.)
- The implementation of `calc_error_and_refine` again makes good use of our visitor function in each phase of our refinement procedure:
  - Marking quadrants for refinement: the `check_if_refinement_needed` function. We locate each of our neighbors, then see if any of them have solution values too far from our own. If one is found, we mark ourself for refinement.

    Marking for refinement involves a little trick in the quadtree data structure. Since we don't have children when we're a leaf, we use a non-zero value in the third child pointer to indicate the refinement mark.
  - Refining marked quadrants: the `refine_if_marked` function. If the leaf being visited is marked for refinement, we refine it. Notice that the new quadrants inherit the solution value from their parent. This means we use the last solution on one grid as the initial solution on the next.

Finally, a bit about printing the solution. As you learned even in the non-adaptive versions you wrote, it is nearly impossible to understand the solution based only on printing out values from the grid. This problem becomes even more difficuly once we include adaptivity. Our approach is to print solutions to a file in a format that includes the coordinates of each leaf quadrant and its solution value. These values may then be plotted with gnuplot. A script that I used to generate some of the solution plots in these notes is available as `solutionplot.gp`. This is a very simple gnuplot script and I am sure you can do better, but it gets the job done.

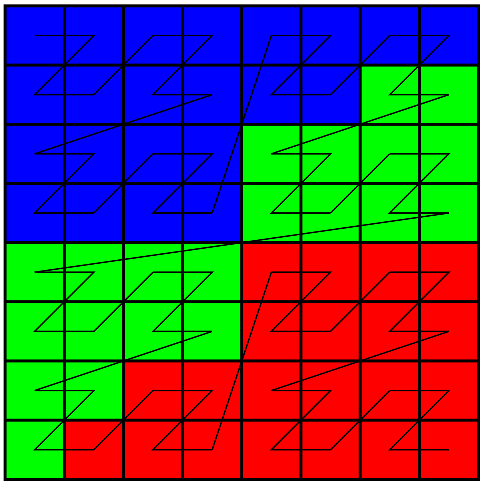
Parallelization

Our ultimate goal is to parallelize this program.

There are many choices to make when parallelizing a program such as this. Which parts of the computation are to be performed by each processor? What is the granularity of the units of work to be divided among the processors? What information must be maintained and what communication is needed to support the computation once the work has been divided? Will the workloads need to be adjusted following adaptive steps? How can such a rebalancing be performed?

We will consider parallelization using the following phased development process:

Phase 1 Build the initial quadtree to the requested refinement level, but replicate it completely on each process. Assign a unique owner process to each leaf quadrant, essentially partitioning the quadtree. Each quadrant's solution is computed only by its owner, but quadrants along partition boundaries will need to exchange solution values between iterations. There are many possible approaches to this partitioning problem, but it should be fairly straightforward if you divide the quadrants into partitions based on a tree traversal. If you have n leaf quadrants and p processes, assign the first $\frac{n}{p}$ to the first partition (process 0), the next $\frac{n}{p}$ to the next partition (process 1), and so on, handling remainder quadrants in some reasonable way. If child quadrants are ordered NW-NE-SW-SE, a partitioning of a base quadtree refined to three levels might be done as follows:



For this phase, we need to implement the message passing needed to send solution information from owned quadrants on partition boundaries to those other processes (and only to those other processes) that will need the information during the solution process. Note that we are ignoring adaptivity for this phase, so the working program should compute only on the initial quadtree. Solution output procedures are needed to compare the solution from the parallel version with those from the sequential version.

Phase 2 Now, to reintroduce adaptivity. Adaptivity will work much as it did in the sequential program, except that when a quadrant is refined, it need only be refined on the owner process. This means that the quadtree, which was originally completely replicated on each process, will grow beyond the initial refinement level only on the owner process of a given quadrant. This should be easy on the interiors of partitions, but will complicate both the error checking (neighbors may now be off-process) and the solution process (off-process copies of initial leaf quadrants may have been refined). The first can be addressed by a similar solution-value exchange as you needed for phase 1 computation. The second may be addressed by doing at least partial refinements of the off-process copies of quadrants involved in interprocess communication.

Phase 3 All of this adaptivity will likely introduce a load imbalance. If all or most of the refinement takes place in just a few processes, those processes will have a larger workload during each Jacobi iteration, causing other processes to wait before the boundary exchange. After a refinement phase, we will implement a rebalancing phase, where the partitions of the initial quadtree structure can be adjusted (and refined parts of the tree migrated appropriately) to ensure that each process has approximately the same number of owned leaf quadrants. To keep this relatively straightforward, the units of work that are allowed to be migrated are the original leaf quadrants. This means that the same techniques you used to keep track of off-process neighbor quadrants in phase 1 can be used here. The downside is that the granularity of the "work objects" that you are partitioning can get large after several refinement steps have occurred, meaning a perfect load balance may not be possible. I suggest using the same rule for deciding which parts of the quadtree are assigned to which processes. Traverse the tree (only to the level of the original quadrants) and fill the partitions. If you broadcast the sizes of each quadrant (that is, the number of leaf quadrants below it), each process can compute this new decomposition independently and determine which parts of its tree need to be sent elsewhere.

Partitioning and Dynamic Load Balancing

We have considered partitioning and dynamic load balancing in some specific situations. Let's now think about it in more general circumstances.

Our assumption here is that we have a computation whose memory and computational requirements are dominated by some set of objects that we distribute among a set of cooperating processors. We will most often think of this as a mesh being used to solve a PDE, but other structures are possible.

Typically, one process is assigned to each processor. Data are distributed among the processes, and each process computes the solution on its local data (its *subdomain*). Inter-process communication provides data that are needed by a process but "owned" by a different process. This model introduces complications including

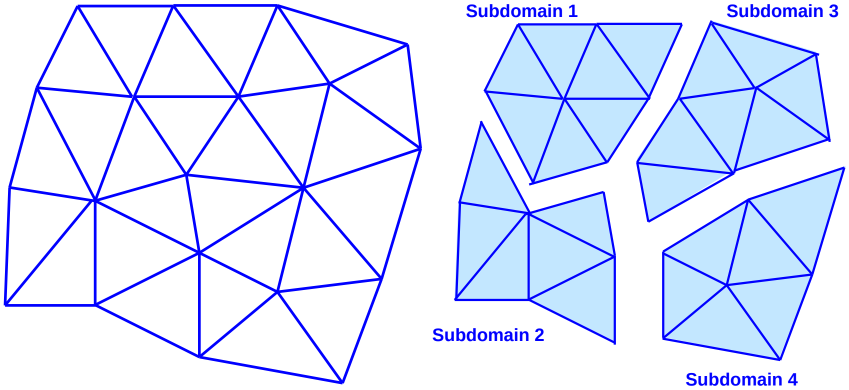
1. assigning data to subdomains (i.e., *partitioning*, or when the data is already distributed, *dynamic load balancing*)
2. constructing and maintaining distributed data structures that allow for efficient data migration and access to data assigned to other processes, and
3. communicating the data as needed during the solution process.

The Partitioning Problem

The computational work of PDE simulation is often associated with certain “objects” in the computation. For particle simulations, computation is associated with the individual particles; adjusting the distribution of particles among processors changes the processor load balance. For mesh-based applications, work is associated with the entities of the mesh — elements, surfaces, nodes — and decompositions can be computed with respect to any of these entities or to a combination of entities (e.g., nodes and elements).

The partitioning problem, then, is the division of objects into groups or subdomains that are assigned to cooperating processes in a parallel computation.

At its simplest, a partitioning algorithm attempts to assign equal numbers of objects to partitions while minimizing communication costs between partitions. A partition’s subdomain, then, consists of the data uniquely assigned to the partition; the union of subdomains is equal to the entire problem domain. For example, the following figure shows a two-dimensional mesh whose elements are divided into four subdomains.



Often communication between partitions consists of exchanges of solution data for adjacent objects that are assigned to different partitions. For example, in finite element simulations, “ghost elements” representing element data needed by but not assigned to a subdomain are updated via communication with neighboring subdomains.

Objects may have weights proportional to the computational costs of the objects. These nonuniform costs may result from, e.g., variances in computation time due to different physics being solved on different objects, more degrees of freedom per element in adaptive p-refinement, or more small time steps taken on smaller elements to enforce timestep contraints in local mesh-refinement methods. Similarly, nonuniform communication costs may be modeled by assigning weights to connections between objects. Partitioning then has the goal of assigning equal total object weight to each subdomain while minimizing the weighted communication cost.

Additionally, we may wish to form partitions of varying sizes to account for nonuniform computational capabilities of the processors to which the partitions are assigned.

Dynamic Repartitioning and Load Balancing Problem

Workloads in dynamic computations evolve in time, so a partitioning approach that works well for a static problem or for a slowly-changing problem may not be efficient in a highly dynamic computation. For example, in finite element methods with adaptive mesh refinement, process workloads can vary dramatically as elements are added and/or removed from the mesh. Dynamic repartitioning of mesh data, often called *dynamic load balancing*, becomes necessary.

Dynamic repartitioning is also needed to maintain geometric locality in applications like crash simulations and particle methods. In crash simulations, for example, high parallel efficiency is obtained when subdomains are constructed of geometrically close elements. Similarly, in particle methods, particles are influenced by physically near particles more than by distant ones; assigning particles to processes based on their geometric proximity to other particles reduces the amount of communication needed to compute particle interactions.

Dynamic load balancing has the same goals as partitioning, but with the additional constraints that procedures

1. must operate in parallel on already distributed data,
2. must execute quickly, as dynamic load balancing may be performed frequently, and
3. should be incremental (i.e., small changes in workloads produce only small changes in the decomposition) as the cost of redistribution of mesh data is often the most significant part of a dynamic load-balancing step.

While a more expensive procedure may produce a higher-quality result, it is sometimes better to use a faster procedure to obtain a lower-quality decomposition, if the workloads are likely to change again after a short time.

Partition Quality Assessment

The goal of partitioning is to minimize time to solution for the corresponding PDE solver. A number of statistics may be computed about a decomposition that can indicate its suitability for use in an application.

The most obvious measure of partition quality is computational load balance. Assigning the same amount of work to each processor is necessary to avoid idle time on some processors. The most accurate way to measure imbalance is by instrumenting software to determine processor idle times. However, imbalance is often reported with respect to the number of objects assigned to each subdomain (or the sum of object weights, in the case of non-uniform object computation costs).

Computational load balance alone does not ensure efficient parallel computation. Communication costs must also be considered. This task often corresponds to minimizing the number of objects sharing data across subdomain boundaries, since the number of adjacencies on the bounding surface of each subdomain approximates the amount of local data that must be communicated to perform a computation. For example, in element decompositions of mesh-based applications, this communication cost is often approximated by the number of element faces on boundaries between two or more subdomains. (In graph partitioning, this metric is referred to as “edge cuts”.) A similar metric is a subdomain’s *surface index*, the percentage of all element faces within a subdomain that lie on the subdomain boundary. Two variations on the surface index can be used to estimate the cost of interprocess communication. The *maximum local surface index* is the largest surface index over all subdomains, and the *global surface index* measures the percentage of all element faces that are on subdomain boundaries [**?**]. In three dimensions, the surface indices can be thought of as surface-to-volume ratios if the concepts of surface and volume are expanded beyond conventional notions; i.e., the “volume” is the whole of a subdomain, and the elements on subdomain boundaries are considered the “surface.” The global surface index approximates the total communication volume, while the maximum local surface index approximates the maximum communication needed by any one subdomain.