Load Balancing and Communication Optimization for Parallel Adaptive Finite Element Methods

J. E. Flaherty, R. M. Loy, P. C. Scully, M. S. Shephard,
B. K. Szymanski, J. D. Teresco, and L. H. Ziantz

Computer Science Department and
Scientific Computation Research Center
Rensselaer Polytechnic Institute
Troy, NY, U.S.A.

Abstract *This paper describes predictive load balancing schemes designed for use with parallel adaptive finite element methods. We provide an overview of data structures suitable for distributed storage of finite element mesh data as well as software designed for mesh adaptation and load balancing. During the course of a parallel computation, processor load imbalances are introduced at adaptive enrichment steps. The predictive load balancing methods described here use* a priori *estimates of work load for adaptive refinement and subsequent computation to improve enrichment efficiency and reduce total balancing time. An analysis code developed with these components for solving compressible flow problems is used to obtain predictive load balancing results on an IBM SP2 computer. Our test problem involves the transient solution of the three-dimensional Euler equations of compressible flow inside a perforated shock tube. We also present a message passing library extension in development which allows for automated packing of messages to improve communication efficiency.*

1. Introduction

Partial differential equations (PDEs) arise in many areas of scientific interest. The finite element method (FEM) is a standard analysis tool for solving systems of PDEs. In order to provide greater reliability, robustness, and efficiency in time and space, the discretization may be changed periodically during the solution process. These adaptive methods include spatial refinement or coarsening (h-refinement), method order variation (p-refinement), and/or moving the mesh to follow evolving phenomena (r-refinement). Each technique concentrates or dilutes the computational effort in areas needing more or less resolution.

Three-dimensional problems are computationally demanding, making parallel computation essential to obtain solutions for large problems in a reasonable time. However, parallelism introduces complications such as the need to balance processor loading, to coordinate interprocessor communication, and to manage distributed data. Often, the parallelization of FEM programs is done using a static partitioning of the mesh across the cooperating processors. However, a good initial partition is not sufficient to assure high performance throughout an adaptive computation. A dynamic repartitioning is needed to correct for load imbalance introduced by adaptive enrichment. In Section 2, we briefly describe reusable tools developed by the Scientific Computation Research Center (SCOREC) at Rensselaer that facilitate the development and use of parallel adaptive finite element software.

These tools have been used in many applications, including in the construction of a parallel finite element code described in [10] which can solve three-dimensional conservation laws. Parallel mesh enrichment routines are used for spatial refinement and coarsening (h-refinement) [18]. Refinement is also performed in time using a spatially dependent local refinement method [10]. The consideration of heterogeneous element weights allows us to balance processor loads based on the temporal as well as spatial refinement.

Typically, balancing follows mesh refinement and coarsening. The ability to predict and correct for imbalance prior to refinement can improve performance during the refinement stage while maintaining computational load balance during the successive solution phase similar to that achieved by an *a posteriori* balancing. Strategies for this are described in Section 4. This work focuses on balancing the spatial enrichment process itself in addition to balancing the subsequent computation.

The capabilities of the parallel adaptive system are demonstrated by the solution of compressible transient flow problems on an IBM SP2 computer. The solution of a transient flow in a perforated shock tube is shown in Section 5 along with results illustrating the advantages of our predictive balancing methods.

Interprocessor communication is implemented in our software using the *Message Passing Interface* (MPI). Its wide availability and portability make it an attractive choice; however, the communication requirements of parallel adaptive computation are sometimes inconvenient or inefficient when implemented using the primitives provided by MPI. Section 6 describes a set of library functions currently under development that are layered on top of MPI and accomplish these operations more easily and efficiently. We concentrate here on issues of dynamic buffer allocation and automatic message packing.

Finally, in Section 7, we discuss results and present future research directions.

2. SCOREC tools

To allow the author of an analysis code to concentrate on issues specific to a given problem rather than the details of the underlying data structures or parallelization issues, a number of tools have been developed. They provide a uniform way to implement reusable parallel adaptive software.

2.1. Mesh data structures

The *SCOREC Mesh Database* (MDB) [2] provides an object-oriented representation of a finite element mesh and operators to query and update the mesh data structure. The mesh entity hierarchy consists of three-dimensional *regions*, and their bounding *faces*, *edges*, and *vertices*, with bidirectional links between mesh entities of consecutive dimensional order. Tetrahedral regions are used as finite elements in three-dimensional meshes, while triangular faces serve as elements in the two-dimensional case. Initial meshes are created from a CAD model of the problem domain using the *SCOREC Finite Octree Automatic Mesh Generator* [19]. In order to maintain appropriate domain geometry as the mesh is generated, and later as it is enriched, mesh entities are explicitly classified relative to a geometric model of the domain.

A *Parallel Mesh Database* (PMDB) [9, 18] is built on top of MDB. Using PMDB, each processor holds MDB data associated with a subset of the complete mesh, and operators are provided to create and manipulate distributed mesh data. A partition boundary data structure is maintained to account for entities along partition boundaries that are shared by more than one processor. Fast boundary query and update operations are provided. PMDB allows arbitrary multiple migration of elements between processors to maintain a balanced computation. Although any element may be marked for migration, it is frequently boundary elements which are moved. An owner-updates rule is used to coordinate the update of partition boundary structures during migration.

2.2. Dynamic load balancers

Since a parallel adaptive computation generally involves large partitioned meshes, any dynamic load balancing scheme must operate on distributed mesh data. Recursive spectral bisection (RSB) [16] and, more recently, MeTiS [12] are generally regarded as good static partitioners. Multilevel Recursive Spectral Bisection (MRSB) has improved the efficiency of RSB but relies heavily on a shared-memory architecture and is likely to be inefficient in a true message passing environment [1]. Other enhancements to RSB [20, 21, 22] may make it more useful as a dynamic repartitioner, but doubts remain. MeTiS has recently been extended to operate in parallel (ParMeTiS [17]) and may be a viable option.

Several dynamic load balancing techniques are available for use with our software. *Iterative Tree Balancing* [9, 18] (ITB) is a diffusive algorithm based on repeated local migration. Processors request load from their most heavily loaded neighbors, and these requests form a forest of trees. Load flows are computed on linearized representations of these trees using a logarithmic-time scan. Elements on interprocessor boundaries are moved from heavily loaded to lightly loaded processors to achieve balance within each tree. This process is iterated to achieve a global balance within the required tolerance. *Parallel Sort Inertial Recursive Bisection* [18] (PSIRB) is an implementation of inertial bisection. The domain is bisected repeatedly in a direction orthogonal to its principal axis of inertia. Inertial coordinates are sorted in parallel to allow operation on a distributed mesh. *Octree Partitioning* [10] (OCT) uses a one-dimensional ordering of the nodes of an octree structure underlying the mesh. The ordered list of nodes is divided into segments corresponding to nearly equal load. Octants adjacent to one another in the ordered list tend to be spatially adjacent and, thus, form a good partition. The use of space-filling curves [15] produces similar results, keeping neighboring elements of the ordering in close spatial proximity. A recent addition to our software is an interface to convert our data structures into a format usable by the ParMeTiS package.

With h-refinement, the cost function that reflects the computational load on each processor is often chosen to be the number of elements on a processor. However, heterogeneous costs (weighted elements) are necessary when (i) using p-refinement or spatially-dependent solution methods, (ii) using spatially-dependent time steps (Section 3), (iii) enforcing boundary conditions, or (iv) using predictive load balancing (Section 4). PMDB includes an element weighting scheme that can be used to address each of these needs. ITB, PSIRB, and OCT each honor such weights when determining partitions.

3. Adaptive techniques

Results presented here were obtained using our Parallel Adaptive Euler Solver in which three-dimensional conservation laws are discretized using a discontinuous Galerkin finite element method [3, 5, 6]. More details may be found in [10]. The software makes use of both spatial and temporal refinement to concentrate computational effort in areas of the problem domain where it is most needed.

3.1. Spatial refinement

The SCOREC *mesh enrichment* procedure [18] is used to perform spatial (h-) refinement and coarsening in parallel. Error indicator information and threshold values, provided by an application code, are used to mark mesh edges to be coarsened, refined, or to remain unchanged. Stages of the enrichment process follow the order of (i) coarsening, (ii) optimization, (iii) refinement, (iv) optimization, (v) refinement vertex snapping, and (vi) optimization, with each mesh optimization stage being optional. Coarsening is done by an edge-collapsing process. Mesh optimization improves the quality of triangulations with respect to a given criterion ($e.g.$, element shape). Refinement is performed using subdivision patterns. After faces on partition boundaries with marked edges have been triangulated using two-dimensional refinement templates, each processor independently applies three-dimensional patterns which account for all possible configurations of marked edges (Figure 1). An over-refinement option may be used to reduce element shape degradation at the expense of creating more elements. A global shape control parameter can prevent any step of the enrichment process from creating elements with shapes worse than a specified value of an element shape quality measure. In the refinement vertex snapping stage, any newly created vertex classified as belonging to a curved model boundary must be "snapped" to the appropriate model entity to ensure mesh validity with respect to the geometry of the problem domain. With a small per-iteration cost, ITB is often executed for a few iterations between stages of mesh enrichment to improve balance without a large time penalty.

3.2. Temporal refinement

In a time-dependent calculation, such as the one described in Section 5, elements may choose spatially-dependent time steps based upon the "Courant stability condition" for explicit time integration. When this temporal *Local Refinement Method* (LRM) [10] advances the global solution time, a small number of larger time steps will be taken on large elements, while smaller elements, with smaller stable time steps, will require a larger number of time steps. Periodic synchronizations are used to calculate error estimates or indicators and to perform h-refinement. The sychnronization "goal time" is typically a small multiple of the smallest time step on any element of the mesh. The simpler *Method of Lines* (MOL) approach, where each element is advanced only by the smallest stable time step of any element in the mesh, is far less efficient than the LRM.

The time step for an element Ω_j is determined from the Courant condition as

$$\Delta t_j = \alpha \frac{r_j}{v_j}, \quad \alpha \leq 1, \tag{1}$$

where r_j is the radius of Ω_j's inscribed sphere and v_j is the maximum signal speed on Ω_j. For the Euler equations, v_j is the sum of the fluid's speed and the speed of sound. The parameter α is introduced to maintain stability in areas of mesh gradation. A value of $\alpha = 0.65$ is chosen empirically, but a more thorough analysis of the selection of this value is necessary.

While LRM greatly improves the efficiency of the computation, it complicates load balancing as work contributions vary from element to element. A size-based weighting scheme is used to account for this. Each element Ω_j is assigned weight w_j

$$w_j = \frac{1}{r_j} \tag{2}$$

where r_j is the radius of Ω_j's inscribed sphere. Smaller elements are thus given larger weights to account for the additional time steps they will take during the computation phase. Balancing in this situation is achieved by distributing weight evenly among the processors.

4. Predictive load balancing

Generally, load balancing follows the entire enrichment process, although a few ITB iterations may be performed between stages of the enrichment. However, balancing the work done during the refinement stage is not necessarily the same as balancing the computation phase. Thus, using a cost function designed to maintain computational balance may not be appropriate for ensuring an efficient enrichment process. Additionally, by anticipating the results of the refinement stage in conjunction with a user-supplied weighting function, one can achieve a reasonable computational load balance while moving a coarser (smaller) structure between processors as compared to migration after refinement.

4.1. Uniform workloads

Error indicator data generated during the computation phase may be used to select element weights and to perform load balancing before the refinement stage of the enrichment process. A similar technique has also been used by Oliker, Biswas, and Strawn [14] in their enrichment procedure. Without predictive balancing, some processors may have nearly all of their elements scheduled for refinement which can lead to a memory overflow. In such an instance, the predictive technique would tend to disperse these elements among several other processors, thus reducing the likelihood of a memory allocation problem. For the results presented in Section 5, predictive load balancing is performed using OCT; however, any load balancing procedure that supports elemental weights may be used.

In preparation for refinement, mesh edges are marked for splitting based on error indicator information and a refinement threshold. These markings determine the three-dimensional refinement templates that will be applied during refinement (Section 3). The weighting factors assigned during predictive balancing are based on these subdivision patterns. The numbers in parentheses in Figure 1 indicate the weights associated with each edge marking. For an element that has no edges marked, a weighting factor of one is used.

Several factors may lead to a small imbalance during the refinement even with predictive balancing. An element is an atomic unit regardless of its weight, so its weight cannot be subdivided, which may be necessary to achieve an exact balance. Topological constraints, which cannot be determined efficiently beforehand, may force more subdivisions than predicted by the original edge marking. Imbalance would also result during refinement if many elements with large weights (those for which refinement will take place) come to reside on a few processors while the remaining processors have primarily lightly weighted elements (which will be left unchanged by the refinement procedure). The last case has not been seen in the tests performed thus far.

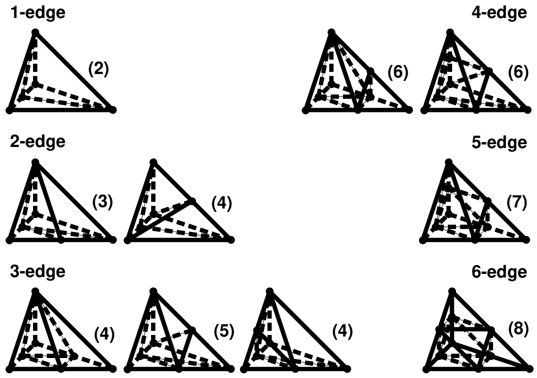

Figure 1. Weights for subdivision patterns and predictive load balancing.

Predictive balancing proved effective in balancing a transient flow problem solved using our Parallel Adaptive Euler Solver with time steps determined by the MOL [9]. Since the workload per element in the computation phase was homogeneous in this case, the *Uniform element workload Predictive balancing* (UP) technique described above also balanced the numerical calculations subsequent to each adaptive step. The UP method has also been used in tests with two different weighted balancing algorithms to improve enrichment and balancing times for a nonhomogeneous workload per element solution technique [11]. A standard partitioner was used to balance the computation.

When a UP method is used, an additional source of computational imbalance may arise as a result of (*i*) the mesh element migration that occurs during refinement vertex snapping and (*ii*) the migration, deletion, and creation of elements during the optimization stage. In practice, however, these changes tend to be small and localized with little impact on the subsequent computational load balance.

4.2. Variable workloads

A uniform workload per element is not appropriate in many situations where predictive balancing is desirable. UP has been extended to accept a user-defined cost function to be used in conjunction with the predictive method. This cost function is an element-based load estimator which can be tailored to the solution technique. The predictive balancing library provides access to the element subdivision weights (Figure 1), so the function can consider the effects of spatial refinement when generating load values.

As an example, consider the LRM technique discussed in Section 3. To balance the computation after spatial enrichment, element weights are assigned as in (2). If an element Ω_j is to be subdivided into n elements, $\Omega_{j_1}, \ldots, \Omega_{j_n}$, we may approximate the inscribed radius of Ω_{j_i}, $1 \leq i \leq n$, as

$$r_{j_i} = \left(\frac{3}{4\pi} \cdot \frac{V_j}{n} \right)^{\frac{1}{3}} \tag{3}$$

where V_j is the volume of the inscribed sphere of element Ω_j. The element weight is then computed as in (2).

Like UP, *Variable element workload Predictive balancing* (VP) generates less data movement during balancing than *a posteriori* balancing and, given a good estimator, eliminates the need for balancing after refinement. However, since the weighting used in the balancing is based not only on the number of elements produced by refinement but also on other factors needed to balance the computational load afterwards, this is a compromise between balancing the refinement stage of enrichment and allowing computation to be balanced without an *a posteriori* balancing. In addition, there may be solution techniques that generate heterogeneous workloads which are not easily estimated before new elements are created.

5. Results

Consider the three-dimensional unsteady compressible flow in a cylinder containing a cylindrical vent. This problem was motivated by flow studies in perforated muzzle brakes for large calibre guns [8]. A quasi-steady flow exists behind the contact surface for a short time. The larger cylinder (the shock tube) initially contains air moving at Mach 1.23 while the smaller cylinder (the vent) is quiescent. A Mach 1.23 flow is prescribed at the tube's inlet. Simulation flow parameters match those of shock tube studies of Dillon [8] and Nagamatsu *et al.* [13]. The simulation begins with an initial mesh of 69,572 tetrahedral elements distributed across 16 processors of an IBM SP2 computer. Figure 2 shows the Mach number with velocity vectors at time $t = 0.6$ in the simulation. Flow features compare favorably with experimental and numerical results of Nagamatsu *et al.* [13].

Iterative balancing methods like ITB tend to have low per-iteration costs. However, global repartitioners generally maintain better partition quality than ITB [4]. Thus, the standard nonpredictive method executes a few iterations of ITB between spatial enrichment stages followed by a global size-weighted repartitioning (using

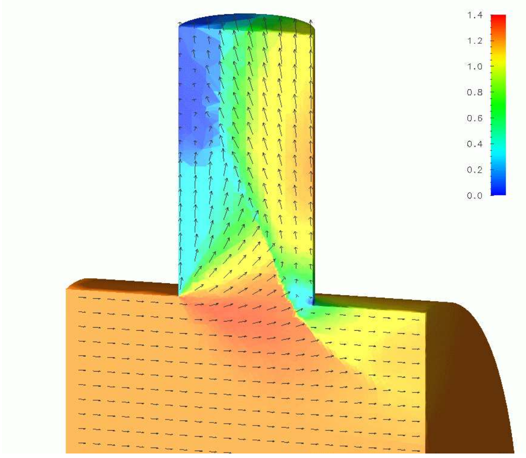

Figure 2. Projections of the Mach number and velocity vectors onto the surfaces of a perforated cylinder at time 0.6.

OCT) after enrichment. No such global repartitioning is necessary after VP, since it is designed to balance the computation stage. However, OCT is used with VP (VPOCT) in order to maintain good partition quality even with repeated application.

Over-refinement and mesh optimization options were disabled in these runs. For each given quantity, q, being compared, the relative percent differences given in this section were computed as

$$q_{\text{diff}} = \frac{q_{NONPRED} - q_{PRED}}{q_{NONPRED}} \times 100 \tag{4}$$

where $q_{NONPRED}$ is the quantity for the nonpredictive method and q_{PRED} is the quantity for a predictive technique. Thus, a positive q_{diff} indicates that the predictive method outperforms the nonpredictive one.

While this is essentially the same problem solved in [11], the test conditions have changed enough to warrant presenting a new set of results to ensure that improvements in other areas, such as solver efficiency and estimator performance, did not impact the advantages of VP. These changes include the addition of global shape control to the spatial enrichment, improvements to the predictive library, and an adjustment to the estimator for LRM.

5.1. Migration volume

The improvement in data migration over the nonpredictive method using VPOCT is shown in Figure 3. Since VPOCT attempts to balance the computational load without an *a posteriori* rebalancing, its data movement is compared to the combined migration generated by ITB during enrichment and by OCT afterwards. The average improvement on a mesh by mesh basis ranged from 55-83%. Thus, VP consistently outperforms nonpredictive balancing to a large degree in this area.

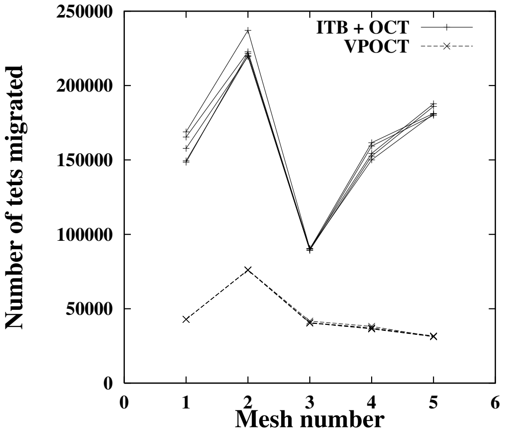

Figure 3. Comparison of data migrated for the nonpredictive method vs. VPOCT.

5.2. Enrichment and balancing times

Combined time spent in spatial enrichment and balancing for a sequence of runs is presented in Figure 4. These results are expressed as percent differences comparing each predictive run to a corresponding nonpredictive run. The values compare the VPOCT enrichment time against the combined enrichment and *a posteriori* OCT balancing time for the nonpredictive method. The graph shows a 38-71% improvement in combined enrichment and balancing times for VP.

The same graph also gives the percent improvements in the average combined time taken by refinement and balancing for the method as compared to the nonpredictive scheme. On average, VP bests nonpredictive balancing by 47-69% in this comparison.

5.3. Estimator accuracy

The accuracy of the LRM estimator function for the VP method is reflected in the resulting weight imbalance. The values given in Figure 5 were computed using the weight function given by (2). Preliminary results presented in [11] showed a 15-45% imbalance level with an average imbalance of 28%; however, this has been improved to imbalances from 4-23% with an average

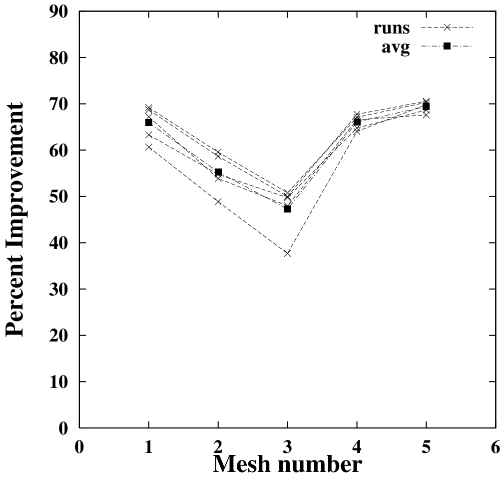

Figure 4. Time spent in enrichment and balancing for a sequence of five predictive runs relative to nonpredictive runs.

imbalance of 10%. Note that the 23% is a spike. The percentages returned to 10% for the subsequent mesh which is not shown because corresponding data is not available for the nonpredictive run. This indicates that while not exactly matching OCT, the estimator under the current system provides a good approximation of the size weighting used in *a posteriori* balancing. Such a level of accuracy means that the efficiency of the computation stage will remain high. More evidence of this follows.

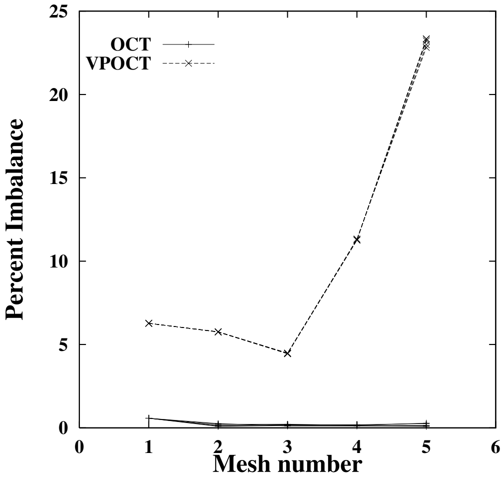

Figure 5. Estimator accuracy for VPOCT.

5.4. Computational imbalance

The variation seen in some comparisons between runs is due in large part to variations in the spatial enrichment process. Mesh data migrated to a processor are added to the local data structure's linked lists in the order they arrive. Because enrichment is performed in the order that elements are encountered, different meshes can result from the same input mesh and error indicators. Additionally, the enrichment process may perform different mesh operations depending on the partitioning so as to avoid unnecessary element migration. The impact of this on enrichment and balancing times across runs has already been seen. There is also a large influence on solution times because the size distribution of elements has a major effect on the LRM used by the solver. Thus, variations in solution times prevent drawing any conclusions on the relative performance of VPOCT and size-weighted OCT after enrichment in terms of time spent in computation.

However, two important and accurate measures of computational imbalance that exhibit less variation between runs have been developed [7, 10]. *Time-step imbalance* is defined as the maximum number of elements time stepped on a processor normalized by the average number stepped on all processors. Likewise, the *flux imbalance* is the maximum number of fluxes computed on a processor relative to the average number computed on all processors. In either case, let the average imbalance at simulation time t be a weighted average of all imbalances to time t. The weighting is the wall-clock duration of an imbalance relative to the total wall-clock time of the computation.

Average flux and time-step imbalances are shown in Figures 6 and 7 for a series of five runs using both the nonpredictive technique and VPOCT. As seen in the graphs, VPOCT maintains good flux and time-step imbalances as compared to size-weighted OCT performed after enrichment. In these runs, it differed from OCT by 1-6% in average flux imbalance and by 1-4% in average time-step imbalance. Thus, despite balancing before refinement to move a coarser structure, VPOCT still maintains a reasonable computational load balance for this problem.

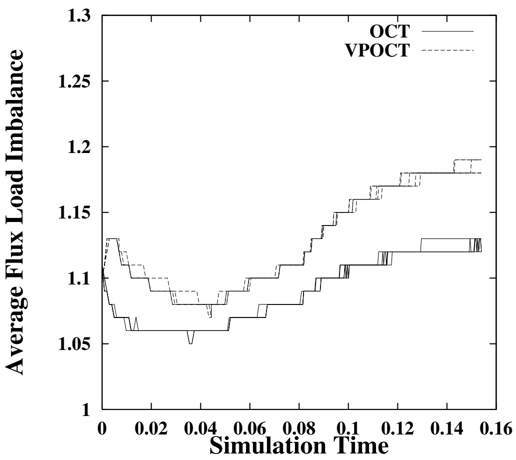

Figure 6. Average flux load imbalances for a sequence of five runs using nonpredictive and predictive balancing.

6. Message passing enhancements

MPI is a portable and widely available library for sending messages in a multiprocessor environment. However, parallel adaptive computation has particular communications requirements, and in several situations the primitives provided by MPI make implementing these inconvenient or inefficient. We describe here a set of library functions layered on top of MPI that accomplish these operations more easily and efficiently. Specifically, we focus on dynamic send and receive buffer allocation and automatic message packing.

6.1. Message buffer allocation

Sending messages. An adaptive solution typically involves dynamic linked data structures such as those of PMDB (Section 2). Communication may be performed selectively depending on the contents of a data structure. However, it is often impossible to determine the number of messages to be sent without actually traversing the data structure, effectively doing the work that would be required to send the messages. It is desirable to send messages during the initial traversal, but buffer space for the outgoing messages cannot be allocated in advance because the size is unknown. Dynamic message buffer allocation is needed to send the messages without a second traversal.

The user code may allocate the message buffer memory itself and call MPI to send the buffer (*e.g.* , via `MPI_Isend()`). The user code is responsible for managing the memory buffers: allocating them before the message is sent, polling to detect completion, and finally deallocating them. This is cumbersome at this level, especially if the number of buffers is not known *a priori*.

The only other option under MPI is the buffered send mode (`MPI_Bsend()`). This manages outgoing message buffer space transparently, simplifying user code as

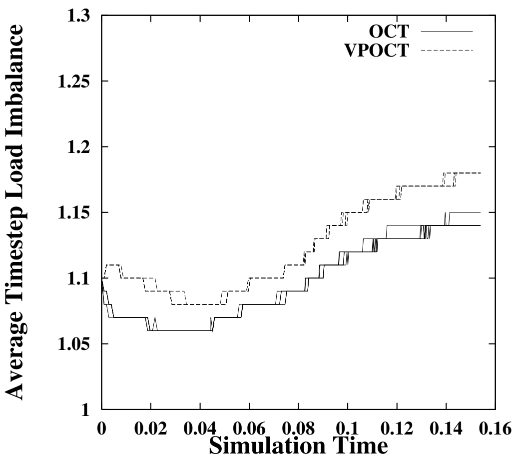

Figure 7. Average time-step load imbalances for a sequence of five runs using nonpredictive and predictive balancing.

it need not be concerned with the message buffer after the send call. However, `MPI_Bsend()` has two serious drawbacks. The memory used by these calls must be allocated in advance using `MPI_Buffer_attach()` and MPI does not change its size dynamically. A buffer which is too small causes failure when large data is sent, while a buffer which is too large may cause user code to fail due to lack of memory available for other purposes. Secondly, if the message cannot be sent immediately, the user message is copied into this buffer space, introducing an expensive and unnecessary memory-to-memory copy.

To improve the situation, we define a set of library routines which dynamically allocate, send, and free message buffers. To compose a message, the user code calls a library routine, `com_allocate_buffer()`, which dynamically allocates and returns an outgoing message buffer, much like `malloc()`. The routine allocates memory in large chunks and dispenses it as requested. The underlying allocation size is set by the user and may be adjusted as appropriate.

Once the user code has written the message data to the buffer, a call to `com_send_buffer()` initiates the send. At this point, management of the buffer is the exclusive responsibility of the library. This is only possible when the message is originally allocated by the library. This scheme provides the advantages of MPI's buffered send mode while avoiding the potential memory-to-memory copying.

User code may reclaim memory previously used for outgoing send buffers with a call to a garbage collection routine, `com_free_send_buffers()`. It checks the allocated send buffers, frees those no longer in use, and reports whether any outstanding buffers remain. If so, the user code should call it periodically if it wishes to free the remaining buffer memory. Often, there will be a point in the user code where the sends can be expected to have completed. The overhead of the garbage collection is minimal in that situation. In any case, it is no greater than that of managing the buffers directly in the user code.

Receiving messages. A similar buffer management problem exists on the receiver side. User code does not typically know the quantity or sizes of the messages it will receive.

The MPI receive routines (*e.g.*, `MPI_Irecv()`) require the user to provide a buffer for incoming messages. In cases where the message size is unknown, the user must either assume a maximum size or query MPI (*e.g.*, via `MPI_Iprobe()`) before allocating the receive buffer. The former is undesirable because of the adaptive nature of the user code. The latter provides the needed information, but its semantics are inefficient. To determine the size of the incoming message, it must be the case that either MPI has already received it internally or it must perform communication to query the sender. In the former case, inefficiency results from the lack of an MPI operator to receive this message without copying it to a user-supplied buffer. In the latter case, an unnecessary round-trip communication would result.

Given these constraints, we provide library support for dynamic allocation and deallocation of received messages. Rather than providing a buffer for the incoming message, the user simply calls a library routine (`com_receive_buffer()`) specifying the type of message it desires. If a message matching the criteria is available, the library dynamically allocates a buffer for the message, receives the message, and returns a pointer to the user. These semantics minimize the memory-to-memory copying during the receive.

When the user has finished processing the message buffer, it calls `com_free_receive_buffer()`, analogous to the standard `free()` routine.

6.2. Automatic message packing

Another difficulty in performing efficient communication in a parallel adaptive environment relates to message size. The most straightforward implementation of an algorithm often results in sending many small messages. For example, in the current version of our Parallel Adaptive Euler Solver [10], each time step of an element on an interprocessor boundary generates communication. Only 52 bytes of information must be passed to the adjoining element on the neighboring processor, and it would be simplest to send this information in individual messages. However, startup costs for message passing may be large, reducing efficiency when sending multiple small messages. The user code may explicitly pack the messages into a single message (as in the implementation of PMDB), but this adds complexity to the user code.

MPI includes library routines (`MPI_Pack()` and `MPI_Unpack()`) which eliminate the problem of sending the small, inefficient messages, but user code must still allocate and manage the buffers to be packed. Construction of the packed object incurs an unnecessary memory-to-memory copy, and packed messages must be specially handled by the receiving user code to unpack the contents.

We introduce a mechanism for automatically packing outgoing messages. Only two modifications to the buffer management scheme are necessary to accomplish this. First, when the user code allocates an outgoing message buffer it must specify the destination processor. This allows the grouping of the messages in memory from their inception and eliminates the need to copy the messages for packing. As in the non-packing mode, the user notifies the library that it may send the message, but here the library will defer the actual send until it has accumulated a sufficiently large package. Also, the user must call a library routine (`com_flush_sends()`) to force sending of any incomplete packages when no additional messages remain.

The library's memory allocation size also determines the package size. When a new outgoing message buffer is requested which will not fit in the current memory block, the package is considered full. A new memory block is allocated, and a pointer into the new block is returned to the user. Once the full block contains only messages that the user has asked to be sent, the entire block is sent off as a single message.

The call to flush the buffers has no effect when packing is not selected. Thus, performance tuning by enabling or disabling packing and by varying the package size is done by simply changing these parameters, with no other user code changes.

On the receiving side, no changes to the user code are necessary to handle the packed messages. When a package is received by the library, it is automatically unpacked; the individual messages are returned by calls to `com_receive_buffer()` in the same manner as for unpacked messages.

7. Conclusions

Variable element workload predictive balancing tends to reduce the volume of data migrated during balancing as compared to nonpredictive methods, showing a 72% average improvement for the examples presented here. In addition, combined enrichment and balancing times were improved by approximately 62% on average for VP. The average size-weighted imbalance produced by the predictive method was 10%, indicating that the workload estimator used for balancing the LRM computation worked well. Additionally, the overall mean for the average flux imbalance after VP was within 4.0% of that produced by the nonpredictive method, and the predictive technique was within 2.6% of OCT with respect to average time-step imbalance. Thus, the VP technique shows improvements in volume migrated and combined enrichment and balancing times over the nonpredictive method while maintaining a high degree of efficiency in the computation phase.

This technique reduces the time spent in enrichment and allows a single load balancing invocation to replace several calls performed by nonpredictive balancing. However, as seen here, computation performed on the resulting partitions is generally no more efficient than calculations done on partitions generated by an *a posteriori* method. Hence, a problem which adapts the mesh infrequently will show less improvement in overall performance than one that requires frequent enrichments.

Adaptive p-refinement is being added to the system and will require extension of the predictive balancing to account for the number of degrees of freedom associated with each element.

We are also investigating ways to reduce or eliminate factors that produce variations across runs having the same parameters. Currently, these variations make the direct use of wall-clock timings in evaluating solution efficiency very difficult.

The message passing enhancements discussed here have shown promise in small test cases. We are working to refine them and plan to gather data on their performance using larger tests. Their design will also facilitate easy incorporation into the Parallel Euler Solver which will allow us to present performance results from a computational fluid dynamics problem.

Acknowledgements

We would like to thank our colleagues in Computer Science and in the Scientific Computation Research Center at Rensselaer for the use of their software and many valuable suggestions. Computer systems used in the development and analysis runs include the 36-node IBM SP2 computer at Rensselaer, the 400-node SP2 at the Maui High Performance Computing Center, and the 512-node SP2 at the Cornell Theory Center. Authors were supported by AFOSR Grant F49620-95-1-0407, ARO grant DAAH04-95-1-0091, and NSF Grant CCR-9527151.

References

[1] S. T. Barnard. PMRSB: parallel multilevel recursive spectral bisection. In F. Baker and J. Wehmer, editors, *Proc. Supercomputing 95*, San Diego, December 1995.

[2] M. W. Beall and M. S. Shephard. A general topology-based mesh data structure. To appear *Int. J. Numer. Meth. Engng.*, 1997.

[3] R. Biswas, K. D. Devine, and J. E. Flaherty. Parallel, adaptive finite element methods for conservation laws. *Appl. Numer. Math.*, 14:255–283, 1994.

[4] C. L. Bottasso, J. E. Flaherty, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. The quality of partitions produced by an iterative load balancer. In B. K. Szymanski and B. Sinharoy, editors, *Proc. Third Workshop on Languages, Compilers, and Runtime Systems*, pages 265–277, Troy, 1996.

[5] B. Cockburn, S.-Y. Lin, and C.-W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: One-Dimensional systems. *J. Comput. Phys.*, 84:90–113, 1989.

[6] B. Cockburn and C.-W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws II: General framework. *Math. Comp.*, 52:411–435, 1989.

[7] K. D. Devine, J. E. Flaherty, R. Loy, and S. Wheat. Parallel partitioning strategies for the adaptive solution of conservation laws. In I. Babuška, J. E. Flaherty, W. D. Henshaw, J. E. Hopcroft, J. E. Oliger, and T. Tezduyar, editors, *Modeling, Mesh Generation, and Adaptive Numerical Methods for Partial Differential Equations*, volume 75, pages 215–242, Berlin-Heidelberg, 1995. Springer-Verlag.

[8] R. E. Dillon Jr. A parametric study of perforated muzzle brakes. ARDC Technical Report ARLCB-TR-84015, Benet Weapons Laboratory, Watervliet, 1984.

[9] J. E. Flaherty, R. M. Loy, C. Özturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. SCOREC Report 22-1996, Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, 1996. To appear *Appl. Num. Math.*

[10] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load-balancing for the parallel solution of three-dimensional conservation laws. IMA Preprint Series 1483, Institute for Mathematics and its Applications, University of Minnesota, 1997. Submitted for publication.

[11] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Predictive load balancing for parallel adaptive finite element computation. In H. R. Arabnia, editor, *Proceedings PDPTA '97*, volume I, pages 460–469, 1997.

[12] G. Karypis and V. Kumar. Metis: Unstructured graph partitioning and sparse matrix ordering system. Technical report, University of Minnesota, Department of Computer Science, Minneapolis, MN, 1995.

[13] H. T. Nagamatsu, K. Y. Choi, R. E. Duffy, and G. C. Carofano. An experimental and numerical study of the flow through a vent hole in a perforated muzzle brake. ARDEC Technical Report ARCCB-TR-87016, Benet Weapons Laboratory, Watervliet, 1987.

[14] L. Oliker, R. Biswas, and R. C. Strawn. Parallel implementaion of an adaptive scheme for 3D unstructured grids on the SP2. In *Proc. 3rd International Workshop on Parallel Algorithms for Irregularly Structured Problems*, Santa Barbara, 1996.

[15] A. Patra and J. T. Oden. Problem decomposition for adaptive *hp* finite element methods. *Comp. Sys. Engng.*, 6(2):97, 1995.

[16] A. Pothen, H. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Mat. Anal. Appl.*, 11(3):430–452, 1990.

[17] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. Technical Report 97-014, University of Minnesota, Department of Computer Science and Army HPC Center, Minneapolis, MN, 1997.

[18] M. S. Shephard, J. E. Flaherty, H. L. de Cougny, C. Özturan, C. L. Bottasso, and M. W. Beall. Parallel automated adaptive procedures for unstructured meshes. In *Parallel Computing in CFD*, number R-807, pages 6.1–6.49. Agard, Neuilly-Sur-Seine, 1995.

[19] M. S. Shephard and M. K. Georges. Automatic three-dimensional mesh generation by the Finite Octree technique. *Int. J. Numer. Meth. Engng.*, 32(4):709–749, 1991.

[20] A. Sohn, R. Biswas, and H. D. Simon. Impact of load balancing on unstructured adaptive computations for distributed-memory multiprocessors. In *Proc. Eighth IEEE Symposium on Parallel and Distributed Processing*, pages 26–33, New Orleans, LA, October 1996.

[21] R. Van Driessche and D. Roose. An improved spectral bisection algorithm and its application to dynamic load balancing. *Parallel Computing*, 21:29–48, 1995.

[22] C. H. Walshaw and M. Berzins. Dynamic load balancing for PDE solvers on adaptive unstructured meshes. *Concurrency: Practice and Experience*, 7(1):17–28, 1995.