Dynamic Load Balancing for Parallel Finite Element Methods with Adaptive h- and p-Refinement*

Karen D. Devine† and Joseph E. Flaherty§

Abstract

We describe a dynamic load-balancing strategy for parallel finite element methods with adaptive mesh (h-) and order (p-) refinement. The load-balancing algorithm is based on the tiling load-balancing system, where global balance in achieved by performing local balancing within overlapping neighborhoods of processors. Tiling is applied to each mesh level created by the adaptive h-refinement. Weights are used in the migration routines to reflect the nonuniform elemental work loads caused by adaptive p-refinement. The combination of adaptive refinement and tiling significantly reduces total execution time relative to fixed-mesh, fixed-order methods yielding comparable accuracy, as we demonstrate with experiments on an nCUBE/2.

1. Introduction

Adaptive numerical methods, such as mesh refinement and order enrichment, provide greater efficiency than traditional numerical methods by concentrating computational effort in regions of the problem domain where the solution is difficult to obtain. In adaptive methods, a solution and an estimate of its error are computed on a base mesh. The mesh is "improved" by an adaptive technique in regions where the error estimate is high, and the solution is recomputed on the improved mesh. This process is repeated until the solution satisfies a specified error criterion. In this work, we use combinations of two adaptive techniques: mesh refinement and order enrichment. With mesh (h-) refinement, additional small elements are placed in high-error regions to obtain greater resolution of the solution, while large elements are used in regions where the solution is accurate. With order (p-) enrichment, high-order approximations are used in high-error regions while low-order approximations are used elsewhere.

On parallel computers, adaptive methods suffer from load imbalance. Adaptive p-refinement creates nonuniform elemental work loads, and adaptive h-refinement causes nonuniform numbers of elements per processor. Moreover, the number of elements and per-element work loads can vary as the computation proceeds, necessitating a dynamic load-balancing strategy. The strategy must be fast, since it is executed frequently. It must be distributed across the processor array and use little memory, so that the application can scale with the number of processors. It should also be incremental, since localized refinement does not require global rebalancing.

We have applied a dynamic load-balancing strategy called tiling [10, 11] to an adaptive finite element method for solving hyperbolic conservation laws. Tiling achieves global balance by performing local load balancing within overlapping neighborhoods of processors. The algorithm is modified to account for the nonuniform elemental work loads of adaptive p-refinement and multiple meshes created by adaptive h-refinement. We present results on a 1024-processor nCUBE/2 that demonstrate the tiling algorithm's effectiveness.

2. Parallel Adaptive Methods

We have developed adaptive p-, h-, and hp-refinement strategies for the solution of hyperbolic conservation laws [7]. The adaptive methods are based on the local finite element method of Cockburn et al. [4, 5]. Spatially, the solution is approximated by a piecewise Legendre polynomial that is continuous on an element but may have discontinuities at interelement boundaries. A projection limiter that restricts solution moments is applied to high-order approximations to remove spurious oscillations near discontinuities [3, 7]. For two-dimensional problems, the spatial discretization yields a system of $(p+1)^2$ ordinary differential equations (ODEs) per element,

* This work was supported by Sandia National Laboratories under Research Agreement AD-9585.

† Parallel Computing Sciences Department, Sandia National Laboratories, Albuquerque, NM 87185.

§ Department of Computer Science and Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, NY 12180.

where p is the degree of the polynomial approximation. We integrate this system in time with high-order, explicit Runge-Kutta methods.

The implementation of the non-adaptive method exhibits high scaled parallel efficiency on MIMD computers [3, 7]. The physical domain is divided equally among the processors. The processors compute each time step synchronously, with communication occurring between each Runge-Kutta stage. Information is needed from only the elements sharing edges with an element to compute its solution. This compact stencil necessitates only nearest-neighbor communication and additional storage for one row of "ghost" elements around processor subdomains.

The adaptive hp-refinement strategy attains improved computational efficiency by using high-order approximations in regions where the solution is smooth and refined meshes near discontinuities. The solution and an *a posteriori* estimate of its error [2, 3, 7] are computed for one time step on a coarse base mesh. The degrees of the approximation are increased for high-error elements in smooth regions, and the solution is recomputed on the enriched elements. High-error elements near discontinuities are then subdivided. Refinement must be performed in both space and time to enforce the CFL condition, requiring more complicated data structures than the adaptive p-refinement [1]. The fine elements are maintained in separate meshes, creating a tree of meshes with the base mesh as the root and the finer meshes as the children. The fine elements are initialized with data projected from the coarse mesh. The solution on the fine mesh levels is obtained recursively, with more, smaller time steps taken on each refined mesh, and interpolated to the coarse mesh. Because of the synchronization of the Runge-Kutta method, all processors must know when the solution is to be recalculated on high-error elements. A global exchange at the end of a time step communicates whether p- and/or h-refinement was performed in the problem domain, allowing the processors to determine whether to backtrack for p-refinement, compute on the next mesh level for h-refinement, or accept the current time step on the current mesh level.

3. Tiling Algorithm

We have extended the tiling load-balancing system of Wheat [10, 11] to dynamically balance processor loads in adaptive methods. Global balance is achieved by performing local balancing within overlapping processor neighborhoods. A *processor neighborhood* consists of a center processor and all other processors for which the center has ghost cells. Every processor is the center of one neighborhood and may belong to many neighborhoods. Processors within a neighborhood are balanced with respect to each other using local performance measurements.

In adaptive h- and hp-refinement, each mesh level must be load balanced to prevent processors without refinement from being idle while processors with refinement compute. Thus, we apply the tiling strategy to each mesh level, adding data structures and communication routines to account for the migration of child elements away from their parents' processors.

In every balancing phase, each processor determines its load as the time to process its local data since the previous balancing phase less communication time during the computation phase. Neighborhood average loads are calculated. Each processor requests work from the processor in its neighborhood with the greatest load. Processors then prioritize received work requests based on the request size, and select elements to export to the requesting processors. Highest export priority is given to the elements with the most neighbors in the importing processor. Elements are weighted with their elemental work loads, and elements are selected for migration until the work load of the highest-priority elements would cause the work request to be over-satisfied or the processor's work load to fall below its neighborhood average. The exporting processor sends the migrating elements' identification numbers (IDs) to the importing processor and to processors sharing edges with the migrating elements so they can update pointers to the migrating elements. The importing processors allocate memory for the incoming elements, and the elements are transferred.

4. Data Structures and Communication

Data structures for each mesh level maintain element connectivity and data position information, as shown in Figure 1. Local elements are elements on which the processor performs the application's computation. They are stored in a height-balanced, binary tree (AVL tree) to allow

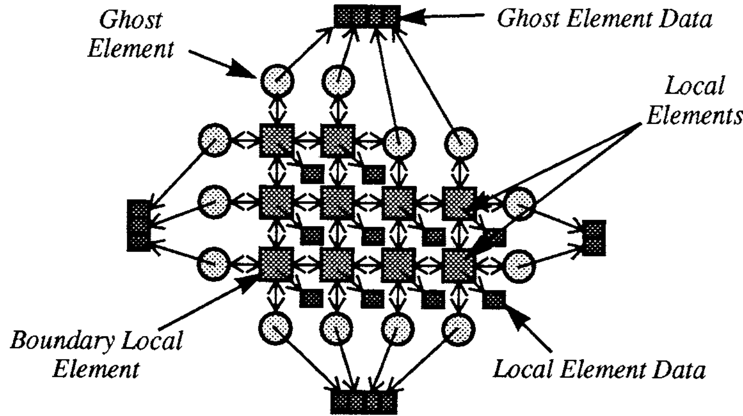

Figure 1. Element interconnection diagram [10].

efficient insertion and deletion during migration. During the computation phase, local elements are accessed via in-order traversal of this tree. Each tiling element contains pointers to its four neighboring elements, its parent element, its child elements, and its application data.

Since the ghost elements needed can change when elements are migrated, they are also stored in an AVL tree. Ghost-element data are stored contiguously, so a processor can receive the data in a single message from each neighbor and read the messages directly into the ghost element data space. Boundary local elements are also maintained in AVL trees, one for each neighboring processor, to facilitate proper ordering of the boundary element data during gather operations needed to send boundary data to neighboring processors.

Elements along the edge of a refined mesh have neighboring elements in the coarse mesh. The fine mesh level maintains an AVL tree to store data for coarse neighbors local to other processors. The coarse mesh level also tracks which of its elements have fine-mesh neighbors on other processors so it can communicate the values to the fine mesh's processors at the end of the coarse mesh time step. These elements are maintained in trees analogous to the boundary element trees within a mesh level.

Resetting neighbor pointers after mesh refinement is complicated by allowing child elements to migrate away from their parents. The child and parent pointers and element IDs stored in the local elements are used to reset the pointers. These local element values must be communicated along processor subdomain boundaries.

When parent and child elements are in different processors, communication between the fine and the coarse mesh levels is needed to allow interpolation of the fine mesh values to the coarse mesh elements and to enable coarse elements to tell their non-local child elements to coalesce when the level of local refinement is decreasing. Trees are constructed in both the coarse and fine levels to mark which elements will receive and send data. This communication is generally more expensive than the boundary exchanges and the coarse neighbor communications since many more elements' values must be communicated, but it is performed less frequently.

5. Results

Example 5.1. We solve

$$u_t + 2u_x + 2u_y = 0,\ t > 0,\ (x, y) \in (0, 1) \times (0, 1), \tag{1a}$$

with initial and boundary conditions specified such that

$$u(x, y, t) = \frac{1}{2}(1 - \tanh(20x - 10y - 20t + 5)),\ t \geq 0, \tag{1b}$$

using the local finite element method with adaptive p-refinement and tiling on 16 processors of the nCUBE/2. The initial decomposition of the domain is uniform. In Figure 2 (left), we show the decomposition after 10 load-balancing steps. Processors with high-order, high work-load elements have far fewer elements in their subdomains than processors with low-order elements.

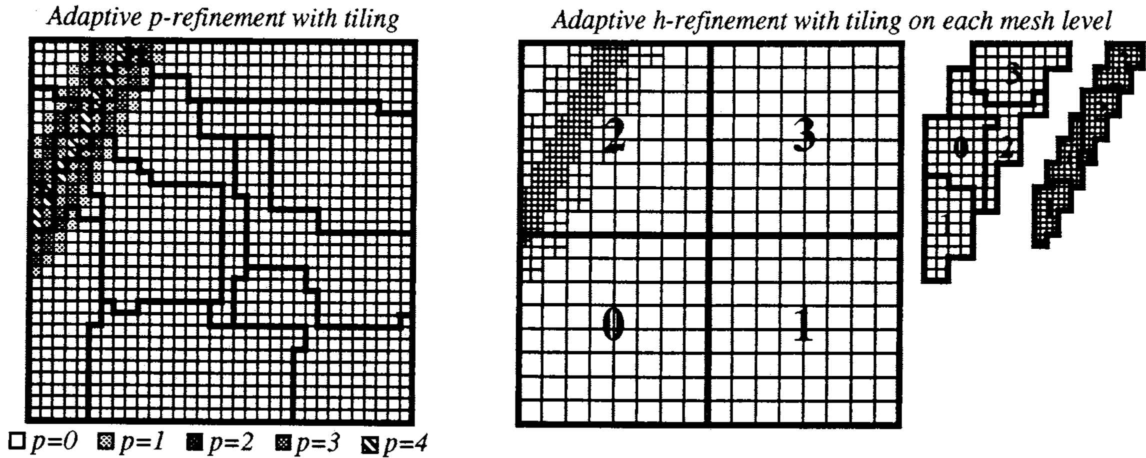

Figure 2. Example of the decomposition generated by the tiling algorithm for an adaptive p-refinement mesh on 16 processors (left) and an adaptive h-refinement mesh on four processors.

We solve (1) again using adaptive h-refinement and tiling on four processors of an nCUBE/2 hypercube. The initial decomposition is uniform and, thus, only processors 0 and 2 have refinement in their subdomains. We apply the tiling algorithm once each time step on each mesh level, and show the decomposition on each level after 10 base-mesh time steps in Figure 2 (right). The decomposition on the uniform base mesh is unchanged. However, the elements on both of the refined mesh levels have been redistributed to allocate work to processors 1 and 3.

Example 5.2. We solve (1) on $(x, y) \in (0, 32) \times (-15.5, 16.5)$ for 225 time steps using a 512×256-element mesh and adaptive p-refinement with local error tolerance 10^{-5} on all 1024 processors of an nCUBE/2. In Table 1, we show the total execution time and the total maximum communication, computation and load-balancing times, where total maximum time is defined by

$$TotMax(time) = \sum_{i=1}^{\#\text{ of time steps}} \max_{\text{all processors } P} (time \text{ used by processor } P \text{ in time step } i). \qquad (2)$$

Without tiling, the adaptive p-refinement computation is highly unbalanced, as indicated by an average to maximum work ratio of 0.359. With tiling performed once each time step, the total maximum communication time is increased 65.9% over the non-balanced case, due to the irregular subdomains created by the tiling algorithm. However, despite the additional communication and load-balancing time, the total execution time of the adaptive method is reduced 47.8% using tiling.

In Table 1, we also show the performance of the fixed-order method with $p = 2$ on a 512×256-element mesh. The computation is well-balanced and achieves 82.7% scaled parallel efficiency. However, using adaptive p-refinement and tiling reduces the total execution time 67.7% relative to the fixed-order computation yielding roughly twice as much error. Applying tiling to the fixed-order computation reduces its total execution time to 1501.90 seconds, but the adaptive method with tiling still requires 65.5% less total execution time.

Table 1. Performance comparison for Example 5.2.

	Adaptive p-Refinement Method		Fixed-Order Method
	No Balancing	With Tiling	No Balancing
Global L^1-Error	0.0384807	0.0384807	0.0640644
TotMax(Computation Time)	928.38 secs.	372.21 secs.	1549.77 secs.
Avg. / Max. Work Ratio	0.359	0.896	0.855
TotMax(Communication Time)	61.45 secs.	101.95 secs.	60.91 secs.
TotMax(Balancing Time)	0.0 secs.	24.51 secs.	0.0 secs.
Total Execution Time	991.29 secs.	517.72 secs.	1601.96 secs.

Example 5.3. We solve (1a) on $(x, y) \in (0, 1) \times (0, 1)$ with initial and boundary conditions specified so that

$$u(x, y, t) = \frac{1}{2}(1 - \tanh(100x - 10y - 180t + 5)), t \geq 0, \quad (3)$$

with $p = 1$ and three levels of adaptive h-refinement with local error tolerance 4.5×10^{-5} on 256 processors of an nCUBE/2 without balancing and with balancing once each local time step. The total maximum communication time (2) is increased 23.2% with balancing, due to the communication between coarse and fine mesh elements that have been migrated to different processors (see Table 2). The total maximum balancing time is larger than the balancing time for p-refinement, since many more balancing phases are used as each mesh level is load balanced. Despite the tiling overhead, however, we see an 82.2% improvement in the total execution time of the h-refinement method.

In Table 2, we also show the performance of the fixed-mesh method on a 640×640-element mesh with $p = 1$. The computation is well balanced and achieves 99.2% scaled parallel efficiency. However, to obtain a solution with comparable accuracy, the fixed-mesh method requires more than eight times as much execution time as the adaptive h-refinement method with tiling.

Table 2. Performance comparison for Example 5.3.

	Adaptive h-Refinement Method		Fixed-Mesh Method
	No Balancing	With Tiling	No Balancing
Global L^1-Error	0.146914	0.146914	0.151939
TotMax(Computation Time)	5566.26 secs.	856.45 secs.	11,596.36 secs.
Avg. / Max. Work Ratio	0.081	0.467	0.999
TotMax(Communication Time)	29.01 secs.	57.33 secs.	91.17 secs.
TotMax(Balancing Time)	0.0 secs.	65.30 secs.	0.0 secs.
Total Execution Time	7830.80	1391.53	11,684.94 secs.

Example 5.4. We solve

$$u_t + (\frac{1}{2}u)_x + (\frac{1}{2}u)_y = 0, -1 < x, y < 1, t > 0, \quad (4a)$$

with

$$u(x, y, 0) = \frac{1}{2} + \frac{1}{2}\sin\pi(x + y), -1 \leq x, y \leq 1, \quad (4b)$$

and periodic boundary conditions on a 16×16-element base mesh with $p = 1$ initially using adaptive hp-refinement with local error tolerance 5.0×10^{-6} on 256 processors of the nCUBE/2. In Table 3, we show the performance statistics of the method without balancing and with balancing once each local time step. The total maximum communication time (2) is increased 43.9% with balancing. The total maximum balancing overhead is 40.99 seconds. Despite the tiling overhead and additional communication time, however, the total execution time of the adaptive method is reduced 55.8% using tiling.

In Table 3, we also show the performance of the fixed-mesh, fixed-order method on a 112×112-element mesh with $p = 2$. The non-adaptive method attains nearly-perfect load balance, but its total execution time to compute a solution with comparable accuracy is more than 4.5 times greater than that using the adaptive hp-method and tiling.

6. Conclusion

We have demonstrated the effectiveness of tiling for load-balancing finite element computations with adaptive p-, h-, and hp-refinement. Global balance is achieved by performing local balancing within overlapping neighborhoods of processors. When elements are selected for migration, they are weighted with their individual work loads to adjust for the nonuniform elemental loads caused by p-refinement. The tiling strategy is applied to each mesh level created

Table 3. Performance comparison for Example 5.4.

	Adaptive *hp*-Refinement Method		Fixed-Mesh, Fixed-Order Method
	No Balancing	With Tiling	No Balancing
Global L^1-Error	0.0220026	0.0220026	0.0218864
TotMax(Computation Time)	2474.40 secs.	585.80 secs.	5291.71 secs.
Avg. / Max. Work Ratio	0.208	0.878	0.994
TotMax(Communication Time)	319.49 secs.	459.73 secs.	583.65 secs.
TotMax(Balancing Time)	0.0 secs.	40.99 secs.	0.0 secs.
Total Execution Time	2909.50 secs.	1285.78 secs.	5858.89 secs.

by h-refinement. The combination of adaptive refinement and tiling significantly reduces total execution times relative to fixed-mesh, fixed-order methods yielding comparable accuracy.

The implementation of the tiling algorithm in three dimensions is straight-forward. The current strategies for local balancing and selecting elements for migration could be used. However, the irregular subdomain boundaries created by the tiling algorithm may cause an unacceptable amount of additional communication in three-dimensions. Investigations currently underway [6] try to optimize the shape of the subdomains to control the volume of interprocessor communication for both two- and three-dimensional problems.

The extension of the tiling system to unstructured meshes would require an interface that specifies the geometry and mesh to be used in the problem. Following Hammond [8], a static load-balancing strategy [9] could be applied to the initial mesh before it is distributed to the processors, reducing the initial load imbalance and the effort required by the tiling algorithm.

Finally, porting the adaptive methods and dynamic load-balancing strategies to other parallel architectures is an easy process. A logical mesh of processors is assumed, simplifying the implementation of the methods on mesh architectures such as the Intel Paragon.

References

[1] M.J. Berger and J. Oliger. "Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations." *Jrnl. Comp. Phys.*, **53** (1984), 484-512.

[2] K.S. Bey. *An hp-Adaptive Discontinuous Galerkin Method for Hyperbolic Conservation Laws.* Ph. D. Dissertation. The University of Texas at Austin, 1994.

[3] R. Biswas, K. Devine and J. Flaherty. "Parallel, Adaptive Finite Element Methods for Conservation Laws." *Appl. Num. Math.*, **14** (1994) 255-283.

[4] B. Cockburn, S. Hou, and C.-W. Shu. "The Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws IV: The Multidimensional Case." *Math. Comp.*, **54** (1990), 545-581.

[5] B. Cockburn, and C.-W. Shu. "TVB Runge-Kutta Local Projection Discontinuous Galerkin Finite Element Method for Conservation Laws II: General Framework." *Math. Comp.*, **52** (1989), 411-435.

[6] H.L deCougny, K.D. Devine, J.E. Flaherty, R.M. Loy, C. Ozturan, and M.S. Shephard. "Load Balancing for the Parallel Adaptive Solution of Partial Differential Equations." RPI Tech. Rep. No. 94-8, Department of Computer Science, Rensselaer Polytechnic Institute, 1994.

[7] K. Devine. *An Adaptive hp-Finite Element Method with Dynamic Load Balancing for the Solution of Hyperbolic Conservation Laws on Massively Parallel Computers.* Ph.D. Dissertation, Rensselaer Polytechnic Institute, 1994.

[8] S. Hammond. *Mapping Unstructured Grid Computations to Massively Parallel Computers.* Ph. D. Dissertation, Rensselaer Polytechnic Institute, 1992.

[9] B. Hendrickson and R. Leland. "The Chaco User's Guide, Version 1.0." Sandia National Laboratories Tech. Rep. SAND93-2339, 1993.

[10] S. Wheat. *A Fine Grained Data Migration Approach to Application Load Balancing on MP MIMD Machines.* Ph.D. Dissertation, University of New Mexico, 1992.

[11] S. Wheat, K. Devine, and A. Maccabe. "Experiments with Automatic, Dynamic Load Balancing and Adaptive Finite Element Computation." *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, 1994, 463-472.