Impact of Load Balancing on Unstructured Adaptive Grid Computations for Distributed-Memory Multiprocessors*

Andrew Sohn
CIS Dept., New Jersey Institute of Technology, Newark, NJ 07102, sohn@cis.njit.edu
Rupak Biswas
RIACS, NASA Ames Research Center, Moffett Field, CA 94035, rbiswas@nas.nasa.gov
Horst D. Simon
NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, simon@nersc.gov

Abstract

The computational requirements for an adaptive solution of unsteady problems change as the simulation progresses. This causes workload imbalance among processors on a parallel machine which, in turn, requires significant data movement at runtime. We present a new dynamic load-balancing framework, called JOVE, that balances the workload across all processors with a global view. Whenever the computational mesh is adapted, JOVE is activated to eliminate the load imbalance. JOVE has been implemented on an IBM SP2 distributed-memory machine in MPI for portability. Experimental results for two model meshes demonstrate that mesh adaption with load balancing gives more than a sixfold improvement over one without load balancing. We also show that JOVE gives a 24-fold speedup on 64 processors compared to sequential execution.

1. Introduction

Unsteady flow computations in complex three-dimensional domains is a challenging task. It is particularly daunting when dynamic mesh adaption is used on unstructured grids. The computational requirements for such problems are extremely large both in terms of processing time and in-core memory, and can only be satisfied by large-scale machines [6, 8]. During a typical adaptive, unsteady computational fluid dynamics (CFD) calculation, the unstructured meshes are locally refined and/or coarsened to capture important flow features. As a result, the computational intensity is not only time dependent, but also varies spatially over the problem domain.

A parallel implementation of such computational methods on distributed-memory machines typically requires two steps [8, 12]. First, the computational mesh is partitioned into smaller submeshes. Second, the partitioned submeshes are assigned to processors based on a mapping strategy. While this static partitioning and mapping approach is adequate for CFD calculations that do not change in computational intensity over time, it is grossly inefficient for unsteady, adaptive calculations. This is because as the computational behavior changes, some processors may have a lot more work than others. The imbalance in the processor loads implies that the initial partitioning of the mesh is no longer acceptable. It is thus imperative that the amount of work assigned to each processor be balanced at runtime to increase processor utilization and improve performance [4, 8].

Balancing the runtime computational load, however, is usually very difficult due to several reasons. These include a reliable measurement of the computational load, the amount of runtime data movement, and the minimization of interprocessor communication. Various methods on dynamic load balancing have been reported to date by numerous researchers; however, most of them lack a global view of loads across processors. A systematic way of measuring loads across all processors and then utilizing that information to balance the workload are needed for a method to be applicable to a variety of realistic applications. For example, the local detection and balancing of loads only among neighboring processors may be inadequate for large scientific applications as it could leave some processors unbalanced. At the same time, the redistribution of processor loads must be efficient so as not to significantly delay the main application. If parallel CFD is to be successful on distributed-memory multiprocessors for practical problems, it is essential that a dynamic load balancing method be developed in a such way that all necessary modules can be combined together to collectively act as a coherent tool. Our purpose is to build such an environment for runtime load balancing with unstructured mesh adaption for unsteady CFD applications.

*Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP'96), New Orleans, LA, October 23–26, 1996.

The dynamic load balancer, called JOVE, is intended to satisfy these requirements. It performs its task by comparing the computational gain for a balanced workload against the communication penalty arising from the data redistribution. Each time the computational mesh is adapted, JOVE decides, based on the information collected from all processors, whether repartitioning will be beneficial. If data movement is expensive compared to the computational gain, the mesh is not repartitioned and the CFD simulation continues without interruption. If, on the other hand, JOVE determines that the cost of data movement is compensated by the improved load balance, the CFD application is interrupted to redistribute the data based on the new partitioning. The numerical simulation is then restarted.

JOVE possesses three novel features. First, a dual graph representation of the computational mesh is used to keep the complexity and connectivity constant during the course of an adaptive computation. Second, a new inertial spectral mesh partitioning method [9] is introduced that performs both faster and better than Recursive Spectral Bisection [7]. Finally, accurate metrics for the computational gain and the communication cost are developed to measure and balance the processor loads between successive adapted grids.

2. Background

2.1. Unstructured tetrahedral mesh adaption

CFD problems are usually represented as a grid of vertices and elements. Flowfield solutions are typically stored at the vertices while an element represents some computation associated with it. During an adaptive calculation, the unstructured mesh is locally refined and/or coarsened to capture important flow features. The mesh adaption scheme used in this work is 3D_TAG [2] which has an edge-based data structure; that is, each tetrahedral element is defined by its six edges rather than by its four vertices. This edge data structure makes the mesh adaption procedure capable of performing anisotropic refinement and coarsening.

At each mesh adaption step, tetrahedral elements are targeted for coarsening, refinement, or no change by computing an error indicator for each edge. Edges whose error values exceed a user-specified upper threshold are targeted for bisection. Similarly, edges whose error values lie below another user-specified lower threshold are targeted for removal. Only three subdivision types are allowed for each tetrahedral element. The 1:8 isotropic subdivision is implemented by adding a new vertex at the mid-point of each of the six edges. The 1:4 and 1:2 subdivisions can result either because the edges of a parent tetrahedron are targeted anisotropically or because they are required to form a valid connectivity for the new mesh. When an edge is bisected, the solution vector is linearly interpolated at the mid-point from the two points that constitute the original edge.

Mesh refinement is performed by first setting a bit flag to one for each edge that is targeted for subdivision in every element that shares it. The edge markings for each element are then combined to form a binary pattern. Elements whose patterns do not match the allowed types are continuously upgraded until none of the edges shows any further change. Each element is then independently subdivided based on its binary pattern. Special data structures are used in order to ensure that this process is computationally efficient.

Mesh coarsening also uses the edge-marking patterns. If a child element has any edge marked for coarsening, this element and its siblings are removed and their parent element is reinstated. The parent edges and elements are retained at each refinement step so they do not have to be reconstructed. Reinstated parent elements have their edge-marking patterns adjusted to reflect that some edges have been coarsened. The mesh refinement procedure is then invoked to generate a valid mesh.

2.2. Dynamic load balancing

A parallel implementation of CFD methods on multiprocessors requires the computational mesh to be divided into smaller grids, each of which is then assigned to a processor. The degree of connectivity and the computational intensity of individual elements determine how they should be grouped to form the subgrids. This partitioning must be done in a way that approximately balances the computational workload among processors.

Figure 1 shows how mesh adaption adversely affects processor loads. To simplify the presentation, a small two-dimensional example is used. The mesh shown in Fig. 1(a) consists of 18 triangular elements. Assuming that four processors are used and that the computational intensity is uniform for all elements, the mesh is initially divided into four subgrids by applying graph partitioning. Processors P0 and P1 are assigned five elements each, while processors P2 and P3 have four elements each.

Changes in the computational mesh due to adaption makes parallel CFD difficult. As the numerical simulation progresses, some regions of the grid may contain more elements due to refinement while other regions may contain fewer due to coarsening. Figure 1(b) clearly indicates this after one refinement step. P0 still has 5 elements; however, P1, P2, and P3 have 13, 12, and 6 elements, respectively. The mesh adaption will cause P1 and P2 to perform more than twice the work of P0. Obviously, there is a severe load imbalance. If another adaption step is performed, the imbalance is likely to become even more critical, resulting in poor performance. In the extreme case, the use of a parallel machine would offer little advantage over sequential ones.

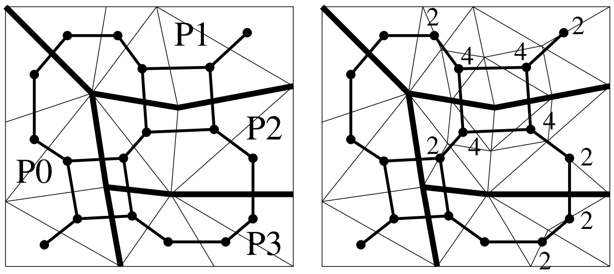

Figure 1. Initial and adapted meshes showing the need for dynamic load balancing. Also shown are the dual graph and the computational weights on the adapted mesh.

As the two snap shots shown in Fig. 1 suggest, it is extremely important to dynamically repartition the new grid; however, it is not straightforward as there are many technical issues involved. Repartitioning must be quick so that there is no significant delay in the CFD calculation. Post-partitioning steps must then be able to estimate the computational gain and the communication cost to decide whether the new partitions are worth accepting.

3. JOVE: The dynamic load balancing scheme

3.1. Overview

It has been shown that dynamic load balancing is absolutely necessary for unsteady adaptive CFD calculations. Figure 2 gives an overview of our approach to dynamic load balancing. The system consists of three modules: the load balancer JOVE, a CFD flow solver [1, 12] and the 3D_TAG mesh adaptor [2]. Details of the CFD solver are beyond the scope of this paper, except to note that it generates error values for each edge that are then used by 3D_TAG to refine and/or coarsen the mesh.

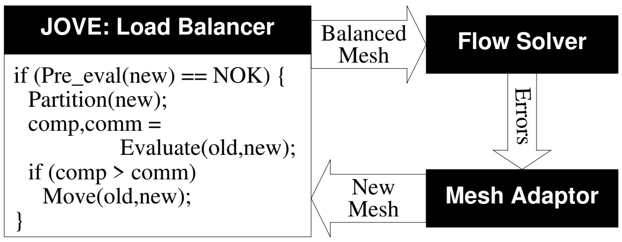

Figure 2. Dynamic load balancing framework.

The first step of JOVE is `Pre_eval(new)` which determines if the new mesh warrants further action in terms of repartitioning and processor reassignment. The objective is to rapidly decide whether the mesh has changed significantly enough to consider repartitioning. If `Pre_eval(new)` recommends repartitioning, the `Partition(new)` step divides the new mesh into subgrids. A new inertial spectral bisection algorithm [9] is used to rapidly update a partition from one grid to the next. The `Evaluate(old,new)` step consists of assigning partitions to processors such that the communication cost for data migration is minimized. It calculates two numbers: the computational gain `comp` that would be achieved by having a balanced partitioning, and the communication cost `comm` of actually moving all the data to correctly map partitions to processors. If `comp` is larger than `comm`, it is advantageous to use the new partitioning. In that case, the CFD simulation is interrupted while all the necessary data is redistributed based on the processor assignments. The CFD calculation is then restarted on the new partitions. Otherwise, the new partitioning is discarded and JOVE waits for the next adapted mesh.

3.2. Dual graph representation

The dual graph representation of the initial mesh is one of the key features of this work. CFD flow solvers usually solve for the solution variables at the vertices of the computational mesh. A parallel implementation requires a partitioning of the computational mesh such that each element belongs to a unique partition. Communication is required across faces that are shared by adjacent tetrahedral elements residing on different processors. Hence for the purposes of partitioning, we consider the dual of the original CFD mesh (cf. Fig. 1). The tetrahedral elements of the CFD mesh are the vertices of the dual graph. An edge exists between two dual graph vertices if the corresponding elements share a face in the original mesh. A graph partitioning of the dual graph thus yields an assignment of tetrahedra to processors.

Each dual graph vertex has two parameters associated with it. The computational weight, w_{comp}, is a measure of the workload for the corresponding element of the CFD mesh. The communication weight, w_{comm}, measures the cost of moving the element from one processor to another. The connectivity pattern and the w_{comp} determine how dual graph vertices should be grouped to form partitions that minimizes the disparity in the partition weights. The w_{comm} determine how partitions should be assigned to processors such that the cost of data movement is minimized.

The most significant advantage of using a dual graph is that its complexity and connectivity remains *unchanged* during the course of an adaptive computation. This is because the vertices of the dual graph correspond to the elements of the initial CFD mesh. The partitioning and load-balancing times therefore depend only on the initial problem size. New grids obtained by mesh adaption are translated to the two weights, w_{comp} and w_{comm}, for every element in the initial

CFD mesh. The normalized w_{comp} values greater than unity are shown for the dual graph vertices in Fig. 1(b).

3.3. Preliminary evaluation of adapted meshes

The objective of the `Pre_eval(new)` step in JOVE is to rapidly determine if the dual graph with a new distribution of computational weights should be considered for repartitioning. If projecting the new values of w_{comp} on the current partitions indicates that they are adequately load balanced, there is no need to repartition the mesh. In that case, JOVE terminates and the CFD application continues uninterrupted on the current partitions.

A proper metric is required to measure the load imbalance. If W_{max} is the sum of the w_{comp} on the most heavily-loaded processor, and W_{avg} is the average load across all processors, the average idle time for each processor is $(W_{\text{max}} - W_{\text{avg}})$. This is an exact measure of the load imbalance. The mesh is repartitioned if the imbalance factor $W_{\text{max}}/W_{\text{avg}}$ is greater than a user-specified threshold.

3.4. Dynamic inertial spectral mesh partitioning

If the preliminary evaluation step determines that the dual graph with a new weight distribution is unbalanced, JOVE invokes the mesh partitioning procedure. Several partitioning algorithms are available for unstructured grids; however, a new procedure that combines the high quality of spectral methods [7] with an efficient update strategy is used. This dynamic spectral bisection algorithm [9] is based on the center of inertia of the unpartitioned dual graph vertices and utilizes information from the initial spectral partitioning. It is thus capable of rapidly updating a partition from one grid to the next. The following algorithm explains the method:

```
for (i=0; i <log(npart); i++)        /* npart = #partitions */
  for (j=0; j < 2^i; j++) {
    Find an inertial vector of the unpartitioned vertices
    Construct an inertial matrix using the inertial vector
    Symmetrize the inertial matrix
    Find the eigenvectors of the inertial matrix
    Project vertex coordinates on eigenvector 0
    Sort projected coordinates
    Divide the unpartitioned vertices into two sets
  }
```

3.5. Similarity metric construction for evaluation

The objective of the evaluation step is to map new partitions to processors such that the communication cost for redistributing data is minimized. It begins by computing a similarity measure S that indicates how the communication weights of the new partitions are distributed over the old partitions. It is represented as a matrix where S_{ij} is the sum of the communication weights of all the dual graph vertices that have moved from old partition i to new partition j.

Consider, for example, a dual graph that generates the measure S in Fig. 3(a) after a repartitioning among eight processors. Only the non-zero entries are shown. Note that there are only three non-zero entries in the first row. This means that the vertices in old partition 0 have been distributed over new partitions 0, 1, and 3. Also, it would cost 389 to move those vertices in old partition 0 that are common to new partition 0, 510 to move those that are common to new partition 1, and 120 to move those that are common to new partition 3.

3.6. Processor reassignment

A new partition j with the largest value of S_{ij} is called the *dominant* partition for old partition i. This is because the communication cost for moving data can be minimized by mapping the processor assigned to an old partition to its corresponding dominant partition. The shaded entries in Fig. 3(a) indicate the largest computational weight for each of the old partitions. These are called the dominant weights. A serious problem is evident by inspecting the dominant weights in Fig. 3(a). Even though every old partition has a dominant partition, every new partition is not necessarily dominant. This affects the new partitions in two ways. First, some new partitions are not dominant at all; their processor assignment entries are marked with an 'X'. Second, some new partitions are dominant for more than one old partition; their processor assignment entries are marked with a '?'.

Our goal is to assign each processor a unique partition. Thus, the dominant partitions need to be rearranged so that there is exactly one dominant weight in every row and column of the similarity matrix S. Processor assignment then simply consists of mapping each dominant partition to the processor to which the old partition was originally assigned. However, this rearrangement constitutes a difficult optimization problem [10]. Due to runtime constraints, a suboptimal solution is obtained in linear time. The following algorithm ensures that each new partition is designated as dominant for exactly one old partition:

```
for (i=0; i < npart; i++)              /* npart: #partitions */
  for (j=1; j < ndp[i]; j++) {     /* ndp[i]: #dom wghts */
    Find min dominant weight S_li from new partition i
    Find max non-dominant weight S_lk from old partition l
                      such that ndp[k] < 1
    Mark S_li non-dominant and S_lk dominant
    ndp[k]=1
  }
```

The inner loop is executed only for those partitions that have more than one dominant weight. Applying the above

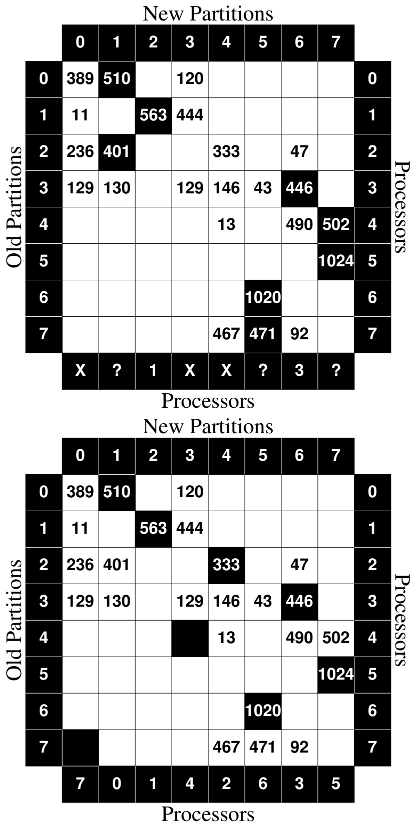

Figure 3. The similarity matrix (a) before and (b) after processor reassignment.

algorithm to the similarity matrix in Fig. 3(a) generates the new processor assignment shown in Fig. 3(b). In general, our method is also applicable if the number of partitions is an integer multiple of the number of processors.

3.7. Computational gain vs. communication cost

The computational gain of repartitioning is proportional to the decrease in the load imbalance achieved by running the adapted mesh on the new partitions rather than on the old partitions. Recall from Sec. 3.3 that the average load imbalance for each processor is given by $(W_{\text{max}} - W_{\text{avg}})$. The decrease in the amount of load imbalance due to the new partitioning on P processors is therefore $P(W^{\text{old}}_{\text{max}} - W^{\text{new}}_{\text{max}})$, where $W^{\text{old}}_{\text{max}}$ and $W^{\text{new}}_{\text{max}}$ are the sum of the computational weights on the most heavily-loaded processor for the old and new partitionings, respectively. If it requires T_{iter} μsecs to run one iteration of the CFD flow solver on one element of the original mesh, and if it is expected that the next mesh adaption will occur after N_{adapt} iterations of the flow solver, the total computational gain for the new partitioning is $PT_{\text{iter}}N_{\text{adapt}}(W^{\text{old}}_{\text{max}} - W^{\text{new}}_{\text{max}})$.

Calculating the communication cost is more complicated. The similarity matrix obtained after processor reassignment determines how data is to be redistributed. Models such as LogP [3] capture communication behavior with various parameters. We, however, use a model based on the similarity matrix and two machine-dependent parameters: the remote-memory latency time T_{lat} and the message setup time T_{setup}. T_{lat} is the time required for memory-to-memory copying of a word, and applies to every dual grid vertex that is moved. T_{setup} is the time required to prepare message headers, load the message buffer, and so on, and applies to each set of vertices that is moved from one processor to another.

Consider the similarity matrix in Fig. 4. Old partition 0 is distributed over new partitions 0, 1, and 3. However, data has to be moved only to partitions 0 and 3 because new partition 1 is assigned to P0, the same processor that old partition 0 was assigned to. This means that a total of 509 computational elements have to be moved from P0. Moreover, since the elements have to be sent to P4 and P7, the setup time for moving two sets of data also has to be included in the total cost. If the CFD and mesh adaption algorithms require M words of storage per computational element, and if C and N are the total number of elements and sets of elements to be moved, respectively, the total communication cost for mapping new partitions to processors is $CMT_{\text{lat}} + NT_{\text{setup}}$.

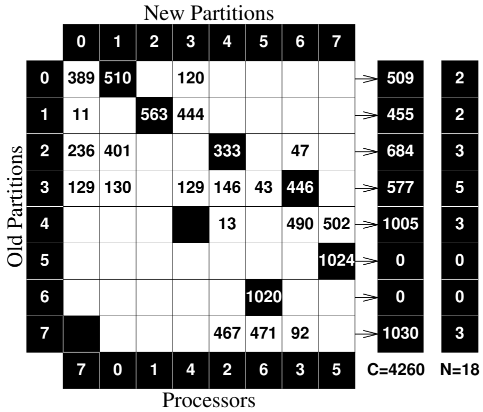

Figure 4. Calculating the total communication cost from the similarity matrix.

The new partitioning and mapping are accepted if the computational gain is greater than the communication cost. The numerical simulation is then interrupted to properly redistribute all the data based on the processor reassignment obtained from the similarity matrix. This completes the load balancing phase for one mesh adaption step.

4. Results and discussions

4.1. JOVE implemented on SP2

The load balancer JOVE, as described in Sec. 3, has been implemented on the IBM SP2 distributed-memory multiprocessor installed at NASA Ames Research Center. The code consists of approximately 3000 lines of C, with the parallel activities implemented in Message-Passing Interface (MPI) for portability. This does not include the 3D_TAG mesh adaption procedure which is another 4000 lines of C code. A master-worker parallel programming paradigm is used to simplify the implementation.

4.2. Test meshes and adaption simulation

Two model unstructured meshes are used in the experiments reported in this paper. The first mesh, called PARC, is two dimensional and has 1240 triangular elements. The second mesh, called BRICK, is three dimensional and has 2500 tetrahedral elements. Both are very small meshes, suitable for investigating fundamental issues in load balancing with reasonable execution times. Using realistic CFD meshes consisting of about a million elements would unnecessarily hinder our investigations as they have long execution times even on large-scale machines. Small meshes, on the other hand, allow us to look into the behavior of the load balancer in a reasonable time frame with a wide range of different parameters and settings.

The actual mesh adaption procedure has been simulated in parallel while retaining its typical behavior. Two fundamental issues need to be addressed in the simulation of mesh adaption: vertex selection and adaption modeling. Vertex selection refers to how and when dual graph vertices are selected as candidates for adaption. Vertices are randomly selected for adaption regardless of its partition number. At each iteration, a vertex is adapted if its id modulo a pseudorandom number lies within a certain range. Adaption modeling refers to how much computation each vertex should perform. Our adaption simulator is defined as three nested loops with the innermost consisting of a floating point division. Each loop has w iterations, where w is the weight of the vertex. Therefore, if a vertex with weight w is selected for adaption, its weight is set to w^3 and goes through w^3 iterations of floating point divisions. We have done substantial mesh adaption on realistic meshes in the past [5, 11] and find that this model is suitable for our experiments.

4.3. Anatomy of the execution time

We discuss how and where the total execution time is spent for each mesh adaption step. For typical, unsteady CFD calculations, the mesh adaption and load balancing phases are invoked several hundred times. However, it suffices to investigate for some reasonable number of adaptions to understand the behavior of the whole system. The execution time is measured for various steps and summarized into four categories: adaption, partitioning, evaluation/decision, and communication. Note that the communication time is a combination of several activities that include sending and receiving weights, and redistributing dual graph vertices among processors. Figure 5 shows the execution time profile for the first 30 adaptions on the BRICK mesh using 16 and 64 processors.

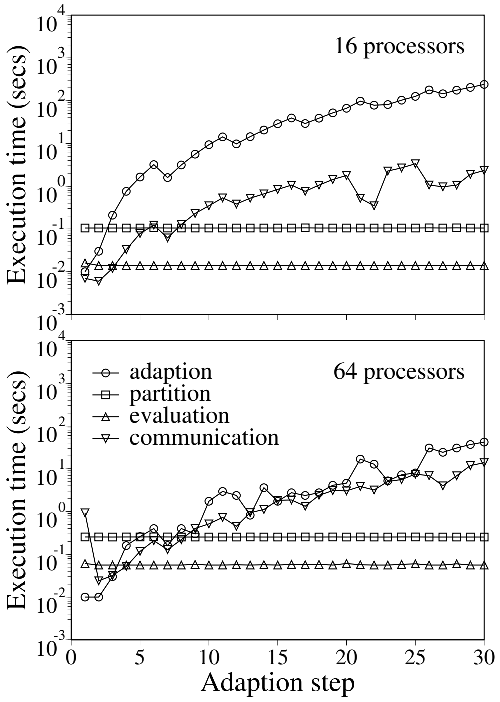

Figure 5. Anatomy of total execution time for BRICK mesh.

We can draw three major conclusions from the plots. First, we find that the partitioning and evaluation times are small compared to the adaption and communication times. It is also noteworthy that the partitioning and evaluation times remain constant throughout the simulation. However, as expected, the partitioning time increases with the number of processors. For example, the partitioning time is about 0.1 secs for 16 processors, but increases to 0.25 secs for 64 processors. This is not surprising because the master processor needs more time to partition the grid into 64 subgrids than into 16 subgrids.

Second, the mesh adaption and communication times dominate the total execution time. In particular, the adaption time is dominant when the number of processors is small, as seen in Fig. 5(a). There is an order difference between adaption and communication times. However, with 64 processors, the two times are comparable (cf. Fig. 5(b)). This trend is expected to continue as the number of processors increases; that is, the communication time will dominate when more processors are used. However, this is not alarming because the adaption time is artificially very small for the model problems. The typical execution time for one mesh adaption step for realistic problems is a few hundred seconds, not fractions of a second [5]. Thus, in real applications, the adaption time will almost always be much more than the communication time. The plots in Fig. 5 indicate that for most parallel applications, an increase in the number of processors will substantially lower the adaption time while increasing the communication time.

Finally, we have analyzed the execution time for only 30 adaptions. Full-scale, unsteady applications typically require several hundred mesh adaption steps. As observed from the plots, the execution time relentlessly increases as the number of adaptions increases.

4.4. Effect of mesh adaption on data movement

Figure 6 shows the percentage of dual graph vertices that are moved at runtime after each adaption. We present results for both the PARC and the BRICK meshes.

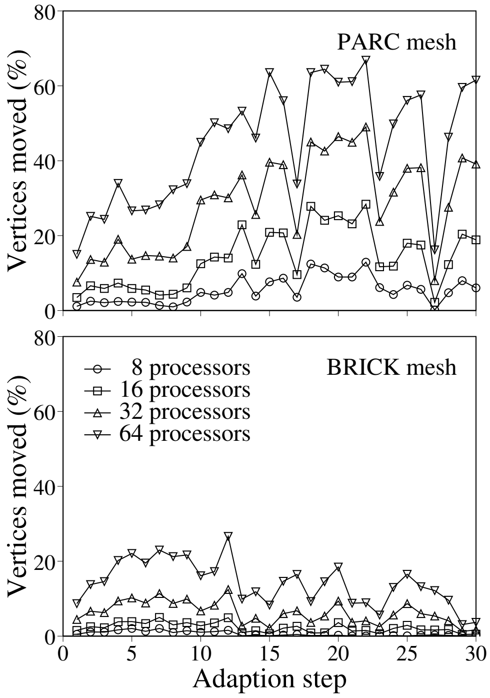

Figure 6. Percentage of dual graph vertices that are moved.

The plots demonstrate that the PARC mesh incurs a lot more relative data movement than the BRICK mesh. This is because BRICK has more vertices than PARC. We also find that the amount of data movement increases with the number of processors for both meshes. This explains the increase in the communication time in Fig. 5.

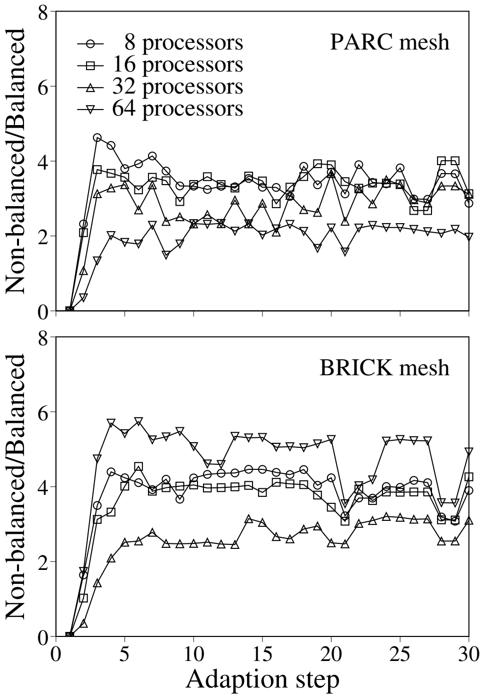

Figure 7. Comparison of total execution times with and without load balancing.

4.5. Impact of dynamic load balancing

Two sets of experiments were performed to measure the effectiveness of JOVE. These represent the key results of this paper. The same vertex selection and adaption modeling procedures were used with and without load balancing. Figure 7 illustrates the impact of load balancing on the total execution time. The plots show that when 8 processors are used for the BRICK mesh, the load balancing gives more than a threefold improvement over no load balancing. However, with 64 processors, the improvement is almost sixfold. In general, the results demonstrate that load balancing is highly nondeterministic but shows some gain for BRICK when the number of processors increases. This improvement is not observed for PARC primarily because it is a very small problem. We expect larger improvement for bigger meshes because of increased computation-to-communication ratio.

Figure 8 demonstrates the implication of this performance improvement when the load balancer JOVE is used with mesh adaption. When compared with the sequential version, JOVE demonstrates a 24-fold speedup for 40 adaption steps. For 10 adaptions, which is quite unrealistic, the speedup is only about 10. We also see a typical phenomenon of early saturation. However, the speedup consistently increases with the number of adaptions. For real problems with several hundred adaption steps, the speedup will increase further as the curves in Fig. 8 suggest.

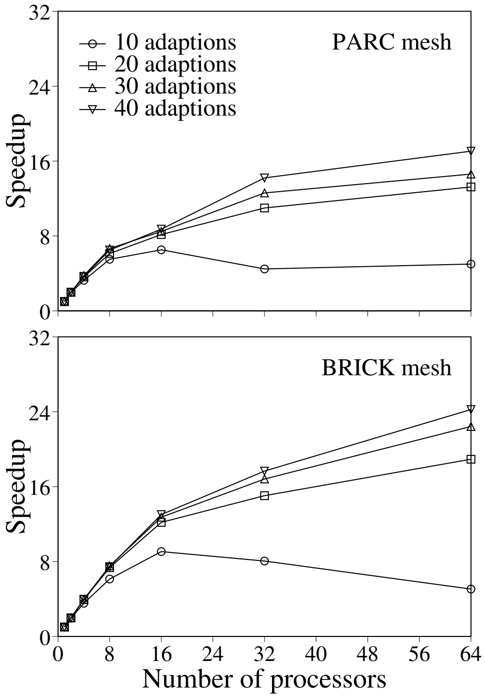

Figure 8. Parallel speedup of JOVE.

5. Conclusions

Dynamic load balancing for unstructured adaptive mesh computations is a complex task, involving many procedures and parameters. While typical load balancing schemes locally exchange information between neighboring processors, we have presented a new method called JOVE that dynamically balances loads across processors with a global view. JOVE has been implemented on an SP2 distributed-memory multiprocessor with approximately 3000 lines of C code. Parallel activities have been implemented in MPI for portability. We have used two model meshes for experiments: one with 1240 elements, and the other with 2500 elements. While these meshes are small, they are suitable for our investigations as the execution times are reasonable.

Two key observations can be made from the experiments reported in this paper. First, the JOVE load balancing module has given a sixfold improvement for mesh adaption, when compared with no balancing regardless of the number of processors. Second, JOVE has given a 24-fold speedup on 64 processors, when compared with a sequential single-processor version that has no parallel constructs. These observations are based on the measurement of only 30 mesh adaption steps. Results have indicated that performance will improve with more adaptions and larger meshes.

We have also drawn some other conclusions that clarify the behavior of load balancing for mesh adaption. First, the partitioning and evaluation times are negligible compared to the adaption and communication times, regardless of the number of processors. This has indicated that even the sequential version of the new inertial spectral partitioner is indeed quite fast. Second, the adaption time decreases while the communication time increases as the number of processors is increased. This is somewhat expected, and our future efforts will be focused on reducing the communication time. Finally, the number of vertices that are moved due to repartitioning does not appear to be a key factor that affects the effectiveness of load balancing. We have found that with 64 processors, performance still sustained a sixfold improvement even when 25% of all vertices were moved.

These experimental results have consistently demonstrated that the JOVE load balancer is effective for unstructured adaptive mesh computations. Our immediate goal is to run JOVE on large meshes with several hundred adaption steps that closely model full-scale experiments. We are planning on applying this method to various realistic applications including helicopter aerodynamics, semiconductor device modeling, and computational nanotechnology.

References

[1] T. J. Barth. A 3-d upwind euler solver for unstructured meshes. *10th AIAA CFD Conf.*, AIAA-91-1548, 1991.

[2] R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Appl. Numer. Math.*, 13:437–452, 1994.

[3] D. Culler, R. M. Karp, D. A. Patterson, A. Sahay, E. E. Schauser, E. Santos, A. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. In *4th ACM PPoPP*, 1993.

[4] K. D. Devine and J. E. Flaherty. Dynamic load balancing for parallel finite element methods with adaptive h- and p-refinement. In *7th SIAM Conf. on Par. Proc. for Sci. Comput.*, pages 593–598, 1995.

[5] E. P. N. Duque, R. Biswas, and R. C. Strawn. A solution adaptive structured/unstructured overset grid flow solver with applications to helicopter rotor flows. *13th AIAA Appl. Aero. Conf.*, AIAA-95-1766, 1995.

[6] P. Mehrotra, J. Saltz, and R. E. Voight. *Unstructured Scientific Computation on Scalable Multiprocessors*. MIT Press, 1992.

[7] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Comput. Sys. in Engrg.*, 2:135–148, 1991.

[8] H. D. Simon. *Parallel Computational Fluid Dynamics*. MIT Press, 1992.

[9] H. D. Simon and A. Sohn. Dynamic inertial spectral partitioning. in preparation.

[10] A. Sohn. Parallel n-ary speculative computation of simulated annealing. *IEEE Trans. on Par. Dist. Sys.*, 6:997–1005, 1995.

[11] R. C. Strawn, R. Biswas, and M. Garceau. Unstructured adaptive mesh computations of rotorcraft high-speed impulsive noise. *J. of Aircraft*, 32:754–760, 1995.

[12] V. Venkatakrishnan, H. D. Simon, and T. J. Barth. A mimd implementation of a parallel euler solver for unstructured grids. *J. of Supercomputing*, 6:117–137, 1992.