Load Balancing Unstructured Adaptive Grids for CFD Problems

Rupak Biswas* Leonid Oliker†

Abstract

Mesh adaption is a powerful tool for efficient unstructured-grid computations but causes load imbalance among processors on a parallel machine. A dynamic load balancing method is presented that balances the workload across all processors with a global view. After each parallel tetrahedral mesh adaption, the method first determines if the new mesh is sufficiently unbalanced to warrant a repartitioning. If so, the adapted mesh is repartitioned, with new partitions assigned to processors so that the redistribution cost is minimized. The new partitions are accepted only if the remapping cost is compensated by the improved load balance. Results indicate that this strategy is effective for large-scale scientific computations on distributed-memory multiprocessors.

1 Introduction

Dynamic mesh adaption on unstructured grids is a powerful tool for computing unsteady three-dimensional problems that require grid modifications to efficiently resolve solution features. By locally refining and coarsening the mesh to capture flowfield phenomena of interest, such procedures make standard computational methods more cost effective. Highly refined meshes are required to accurately capture shock waves, contact discontinuities, vortices, and shear layers. Local mesh adaption provides the opportunity to obtain solutions that are comparable to those obtained on globally-refined grids but at a much lower cost.

Unfortunately, the adaptive solution of unsteady problems causes load imbalance among processors on a parallel machine. This is because the computational intensity is both space and time dependent. An efficient parallel implementation of such methods has not yet been achieved, primarily because of the difficulties associated with the dynamically-changing nonuniform grid. Various methods on dynamic load balancing have been reported to date; however, most of them lack a global view of loads across processors.

Figure 1 depicts our framework for parallel adaptive flow computation. It consists of a flow solver and mesh adaptor, with a partitioner and mapper that redistributes the computational mesh when necessary. Our goal is to build a portable system for efficiently performing adaptive large-scale flow calculations in a parallel message-passing environment. The mesh is first partitioned and mapped among the available processors. A flow solver then runs for several iterations, updating solution variables. Once an acceptable solution is obtained, a mesh adaption procedure is invoked to generate a new computational mesh based on an error indicator. A quick evaluation step determines if the new mesh is

*Research Scientist, MRJ Technology Solutions, MS T27A-1, NASA Ames Research Center, Moffett Field, CA 94035

†Research Associate, Research Institute for Advanced Computer Science, MS T27A-1, NASA Ames Research Center, Moffett Field, CA 94035

sufficiently unbalanced to warrant a repartitioning. If the current partitions are adequately load balanced, control is passed back to the flow solver. Otherwise, a repartitioning procedure is invoked to divide the new mesh into subgrids. The new partitions are then reassigned to the processors in a way that minimizes the cost of data movement. If the remapping cost is less than the computational gain that would be achieved with balanced partitions, all necessary data is appropriately redistributed. Otherwise, the new partitioning is discarded and the flow calculation continues on the old partitions.

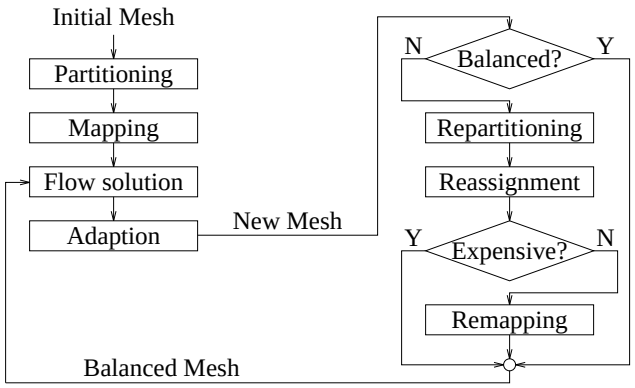

FIG. 1. *Overview of our framework for parallel adaptive flow computation.*

Notice from the framework in Fig. 1 that the computational load is balanced and the runtime communication reduced only for the flow solver but not for the mesh adaptor. This is acceptable since flow solvers are usually several times more expensive. However, the mesh adaption procedure can also be load balanced, if so desired. In any case, it is obvious that mesh adaption, repartitioning, processor assignment, and remapping are critical components of the framework and must be accomplished rapidly and efficiently so as not to cause a significant overhead to the flow computation.

2 Euler Flow Solver

The unstructured-grid CFD solver [7] used for the numerical calculations in this paper is a finite-volume upwind code that solves for the unknowns at the vertices of the mesh and satisfies the integral conservation laws on nonoverlapping polyhedral control volumes surrounding these vertices. Improved accuracy is achieved by using a piecewise linear reconstruction of the solution in each control volume. The Euler equations are written in an inertial frame so that the rotor blade and grid move through stationary air at the specified rotational and translational speeds. Fluxes across each control volume are computed using the relative velocities between the moving grid and the stationary far field. For a rotor in hover, the grid encompasses an appropriate fraction of the rotor azimuth. Periodicity is enforced by forming control volumes that include information from opposite sides of the grid domain. The solution is advanced in time using conventional explicit procedures.

The code uses an edge-based data structure that makes it particularly compatible with our mesh adaption procedure. Furthermore, since the number of edges in a mesh is significantly smaller than the number of faces, cell-vertex edge schemes are inherently more efficient than cell-centered element methods. Finally, an edge-based data structure does not limit the user to a particular type of volume element. Even though tetrahedral elements are used in this paper, any arbitrary combination of polyhedra can be used [3]. This is also true for our global load balancing procedure.

3 Parallel Mesh Adaption

The serial mesh adaption scheme is described in [2]. The 5000-line C code has its data structures based on edges of a tetrahedral mesh. This means that the elements are defined by their edges rather than by their vertices. This feature makes the mesh adaption procedure capable of performing anisotropic refinement and coarsening that results in a more efficient distribution of grid points. Details of the distributed-memory implementation are given in [5]. The parallel version consists of an additional 3000 lines of C++ and MPI code as a wrapper around the original mesh adaption program. An object-oriented approach allowed this to be performed in a clean and efficient manner.

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 are larger (smaller) than a specified upper (lower) threshold are targeted for subdivision (removal). Only three subdivision types are allowed for each element. The 1-to-8 isotropic subdivision is implemented by adding a new vertex at the mid-point of each of the six edges. The 1-to-4 and 1-to-2 subdivisions result either because a tetrahedron is 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.

4 Dynamic Load Balancing

We present a new method to dynamically balance the processor workloads with a global view. Results reported in [6] used a simulated mesh adaption scheme to focus on fundamental load balancing issues. Results reported in [1] did use the actual parallel mesh adaption procedure [5], but lacked solution-based adaption, an efficient mesh repartitioner, and an actual remapper. This paper addresses all of these issues.

Our load-balancing procedure has three novel features. First, it uses a dual graph representation of the initial computational mesh to keep the complexity and connectivity constant during the course of an adaptive computation. Second, a heuristic remapping algorithm assigns partitions to processors so that the redistribution cost is minimized. Finally, accurate metrics estimate and compare the computational gain and the redistribution cost of having a balanced workload after each mesh adaption step.

4.1 Dual Graph of Initial Mesh

The dual graph representation of the initial mesh is one of the key features of this work. The tetrahedral elements of the computational mesh are the vertices of the dual graph. An edge exists between two dual graph vertices if the corresponding elements share a face. A graph partitioning of the dual thus yields an assignment of tetrahedra to processors.

Each dual graph vertex has two weights associated with it. The computational weight, w_{comp}, indicates the workload for the corresponding element. The remapping weight, w_{remap}, indicates the cost of moving the element from one processor to another. The weight w_{comp} is set to the number of leaf elements in the refinement tree because only those elements participate in the flow computation. The weight w_{remap}, however, is set to the total number of elements in the refinement tree because all descendants of the root element must move with it from one partition to another if so required. New grids obtained by adaption are translated to the two weights for every element in the initial mesh. As a result, the repartitioning time depends only on the initial problem size and the number of partitions, but not on the size of the adapted mesh.

4.2 Preliminary Evaluation

The preliminary evaluation step rapidly determines if the dual graph with a new set of w_{comp} should be repartitioned. If projecting the new values on the current partitions indicates that they are adequately load balanced, there is no need to repartition the mesh. In that case, the flow computation continues uninterrupted on the current partitions. If the loads are unbalanced instead, the mesh is repartitioned.

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 mesh is repartitioned if the imbalance factor $W_{\max}/W_{\text{avg}}$ is unacceptable.

4.3 Mesh Repartitioning

If the preliminary evaluation step determines that the dual graph with a new weight distribution is unbalanced, the mesh needs to be repartitioned. Note that repartitioning is always performed on the dual graph with the weights adjusted. A good partitioner should minimize the total execution time by balancing the computational loads and reducing the interprocessor communication time. In addition, the partitioning phase must be performed very rapidly for our load balancing framework to be viable.

Several excellent partitioning algorithms are available [9]; however, we need one that is extremely fast. Multilevel algorithms present a way to reduce the partitioning cost, while maintaining the quality of the partitions. These algorithms reduce the size of the graph by collapsing vertices and edges, applying an eigensolver on the smaller graph, and then uncoarsening it back to construct a partitioning for the original graph. We have used the MeTiS [4] multilevel scheme as the repartitioner for the test cases in this paper.

4.4 Similarity Matrix Construction

New partitions must be mapped to the processors such that the redistribution cost is minimized. We assume that the redistribution cost is proportional to the volume of data moved. In general, the number of new partitions is an integer multiple F of the number of processors. Each processor is then assigned F partitions. The rationale behind allowing multiple partitions per processor is that performing data mapping at a finer granularity reduces the volume of data movement at the expense of partitioning and processor reassignment times. However, setting F to unity suffices for most practical applications.

The first step toward processor reassignment is to compute a similarity measure S that indicates how the remapping weights w_{remap} of the new partitions are distributed over the processors. It is represented as a matrix where entry S_{ij} is the sum of the w_{remap} of all the dual graph vertices in new partition j that already reside on processor i. A similarity matrix for $P = 4$ and $F = 2$ is shown in Fig. 2. Only the non-zero entries are shown.

4.5 Processor Reassignment

In general, each processor cannot be assigned F unique partitions corresponding to their F largest weights. This is the case for the S shown in the left half of Fig. 2 where the F largest weights for each processor are shaded. To minimize the total data movement for all processors, each processor i must be assigned F unique partitions j_{i_f}, $f = 1, 2, \ldots, F$, so that the objective function $\sum_{i=1}^{P} \sum_{f=1}^{F} S_{ij_{i\_f}}$ is maximized subject to the constraint $j_{i\_r} \neq j_{k\_s}$, for all $i \neq k$ and $r, s = 1, 2, \ldots, F$.

This optimization problem reduces to the maximally-weighted bipartite graph partitioning problem [1]. We have implemented both optimal and heuristic solution algorithms.

The psuedo-code for our suboptimal greedy algorithm is given in [1]. Applying the heuristic procedure to the similarity matrix in the left half of Fig. 2 generates the new processor assignment shown in the right half of Fig. 2. Note that this is *not* the optimal solution.

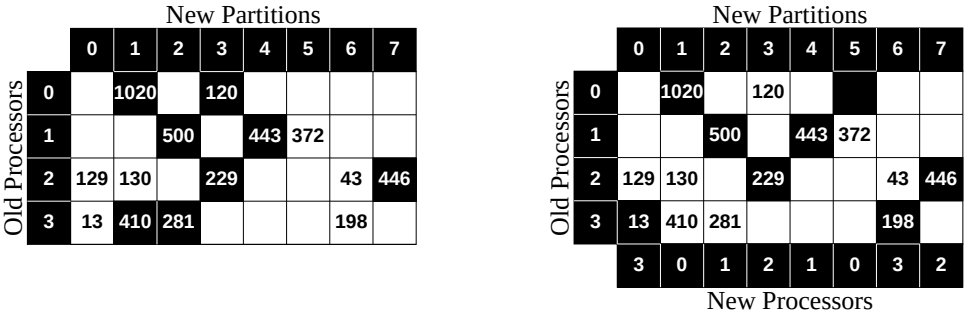

FIG. 2. *A similarity matrix S before (left) and after (right) processor reassignment.*

4.6 Cost Calculation

The computational gain due to 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. It can be expressed as $T_{\text{iter}} N_{\text{adapt}} (W_{\max}^{\text{old}} - W_{\max}^{\text{new}})$, where T_{iter} is the time required to run one solver iteration on one element of the original mesh, N_{adapt} is the number of solver iterations between mesh adaptions, and $W_{\max}^{\text{old}}$ and $W_{\max}^{\text{new}}$ are the sum of the w_{comp} on the most heavily-loaded processor for the old and new partitionings, respectively.

The redistribution cost is calculated from similarity matrix using the remote-memory latency time T_{lat} and the message setup time T_{setup}. It can be expressed as $CMT_{\text{lat}} + NT_{\text{setup}}$, where M is the storage requirements per element for the solver and mesh adaptor, and C and N are the total number of elements and sets of elements to be moved, respectively. For the S in Fig. 2, $C = 1485$ and $N = 6$. The new partitioning and processor assignment are accepted if the computational gain is larger than the redistribution cost.

4.7 Data Remapping

The remapping phase is responsible for physically moving the data when it is reassigned to a different processor. When an element is moved from one processor to another, a communication cost as well as an overhead are incurred. The communication cost includes the time required to pack and unpack the send and receive buffers, and the message setup and remote-memory latency times. The overhead is the time necessary to rebuild the internal and shared data structures in a consistent manner. Note that the relationship between the number of elements moved and the total data volume is not exactly linear. This is due to the movement of the shared data structures whose size is a function of the locations of the old and new partition boundaries. The shared information accounts for a small percentage of the data volume, and is the cause of slight perturbations.

5 Results

The parallel mesh adaption and global load balancing procedures have been implemented in MPI on an IBM SP2. The computational mesh, containing 60,968 tetrahedral elements and 78,343 edges, is the one used to simulate an acoustics experiment where a model UH-1H helicopter rotor blade was tested over a range of subsonic and transonic hover-tip Mach numbers. Numerical results and a detailed report of the simulation are given in [8].

Results are presented for one refinement and one coarsening step using five different edge-marking strategies representing significantly different scenarios. Strategies `Local_1` and `Local_2` targeted 5% and 33% of the edges for refinement in a single compact region of the mesh. The final meshes consisted of about 82,300 and 201,800 elements respectively. Strategies `Real_1` and `Real_2` targeted edges for adaption based on an error indicator [8] calculated from the flow solution such that the mesh sizes were approximately equal to those obtained in the corresponding `Local` cases. The subsequent coarsening phase for `Local_1` and `Real_1` undid all of the refinement to restore the initial mesh. For `Local_2` and `Real_2`, the refined mesh was coarsened down to about 100,200 elements. The fifth strategy, called `Random_2`, consisted of randomly targeting edges for adaption such that the mesh sizes after both refinement and coarsening were approximately equal to those obtained with `Local_2`.

Figure 3 illustrates the parallel speedup of the refinement and coarsening phases of the mesh adaption code for the five edge-marking strategies. As expected, `Random_2` gives the best speedup performance as the processor workloads are inherently balanced. The speedup results are the worst for the `Local_1` case because a single compact region of the mesh is adapted. All the work is thus performed by only a small subset of the available processors. The `Local_2` and `Real_2` speedup values are higher than the corresponding `Local_1` and `Real_1` values because the adaption region is much larger. The coarsening results are similar to those for the refinement step because of the algorithmic similarities of the two methods. However, performance improves significantly for the `Local_1` and `Real_1` cases because undoing all of the previous refinement better balances the processor workloads. Extensive performance analysis of the parallel mesh adaption code is given in [5].

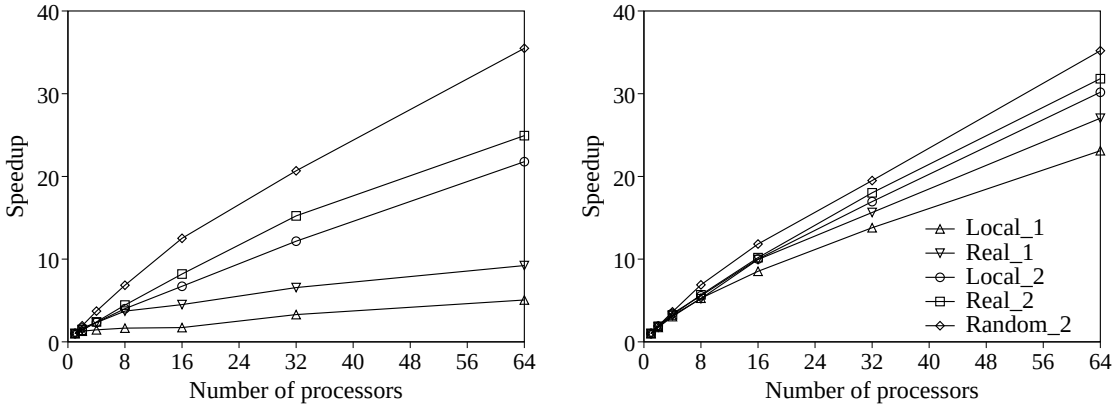

FIG. 3. *Speedup of the parallel mesh adaptor during refinement (left) and coarsening (right).*

Figure 4 compares the execution times and the amount of data movement for the `Local_2` refinement strategy when using the optimal and heuristic processor assignment algorithms. Four pairs of curves are shown in each plot for $F = 1, 2, 4,$ and 8. The optimal method always requires almost two orders of magnitude more time than our heuristic method. The execution times also increase significantly as F is increased. This is because the size of the similarity matrix grows with F. However, the volume of data movement decreases with increasing F. This confirms our earlier claim that data movement can be reduced by mapping at a finer granularity. The relative reduction in data movement, however, is not very significant for our test cases. The results in Fig. 4 illustrate that our heuristic mapper is almost as good as the optimal algorithm while requiring significantly less time. Similar results were obtained for the other edge-marking strategies.

Figure 5 shows how the execution time is spent during the refinement and the subsequent

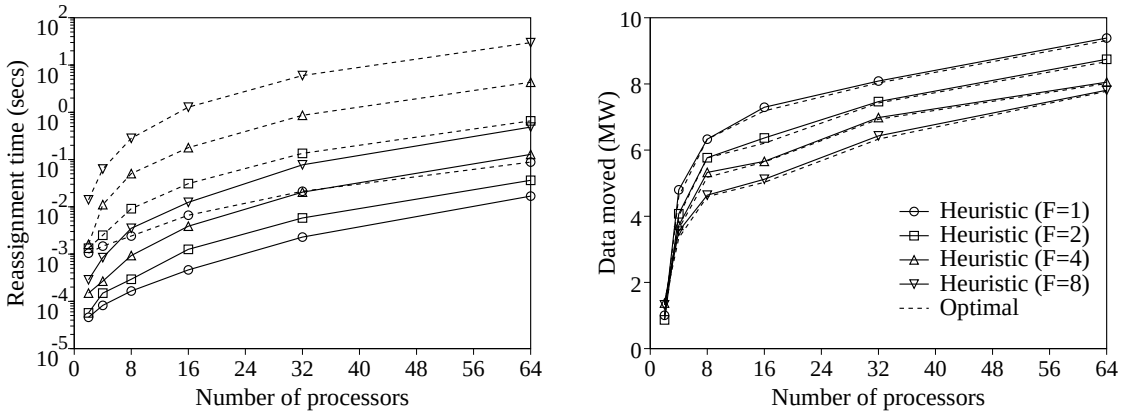

FIG. 4. *Comparison of the optimal and heuristic mappers in terms of execution time (left) and volume of data movement (right) for the* `Local_2` *refinement strategy.*

load-balancing phases of the `Real_1` and `Real_2` strategies with $F = 1$. The processor reassignment times are not shown since they are negligible compared to the other times and are similar to those shown in Fig. 4. The repartitioning curves are almost identical because the repartitioning time mostly depends on the initial problem size and the number of partitions. The serial version of MeTiS [4] is used in this work; significant performance improvement is therefore expected with a parallel repartitioner. The remapping times gradually decrease as the number of processors is increased. This is because even though the total volume of data movement increases with the number of processors, there are actually more processors to share the work. Notice that data remapping is still the most dominant cost and about three times more expensive than mesh adaption. The results indicate that our load balancing strategy will remain viable on large numbers of processors; however, significant future effort needs to be focused on data remapping.

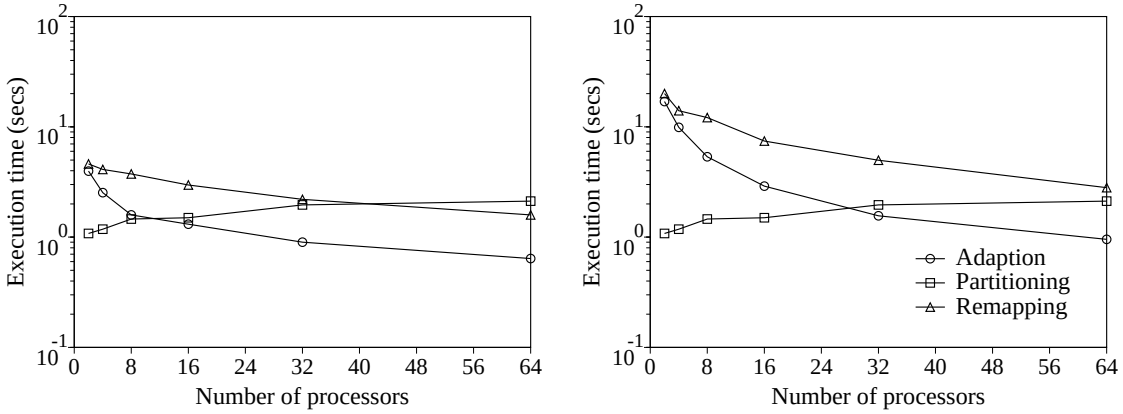

FIG. 5. *Anatomy of execution times for* `Real_1` *(left) and* `Real_2` *(right) refinement strategies.*

Finally, we investigate the impact of load balancing on flow solver execution times. Suppose that there are P processors and N/P elements per processor in a load-balanced configuration. The computational mesh is then refined to generate a total of GN elements, $1 \leq G \leq 8$ for our refinement procedure. If the workload were balanced, each processor would have GN/P elements. But in the worst case, all the elements on a subset of processors are isotropically refined 1-to-8, while elements on the remaining processors remain unchanged. Thus, the maximum improvement due to load balancing for a *single*

refinement step would be $\frac{1}{G}$ min $(8, P(G-1)+1)$.

Figure 6 illustrates the impact of load balancing for the five different refinement cases. The mesh growth factor G is 1.35 for the `Local_1` and `Real_1` cases, giving a maximum improvement of 5.93 with load balancing when $P \geq 20$. The value of G is 3.31 for the other three cases, so the maximum improvement is 2.42 when $P \geq 4$. Note that the curves for the `Local` cases come closest to the `Max_Imb` (maximum imbalance) curves because a single compact region of the mesh was refined to cause severe imbalance among the processors. The impact of load balancing for the `Real` cases are somewhat less significant since they model actual solution-based adaptions that does not necessarily cause worst case scenarios. The `Random_2` case gives only a marginal improvement with load balancing because the computational work is already distributed quite uniformly among the processors. It is important to realize that the results shown in Fig. 6 are for a single refinement step. With repeated refinement, the gains realized with load balancing may be even more significant.

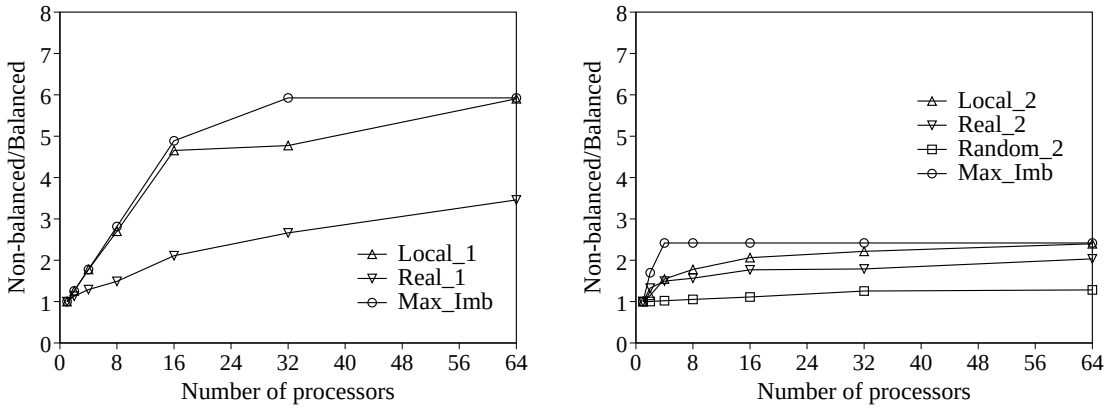

FIG. 6. *Impact of load balancing on flow solver execution times for mesh growth factor G of 1.35 (left) and 3.31 (right).*

References

[1] R. Biswas, L. Oliker, and A. Sohn, *Global load balancing with parallel mesh adaption on distributed-memory systems*, Supercomputing '96, Pittsburgh, PA, 1996, `http://www.supercomp.org/sc96/proceedings/SC96PROC/BISWAS/INDEX.HTM`.

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

[3] ——, *A dynamic mesh adaption procedure for unstructured hexahedral grids*, Paper 96-0027, 34th AIAA Aerospace Sciences Meeting, Reno, NV, 1996.

[4] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, Report 95-035, University of Minnesota, Minnesota, MN, 1995.

[5] L. Oliker, R. Biswas, and R. C. Strawn, *Parallel implementation of an adaptive scheme for 3D unstructured grids on the SP2*, in Parallel Algorithms for Irregularly Structured Problems, LNCS 1117, Springer-Verlag, 1996, pp. 35–47.

[6] A. Sohn, R. Biswas, and H. D. Simon, *Impact of load balancing on unstructured adaptive grid computations for distributed-memory multiprocessors*, 8th IEEE Symposium on Parallel and Distributed Processing, New Orleans, LA, 1996, pp. 26–33.

[7] R. C. Strawn and T. J. Barth, *A finite-volume Euler solver for computing rotary-wing aerodynamics on unstructured meshes*, J. AHS, 38 (1993) pp. 61–67.

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

[9] R. Van Driessche and D. Roose, *Load balancing computational fluid dynamics calculations on unstructured grids*, Report R-807, AGARD, 1995.