CHAPTER 5
Partitioning and Load Balancing

The most common approach to parallelizing adaptive finite element computations involves a domain decomposition which assigns portions of the domain to cooperating processes in a parallel computation. Mesh partitioning and dynamic load balancing algorithms are used to achieve this domain decomposition, and their study remains an active area of research. Adaptive computations evolve in time, so an approach which works well for a static problem or for a slowly-changing problem may not be efficient in a highly-adaptive computation. The presence of parallel environments with heterogeneous processing units (in terms of processor speed, cache size, main memory size) and heterogeneous or hierarchical network structures complicates the partitioning procedure further.

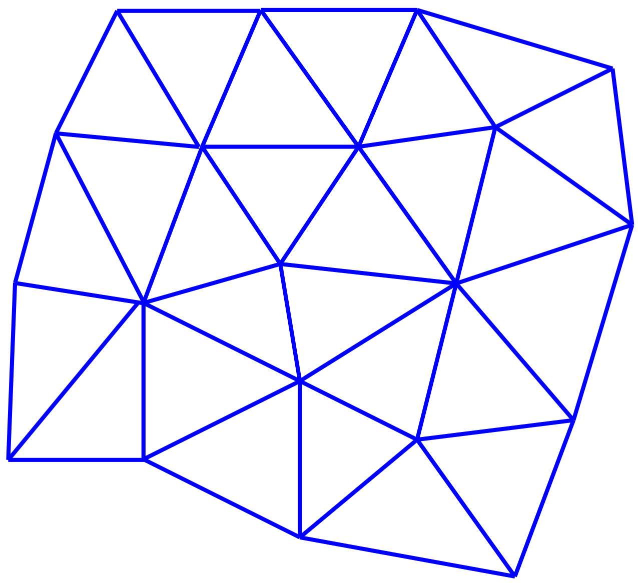

Figure 5.1: Sample two-dimensional mesh to be partitioned.

The mesh partitioning problem is equivalent to partitioning a graph $G = (V, E)$, where mesh elements serve as the graph vertices (V) and connections between mesh elements are the graph edges (E). Figure 5.1 shows a sample two-dimensional mesh, and Figure 5.2 shows the same mesh with such an induced graph. A graph

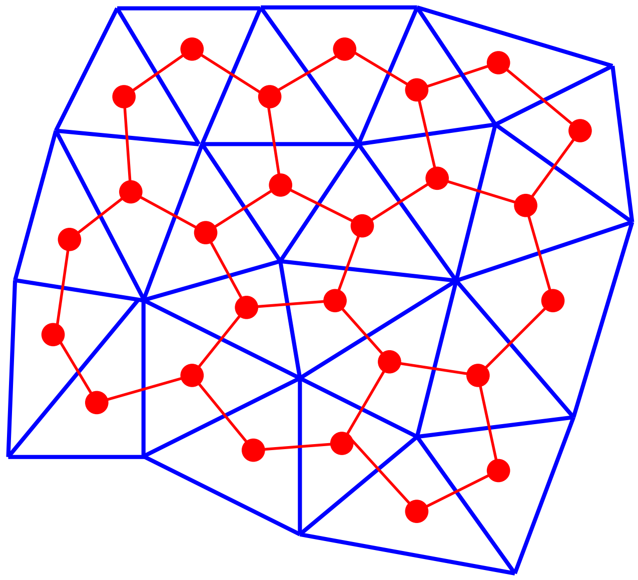

Figure 5.2: Sample two-dimensional mesh from Figure 5.1 with graph induced.

vertex is created for each mesh face and a graph edge is inserted connecting each pair of mesh faces which share a common edge. In three dimensions, mesh regions correspond to graph vertices and graph edges are inserted for each pair of mesh regions which share a common mesh face.

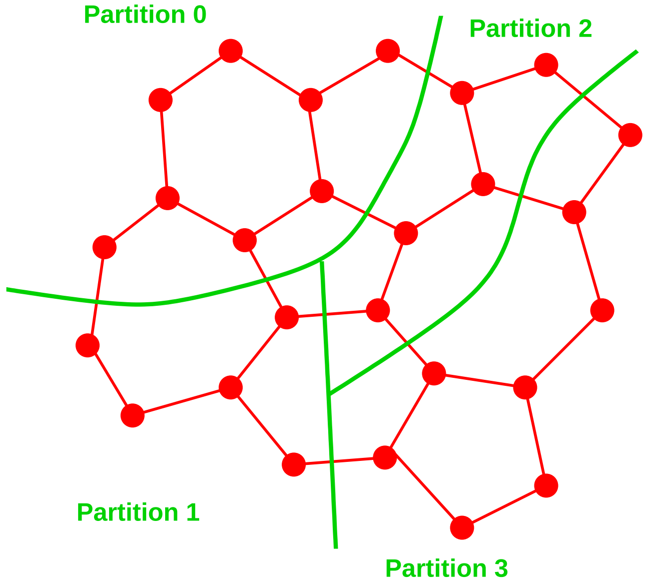

Figure 5.3: Four way partitioning of the graph induced by the mesh from Figure 5.1.

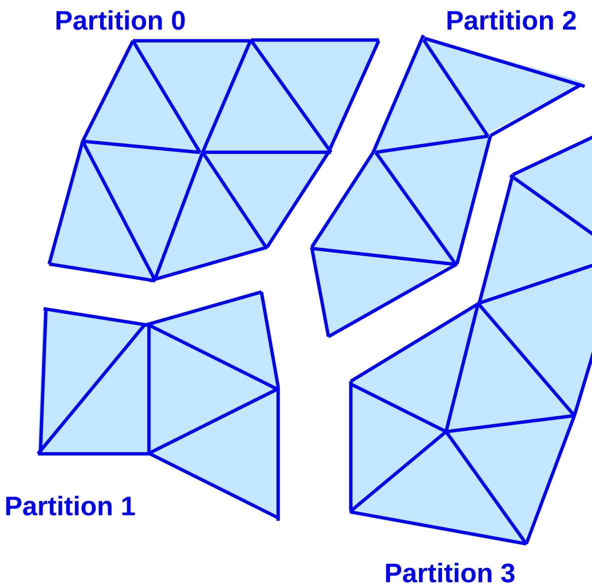

Figure 5.4: Partitioning of the of the mesh from Figure 5.1 when the graph partitioning of Figure 5.3 is applied.

A k-way partitioning of the graph G is obtained by dividing the vertices into subsets $V_1, ..., V_k$ where $V = V_1 \cup ... \cup V_k$ and $V_i \cap V_j = \oslash$ for $i \neq j$. Figure 5.3 shows one possible partitioning of the graph induced by the mesh in Figure 5.1. Here, partition 0 contains 8 vertices, partition 1 contains 5 vertices, partition 2 contains 5 vertices and partition 3 contains 8 vertices. Graph partitioning algorithms produce these partitions with the goals of placing equal numbers of vertices in each V_i while minimizing the number of edges "cut" by the partitions. An edge e_i is cut when one of its vertices belongs to one partition and the other belongs to a different one. In Figure 5.3, a total of 9 edges are cut. Vertices represent units of work to be balanced among the processors, while edges represent the communication needed. Fewer edge cuts in the partitioning generally means less communication overhead during the finite element solution. Algorithms to provide an optimal partitioning are NP complete, so heuristic algorithms are generally used. The graph partitioning is related to the mesh partitioning problem by assigning each partition V_i to a processing unit. Figure 5.4 shows the partitioning of the mesh from Figure 5.1 when divided based upon the graph partitioning of Figure 5.3.

In this chapter, we first discuss partition quality and present some metrics which may be used to quantify the quality of partitions (Section 5.1). Next, we present an overview of current approaches to the partitioning and load balancing problem (Section 5.2). We then present our enhancements to an iterative load balancing algorithm (Section 5.3) and describe our system to attach weights to elements for use by load balancers (Section 5.5). This weighting scheme has been used in the context of predictive load balancing methods [125] and to help account for load imbalances due to an adaptive time-stepping method, enforcement of boundary conditions, heterogeneous algorithms, and adaptive p-refinement. Work related to heterogeneous and hierarchical systems is discussed later in this dissertation (Section 5.10).

5.1 Measuring Partition Quality

The manner in which mesh data is distributed among the cooperating processes in a parallel computation has a profound impact on the performance of analysis software [18]. One measure of partition quality is the percentage of elements which require access to off-process data during the computation. On a distributed-memory parallel computer, poor partition quality results in a higher communication cost during the finite element solution phase.

A balance of computational load is necessary to avoid idle processes, but this alone does not ensure efficient parallel computation. The primary goal of most mesh-partitioning algorithms is minimizing the number of cuts that the subdomains create when the partitioning is viewed as a communication graph [59, 60, 109]. With finite volume and finite element schemes this closely corresponds to the task of minimizing the number of element faces on interprocess boundaries, since the number of faces on the bounding surface of each partition is proportional to the amount of local data that must be communicated to perform a computation.

We appraise the cost of interprocess communication by computing two *surface indices* of partitions [18]. The *maximum local surface index*, r_{M}, measures the maximum percentage of mesh faces on the boundary of any partition, and the *global surface index*, r_{G}, measures the percentage of all faces which are on interpartition boundaries. These surface indices can be thought of as surface-to-volume ratios if the concepts of surface and volume are expanded beyond conventional notions.

Given a partitioning consisting of a set of n partitions $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, let b_i denote the number of faces on the partition boundary of P_i and f_i represent the number of faces in the partition P_i. If the total number of faces on the partition boundaries is given by b_t and the total number of faces on all the partitions is denoted by f_t, the surface index metrics may be defined as

$$r_\mathrm{M} = \max_{i=1,\ldots,n} \left(\frac{b_i}{f_i} \right), \; r_\mathrm{G} = \frac{b_\mathrm{t}}{f_\mathrm{t}} \; .$$

Another measure of partition quality is the geometric connectivity of the submeshes in each partition. Having multiple disjoint connected components within a process' submesh, which has also been referred to as *subdomain splitting* [62], is undesirable. The interprocess boundary is not optimal when the submesh is not a single geometrically connected component, and domain decomposition methods for the solution of the linear systems will converge slowly for partitions with this property [31]. If a relatively small disjoint part of one subdomain can be merged into a neighboring partition, the interprocess boundary will decrease, thereby improving the surface index metrics.

In a three-dimensional mesh, the definition of a "connected partition" depends on the dimension of the entity which must be shared by two regions for them to be considered "connected." We consider all three types of connectivity: *face connectivity*, *edge connectivity*, and *vertex connectivity*. As the names suggest, two regions are face-connected if they share a common face, edge-connected if they share a common edge, and vertex-connected if they share a common vertex. Face-connected partitions are the most desirable, as two regions which share a face also will share the edges of that face, and the vertices of those edges. A higher degree of face connectivity means less interprocessor communication during the computation phase, since communication is often across face boundaries.

For each partition P_i, we compute its face, edge, or vertex connectivity as the number of disjoint face-, edge-, or vertex-connected components within the partition. The average connectivity is computed as the average of these local connectivities over all partitions.

With current technology, interconnection network latency is a significant component of communication cost; therefore, interprocess connectivity (the number of processes with which each process must exchange information during the solution phase) is as significant a factor in performance [18] as the number of boundary faces.

Heterogeneous and hierarchical computer systems also present challenges for partitioning and load balancing. If processor speeds, memory sizes, and communication speeds vary, a distribution which does not take this information into account will not be optimal.

5.2 Available Partitionings

Mesh partitioning has been an active field of research for over a decade. Here, we survey some of the important methods. A common class of algorithms are based on a geometric or spectral decomposition of the mesh. *Orthogonal Recursive Bisection* (ORB) [13], also called *recursive coordinate bisection*, uses the coordinates of element centroids to partition the mesh. At each recursive step, the Cartesian coordinate of the longest dimension of the domain under consideration is bisected, elements are sorted according to the bisecting coordinate, and half of the elements are assigned to each subdomain. *Inertial Recursive Bisection* (IRB) [43], proceeds similarly, but in a direction orthogonal to a domain's principal axis of inertia. *Parallel Sort Inertial Recursive Bisection* (PSIRB) [107] performs IRB using a parallel sort to allow full parallel execution. *Recursive Spectral Bisection* (RSB) [59, 94] is considered to be among the best static mesh partitioning procedures. RSB uses eigenvectors of the Laplacian matrix associated with a graph to sort graph vertices for bisection, and provides better quality partitions than the geometric methods. It is generally too expensive for use in a large-scale three-dimensional adaptive computation. *Multilevel Recursive Spectral Bisection* (MRSB) [7] has improved the efficiency of RSB, but its parallelization relies heavily on a shared-memory architecture and is unlikely to be efficient in a true message passing environment [6]. Other enhancements to RSB [110, 115, 118] may make it more useful as a dynamic repartitioner, but serious doubts remain. The extension of the multilevel graph-theoretic algorithms in Metis [68] to operate in parallel (ParMetis [69]) make them worthy of consideration in this context. The Metis multilevel approach involves the creation of a series of coarsened graphs. The coarsest graph is partitioned using a spectral partitioning. The partitioning is improved incrementally at each level as the graph is refined back to its original size.

In an adaptive computation, the initial partitioning may not remain efficient, necessitating dynamic redistribution throughout the computation. As such, applying global partitioning strategies after each adaptive step can be costly relative to solution time. Thus, a number of iterative dynamic load balancing techniques that incrementally migrate data from heavily to lightly loaded processes have been developed [14, 17, 32, 34, 73, 117, 120]. An iterative method based on load imbalance trees called *Iterative Tree Balancing* (ITB) [91, 107] is implemented within our system. ITB achieves balance through incremental mesh migration, which occurs between processes that control neighboring spatial regions. More details of ITB and our enhancements to the algorithm are described in Section 5.3. The Jostle package [119] provides an iterative partition optimization process which may also be useful for balancing adaptive computation. ParMetis includes algorithms for incremental improvements to existing partitionings [99].

While methods such as ITB provide inexpensive ways to achieve a balanced computation, their repeated use can lead to degradation in partition quality. Octree decomposition is a successful strategy for generating three-dimensional unstructured meshes [108]. We also use an octree-based partitioning (OCTPART) [34, 53, 84] which takes advantage of an underlying tree structure to achieve balance and to maintain reasonable communication costs. Since such trees are easily constructed from arbitrary meshes, the procedure is independent of octree mesh generation. Partitioning may be done serially or in parallel. In either case, it is inexpensive; hence, it may be used with adaptive procedures. Partitioning requires tree traversals that (i) calculate the processing costs of subtrees and (ii) form the partitions. The use of space-filling curves [92] produces similar results, keeping neighboring elements of the ordering in close spatial proximity.

A somewhat different approach is taken by Mitchell's full-domain partition (FuDoP) [86]. The usual procedure when partitioning a finite element mesh is that each process receives a partition covering a disjoint portion of the problem domain, possibly with a small overlap area near partition boundaries. With FuDoP, each process contains a mesh which covers the entire domain, with most of the mesh on each process remaining very coarse. Each process contains a fine mesh on only a small part of the domain, and those refined areas correspond to traditional partitions. Essentially, the extra computation is done to avoid communication. A similar idea has been applied to parallel solution procedures by Bank and Holst to avoid communication during the solution phase [5].

Maerten *et al.* [80] have produced the *Dynamic Re-Allocation of Meshes for parallel finite element Applications* (DRAMA) package, which builds cost functions for use with existing partitioners such as ParMetis [69] and Jostle [119]. Zoltan [35, 36] provides a common interface to allow straightforward selection of partitioning algorithms, and includes implementations of an Octree partitioner [53] among others.

5.3 Improved ITB Partitioning

The iterative tree balancing algorithm suffers from deficiencies which have the potential to degrade the quality of partitions after repeated use. These problems surface as very jagged partition boundaries, often leading to disconnected portions of the mesh being assigned to a process, which is one indication of a poor-quality partitioning. We developed software to detect such cases, and make enhancements that attempt to combat these problems.

5.3.1 ITB Algorithm

ITB follows Leiss and Reddy [73], Wheat [120], and Devine and Flaherty [34] in that lightly loaded processes request load from their most heavily loaded neighbors. However, instead of considering an immediate neighborhood of processes, the algorithm views the requests as forming a forest of trees (Figure 5.5b). Each tree is then linearized, and a logarithmic-time scan operation is used to compute load flows [91] to determine the amount of data to be migrated (Figure 5.5c). Layers of elements on interprocess boundaries are moved from heavily loaded to lightly loaded processes to achieve balance within each tree (Figure 5.6) [91]. ITB is “diffusive,” and a heavily loaded process will distribute load to several lightly loaded neighbors.

ITB may be iterated to achieve a global balance within a specified tolerance or be set to terminate after a fixed number of iterations. With low per-iteration costs, ITB can be executed for a few iterations between substages of operations like mesh enrichment [17, 107] without the large time penalty of a global repartitioning. A more complete discussion of the ITB algorithm may be found in Özturan [91].

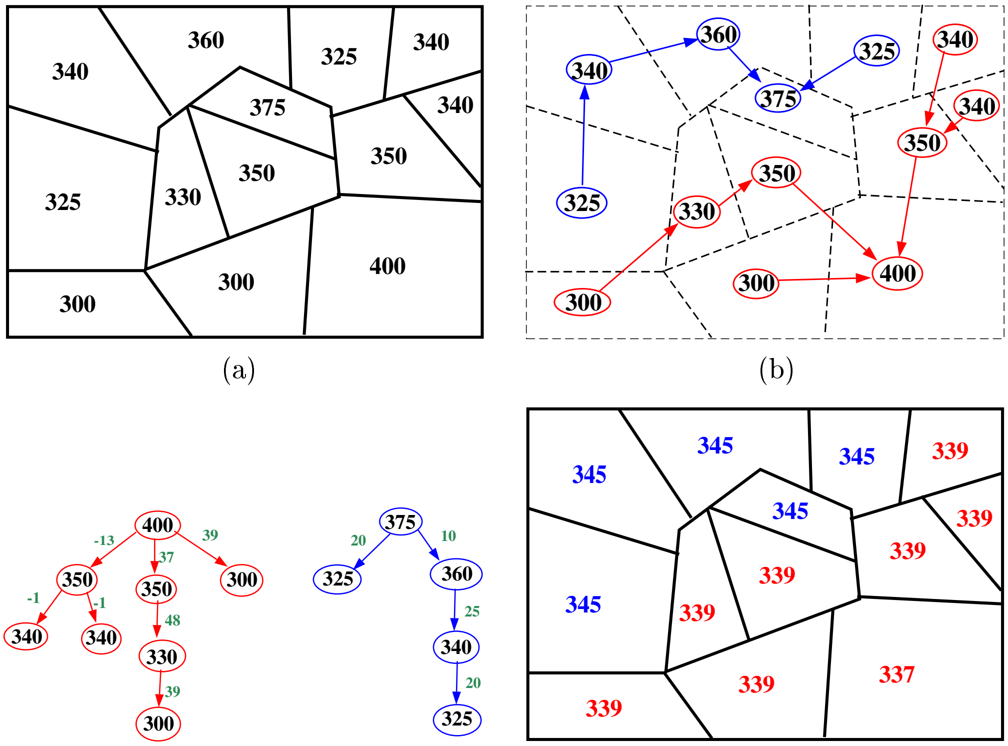

Figure 5.5: An iteration of ITB: (a) original unbalanced loads, (b) load requests, (c) forest of trees induced by load requests, and (d) loads after one iteration.

5.3.2 Geometric Selection Methods

Our first enhancement to ITB concerns the selection of elements for migration. In its original form [91, 107], ITB repeatedly selected "layers" of elements on a partition boundary for migration (Figure 5.6). This selection process has the potential to create disconnected partitions when the partition boundary is near a part of the mesh which is more refined than its neighbors (Figure 5.7). Even in cases where the partitions remain connected, repeated application of the slide-by-slice selection

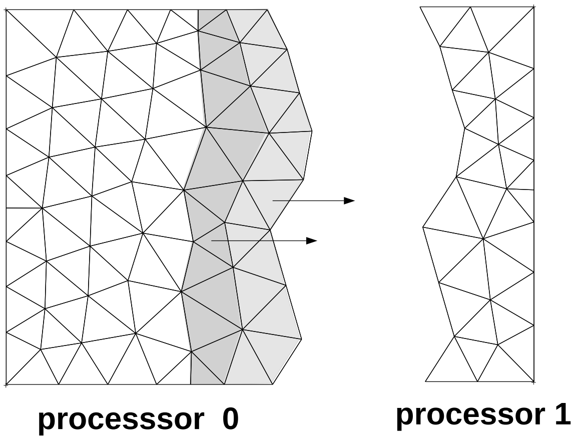

Figure 5.6: "Slice-by-slice" selection method for ITB. Layers of mesh entities are selected from the interpartition interface for migration from process 0 to process 1. Figure from Shephard, *et al.* [107].

scheme leads to undesirable jagged partition boundaries.

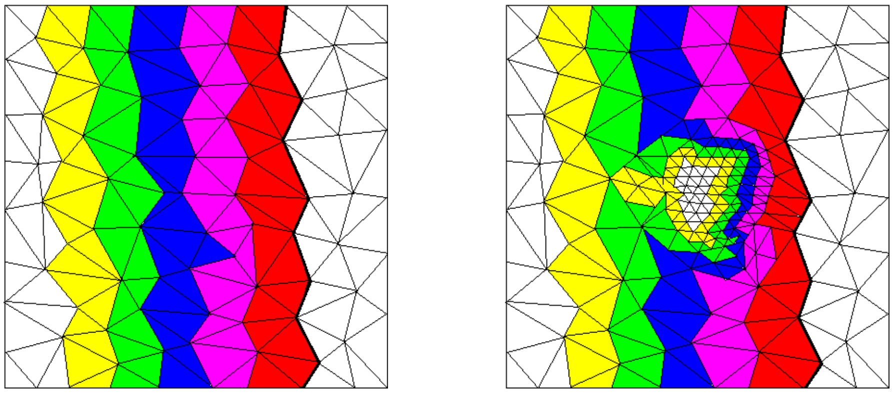

Figure 5.7: Migration of elements from right to left using slice-by-slice selection with all elements nearly the same size (left). Disconnected partition resulting when the same selection is applied following refinement (right).

An improved selection for migration is illustrated in Figure 5.8. Ideally, the boundary between any two partitions would approximate a plane orthogonal to the line through the centroids of the two partitions, intersecting that line at the midpoint of the two centroids. Here, we describe heuristics which help approximate that partitioning.

A run-time option was added to ITB which allows an arbitrary priority to be assigned to elements as they are chosen for migration. Instead of selecting elements

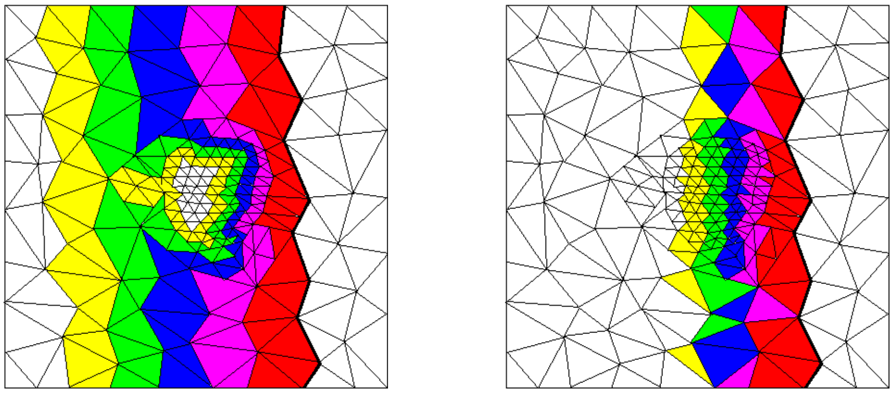

Figure 5.8: Disconnected partition resulting when the slice-by-slice selection is applied following refinement (left). Desired element selection to avoid disconnection and to preserve smooth partition boundaries (right).

for migration based solely on the slice-by-slice traversal, a priority queue of candidate elements is maintained. When an element is selected for migration from the head of the priority queue, any new elements which become candidates for migration (elements adjacent to the one chosen which will become adjacent to the receiving partition following the migration) are assigned a priority and are inserted into the queue.

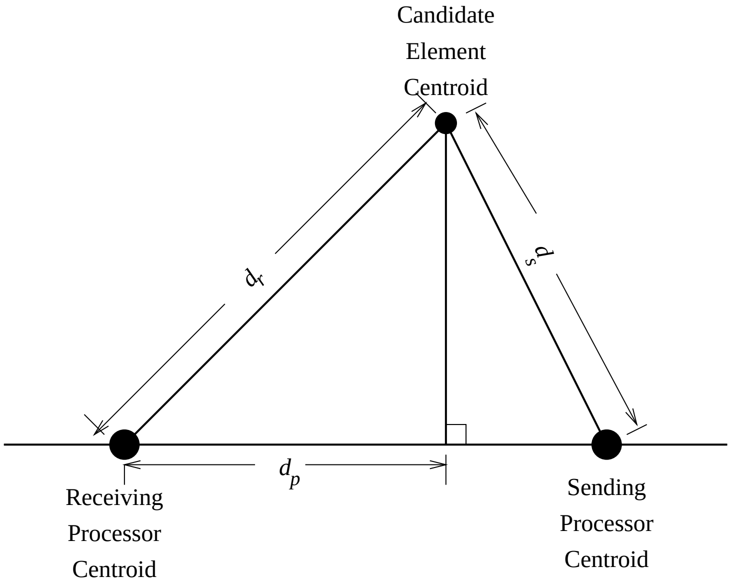

Figure 5.9: Geometric priority calculation for ITB priority-based selection. For direct geometric selection, the priority is computed as $d_s - d_r$. For geometric projection selection, the priority is computed as $-d_p$.

Geometric heuristics were used to assign these priorities in an attempt to prevent the problem demonstrated in Figure 5.7, but results were mixed. Two different priority calculations based on centroids of candidate elements and the partition centroids of the sending and receiving partitions were implemented (Figure 5.9). The direct geometric priority calculates the distance from the candidate element centroid to the partition centroids of the sender (d_s) and receiver (d_r) and assigns the priority as the difference $d_s - d_r$. The direct geometric method did not behave as expected in some circumstances, leading to partition boundaries which were more curved than desired. This led to development of the geometric projection selection, which calculates the projection of the candidate element centroid onto the line passing through the sending and receiving partitions' centroids. The distance from the receiving partition centroid to this projection point is called d_p, and $-d_p$ is used as the priority.

Statistic	Unbalanced	Original ITB	Direct Geom.	Geom. Proj.
Max Load	12701	8156	8157	8155
Balancing Time	0.00	35.5	186.9	221.9
GSI	5.66	7.26	6.63	7.05
Avg Conn (vert)	1.38	1.94	1.19	1.38

Table 5.1: Comparison of original ITB and ITB using geometric heuristics on 16 processors of an IBM SP2 computer. "Unbalanced" represents the partitioning before rebalancing, "Original ITB" is the partitioning following original ITB, "Direct Geom." and "Geom. Proj." are the partitioning following ITB with direct geometric and geometric projection priority, respectively. "Max Load" is the maximum number of regions in any partition. "Balancing Time" is repartitioning time in seconds. "GSI" is the global surface index, and "Avg Conn (vert)" is the average vertex connectivity of the partitions.

As shown in Table 5.1, this method did produce improved partitions, but the additional cost incurred when assigning priorities and managing the priority scheme frequently outweighed the benefits of the improved partition quality. This table shows partitioning statistics for a mesh used in a simulation of the flow about an Onera M6 wing. The initial mesh from this simulation is shown in Figure 5.10. The mesh has been refined adaptively, and contains 130,454 regions and is distributed