JOVE: A Dynamic Load Balancing Framework for Adaptive Computations on an SP-2 Distributed-Memory Multiprocessor

Andrew Sohn[1] and Horst Simon[2]

NJIT CIS Technical Report 94-60

September 23, 1994

Abstract

Computational requirements of full scale computational fluid dynamics change as computation progresses on a parallel machine. The change in computational intensity causes workload imbalance of processors, which in turn requires a large amount of data movement at runtime. If parallel CFD is to be successful on a parallel or massively parallel machine, balancing of the runtime load is indispensable. Here a framework is presented for dynamic load balancing for CFD applications, called Jove. One processor is designated as a decision maker Jove while others are assigned to computational fluid dynamics. Processors running CFD send flags to Jove in a predetermined number of iterations to initiate load balancing. Jove starts working on load balancing while other processors continue working with the current data and load distribution. Jove goes through several steps to decide if the new data should be taken, including preliminary evaluate, partition, processor reassignment, cost evaluation, and decision. Jove running on a single IBM SP2 node has been completely implemented. Preliminary experimental results show that the Jove approach to dynamic load balancing can be effective for full scale grid partitioning on the target machine IBM SP2.

---

1. Dept. of Computer and Information Science, New Jersey Institute of Technology, Newark, New Jersey 07102-1982, sohn@cis.njit.edu. This work was supported in part by the NASA/University Joint Venture in Research Program.
2. NAS Applied Research Branch, Mail Stop T27A-1, NASA Ames Research Center, Moffett Field, CA 94035-1000, simon@nas.nasa.gov. The author is an employee of Computer Sciences Corporation. This work is supported through NASA contract NAS2-12961.

1 Introduction

The computation of the airflow over a complex aircraft in three dimensions is a challenging task. It is particularly challenging, when unstructured grids and finite element or finite volume methods are going to used [2]. The computational requirements for such a problem can only be satisfied by massively parallel machines [6,10]. During a typical adaptive, unsteady CFD calculation, these type of unstructured meshes will have to be locally refined, and derefined, depending on the characteristic of the calculation, for example in order to capture a moving shock. The computational intensity over regions changes in time and in some regions near to the body of the aircraft it is typically much higher than far away from the body.

A parallel implementation of these finite element or finite volume methods on parallel machines typically entails two steps [12]: First, as a preprocessing step the computational mesh (or grid) is partitioned into smaller meshes. Second, the partitioned submeshes are then mapped to processors based on a mapping strategy. While this static partitioning and mapping approach would be appropriate for steady CFD calculations which do not change computational intensity during the iteration, this static approach is simply inefficient for unsteady, adaptive calculations. As the computation progresses on all processors simultaneously, the CFD problem running on a multiprocessor changes its computational behavior. Some processors will have a lot of work to do while some little, or often it is possible that some have no work to do. The work processors perform is not balanced and the initial partition is no longer efficient due to changes in the computation. It is therefore indispensable that the amount of work each processor performs be balanced at runtime to increase the processor utilization and performance [3].

Balancing runtime computational load, however, is often very difficult due to many practical issues such as measurement of computational load, decision of workload distribution, runtime data movement, minimization of communication, etc. It is precisely the purpose of this report to establish a *framework* of runtime load balancing for CFD-related applications. We investigate how the amount of work each processor executes can be balanced at runtime. The dynamic load balancer *Jove* addresses load balancing along computational intensity for workload distribution, and communication intensity for data distribution. A processor designated as Jove monitors computational load while all other processors execute a CFD application. Changes in computational behavior are monitored by Jove in every predetermined number of iterations. The Jove processor then evaluates if moving data from processors to processors will be beneficial. If data movement is costly compared to computational gain, no data movement will take place and the CFD continues without interruption. Should Jove decide that the new data set has intolerable change in computational intensity, it will instruct processors to move data according to the Jove's determination.

We begin our discussion in section 2 by giving an overview of a dynamic load balancer, Jove. Section 3 presents preliminary evaluate step of the Jove. Section 4 explains partitioning of grids using a modified spectral method. Section 5 is the key to our approach, which explains how sets and processors are reassigned such that the data movement be minimized and load be balanced. Section 6 evaluates the new data set and decides if the new data should be taken. Section 7 lists implementation details, followed by the analysis on preliminary experimental results. The last section summarizes our study and briefly gives future plans on our ongoing dynamic load balancing for grand challenge problems.

2 Dynamic Load Balancing

A typical structural analysis or computational fluid dynamics problem is represented as a grid of vertices V and edges E. A vertex representing a finite element has computational intensity associated with it, often called *weight*, or simply

W. Figure 1 shows a very small scale airfoil grid G consisting of 460 vertices and 1303 edges. The darkness of vertices represent computational intensity (weight). One can think of an edge connecting two vertices as representing the existence of a computational relation between the two finite elements. A parallel implementation of CFD on multiprocessors requires the grid of vertices and edges to be divided into smaller grids, each of which is assigned to a processor. The degree of connectivity and computational weights of vertices determine how adjacent vertices should be grouped to form smaller grids (or meshes). The division of vertices of variable computational weights must be carefully done such that the resulting subgroups must have an equal or nearly equal amount of computational intensity. Inappropriate division (or partition) of a grid often results in heavy imbalance of computational intensity since some processors have a lot of work to perform while others don't, or often some none at all.

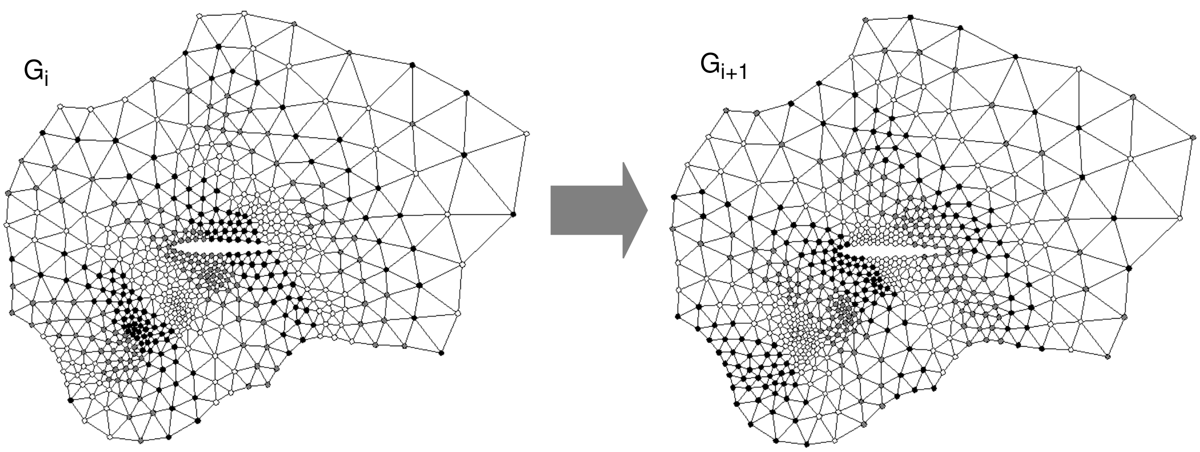

Figure 1: Two successive snap shots of an airfoil grid with 460 vertices and 1303 edges. Darkness of vertices represent computational intensity, or simply weight. The types of weights are made to the simplest for illustration purpose. Real CFD applications require tens to thousands of different weights.

What makes parallel CFD difficult is change in weight (computational intensity). As computation progresses, the weight associated with vertices changes in time due to different computational requirements. Figure 1 shows two grids, G_i and G_{i+1}, separated by an iteration. A grid G_i with W_i changes to a new grid G_{i+1} with W_{i+1} after an iteration. We expect that the partitioning and assignment of subgrids to processors at the beginning of the computation will be only effective for a short period of time. After a few iterations of the CFD computation, the majority of vertices will have their weights changed. The computational load on each processor will be different, resulting in a load imbalance. In Figure 1, note that the small filled circles at the lower left corner of G_i have moved toward the upper right corner of G_{i+1} after one iteration. The two snapshots are simplified here in order to demonstrate the change in computational intensity. Only four different types of weights are illustrated, including hollow circle for the lowest intensity, filled circle for the highest intensity, and two different shaded circles for intermediate intensity. In a real CFD application the range of weights will be much larger. Also the change of weights will follow a different pattern as indicated here.

As the two snap shots shown above suggest, it is extremely important to dynamically *repartition* the grid according to new computational weights. It is however not straightforward to repartition the grid. Partitioning new weights into new smaller grids must be quick such that newly partitioned grid can be evaluated in about the same time while the CFD computation continues working on the next iteration. Evaluation and decision steps must be able take communication loss and computational gain into account in deciding whether the newly partitioned grids will give the best possible balanced load to all processors. The approach presented in this report is centered around the two premises: First, the partitioning of newly generated grid is quick compared to CFD computation time. Second, the decision of accepting newly partitioned grids is based entirely on the evaluation of computational gain and communication loss. Figure 2 gives an overview of our approach to dynamic load balancing as implemented in Jove.

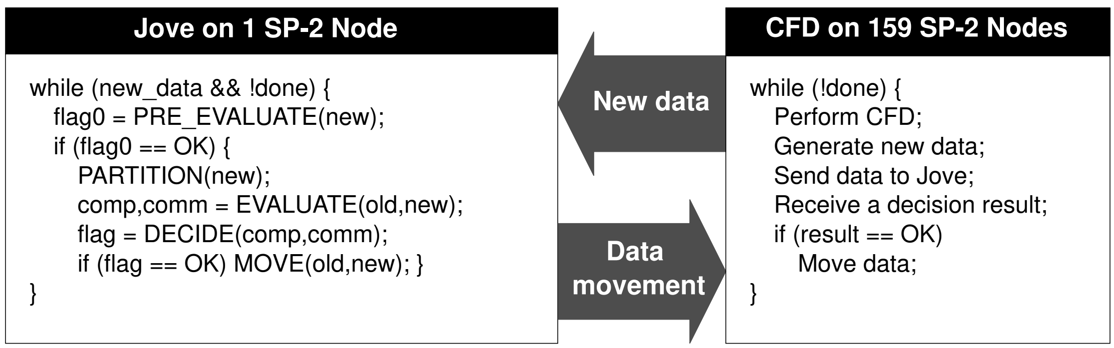

Figure 2: The organization of JOVE.

The entire system consists of two modules: the decision maker Jove running on an SP-2 processor and the CFD module running on 159 SP-2 processors.(We explain Jove here in the context of the 160 node IBM SP-2 at NASA Ames, however the discussion equally applies to any other MIMD message passing computer). Jove executes the while loop shown on the left hand side of Figure 2 while the CFD application executes the one on the right hand side. Details of the CFD application are beyond the scope of this report (see [2,6] for current work on unstructured CFD). The first step of Jove is *pre-evaluate*, which decides if the new weight information generated by the CFD application should be taken into account for further consideration. The objective behind this preliminary evaluation is to find if the newly generated weights have changed so much that repartitioning of the grid should be considered. If the preliminary evaluation determines that the new weights have indeed changed considerably, it will recommend to repartition the grid. The basic criterion for the decision on repartitioning is to use the imbalance factor derived from the projection of new data to old partition.

The partition step divides the grid into new subgrids, expressed by sets of vertices. This step uses a modification of the successful Recursive Spectral Bisection (RSB) algorithm [1,9], which uses an inertial approach to rapidly update a partition from one grid to the next one. This new algorithm is called Dynamic Spectral Bisection and will be explained in detail in [11].

Given now that vertices are partitioned into n sets, Jove continues with the evaluation step, which consists of assignment and cost computation. The assignment step includes processor assignment and set assignment. Processor assignment ensures that each processor receives the same number of partitions. Set assignment ensures that each set has a unique label assigned to it. No two sets will, therefore, have the same set number. In this general framework we assume that it will be possible to map more than one set of vertices to a processor. This is currently not feasible on the IBM SP-2. So for most practical applications the simpler scheme of mapping one partition per processor will suffice.

The cost computation finds computational gain and communication loss for the newly generated partitions. These two values are computed based on the processor assignment and set assignment. Decision step simply decides if the new weights should be taken. If the computational gain is larger than communication loss, the decision will be to use the newly generated partition since it will be advantageous to do so. Otherwise, the newly generated partition is discarded and Jove waits for the next set of weights. Should the decision step accept the new partition, vertex and associated data movements will take place according to the set and processor assignments.

3 Preliminary-evaluate

The main objective of pre-evaluate is to make a quick decision based on a new set of weight data W_{i+1}. If W_{i+1} indicates that the current partition is adequately load balanced, then there is no reason to consider repartitioning, evaluating, and data movement. In this case Jove will simply return, and indicate to the CFD application to continue with the current partition. The main idea of pre-evaluate is to project a new set of weights W_{i+1} to the old sets of vertices S_{old}. Consider for example the grid PARC (Xerox Palo Alto Research Center) which consists of 1240 vertices and 3355 edges. Figure 3 shows two snapshots of PARC, G_i and G_{i+1}. They contain only *four* different types of weights to illustrate the runtime change in computational intensity. Suppose that a grid is partitioned into *eight* sets, which are assigned to *four* processors, resulting in two sets per processor. Table 1 lists various data on sets and weights for the two snapshots.

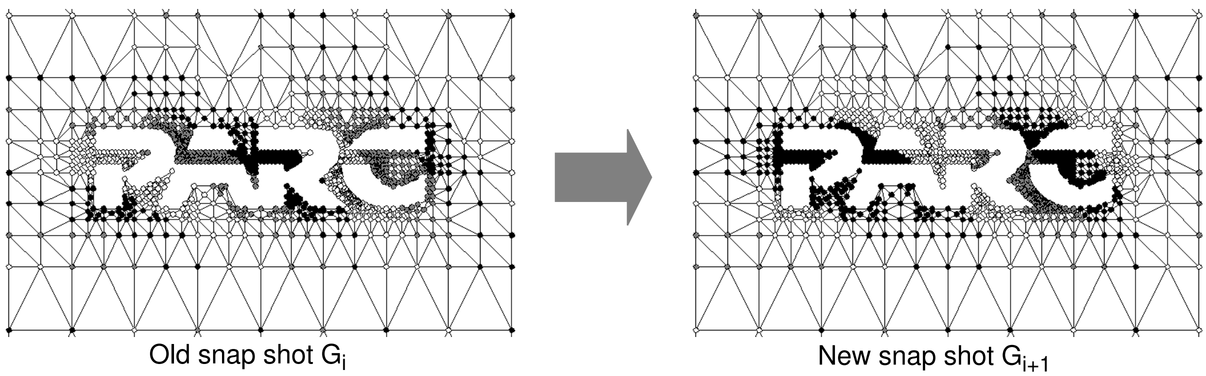

Figure 3: Two successive snap shots of a grid PARC with 1240 vertices and 3355 edges. Some parts of the grid are omitted from the figure since they show little change in the weights.

Set number	0	1	2	3	4	5	6	7	Avg
Old weight projected on *old* sets	1220	1230	1222	1232	1216	1226	1220	1246	1226
New weight projected on *old* sets	765	1041	**563**	693	959	**1534**	1075	**1526**	1019
Imbalance factor (w_i/w_{avg})	0.8	1.0	0.6	0.7	0.9	1.5	1.1	1.5	
Processor number	0	1	2	3	0	1	2	3	

Table 1: Preliminary evaluate on two snap shots of PARC grid.

The first row of Table 1 shows set numbers and the last row indicates processor numbers assigned to sets of vertices. The second row shows the *old weights* projected onto *old sets* S_{old}. Since they were previously partitioned in order to balance computational and communication requirements, the old weight W_i is completely balanced within 1% deviation. The next row shows the effect of the new weights, which are assumed to be produced by an adaptive CFD application. In this example a large number of vertices now has different weights due to computational changes such as refinement of the corresponding grid cell. The third row shows the new weights W_{i+1} projected to old sets S_{old}. The figures show that most of the old sets are affected by the new weights. The fourth row precisely explains the resulting load imbalance. Here we define the as the ratio of weight to average weight. We find from the fourth row that six out of eight sets are experiencing at least 20% or more change in computational intensity while only two sets were not affected or affected little. Sets 2, 5, and 7 are experiencing approximately *half* of their weight change.Thus the preliminary evaluation indicates that repartitioning might be necessary.

More precisely, given the imbalance factors, the preliminary evaluation step now decides if based on the new weights W_{i+1} the grid should be repartitioned, by applying the following condition:

$$nbp > tnp/2 \text{ and } r < \varepsilon$$

where nbp = the number of bad partitions with $w_i/w_{avg} < 0.5$ and $w_i/w_{avg} > 1.5$,

tnp = the total number of partitions,

r (ratio) = w_{max} / w_{min},

ε (threshold error) = 2.

The above rule accepts the running example shown in Table 1 for partitioning and evaluation since $nbp < tnp/2$, where $nbp = 3$ and $tnp/2=4$.

4 Partition

Preliminary evaluation determines if a new grid G_{i+1} with a new weight W_{i+1} should be considered for *re-partitioning*. The partition step divides the new grid into a desired number of sets of vertices. While there are a number of partitioning algorithms, we use here a new algorithm which combines the high quality of spectral algorithms [1,9] with an update algorithm. The update algorithm is based on a center of inertia based method for speed (for details see [11]). It attempts to utilize information from the initial spectral partitioning. The actual details of this partitioning algorithm are not important for the understanding of the framework. Essentially any partitioning algorithm could be used here, as long as it is fast, and delivers a reasonably balanced partition based on the new weights. The main objective behind the above inertial strategy is to perform quick repartitioning such that the runtime decision of accepting/rejecting a new grid can be minimized.

5 Processors and Set Reassignment

The evaluation step consists of three steps: (1) similarity metric construction, (2) processor and set reassignment, and (3) cost computation. Two similarity metrics are considered in this study: vertex similarity $\boldsymbol{A}$ and weight similarity $\boldsymbol{B}$. Processor reassignment ensures each processor to receive an equal number of sets while set reassignment ensures a set to get a unique set number. Cost computation finds computational gain and communication loss for a new grid.

The evaluate step starts by computing two similarity measures. Vertex similarity $\boldsymbol{A}$ indicates how the vertices of new sets are distributed over old sets while weight similarity $\boldsymbol{B}$ explains how the weights of new sets are distributed over old sets. Similarity metrics are constructed by projecting the new weights on the old sets. Consider below two similarity metrics constructed from two snapshots of the PARC grid. We continue to use the same example with eight sets assigned to four processors, i.e., each processor has two sets assigned to it.

(a)

Old set \ New set (Set#)	0	1	2	3	4	5	6	7	PE#
0	**122**	75		30					0
1	3		94	**110**					1
2	**96**	48			57		6		2
3	44	16		43	38	11	**58**		3
4					1		**62**	53	0
5								**88**	1
6						**90**			2
7					**52**	31	12		3
PE#	0	1	2	3	0	1	2	3	

(b)

Old set \ New set (Set#)	0	1	2	3	4	5	6	7	PE#
0	389	**510**		120					0
1	11		**563**	444					1
2	236	**401**			333		47		2
3	129	130		129	146	43	**446**		3
4					13		490	**502**	0
5								**1024**	1
6						**1020**			2
7					467	**471**	92		3
PE#	0	1	2	3	0	1	2	3	

Figure 4: Similarity metric: (a) node similarity $\boldsymbol{A}$, (b) weight similarity $\boldsymbol{B}$.

Figure 5(a) shows a vertex similarity $\boldsymbol{A}$ and (b) shows a weight similarity $\boldsymbol{B}$ between S_{old} and S_{new}. The first row of Figure 5(a) indicates that old set 0, $S_{0,\text{old}}$, is distributed over three new sets, $S_{0,\text{new}}$, $S_{1,\text{new}}$, and $S_{3,\text{new}}$, after partitioning. To be more precise, 227 vertices of $S_{0,\text{old}}$ are now distributed to $S_{0,\text{new}}$, $S_{1,\text{new}}$, and $S_{3,\text{new}}$ respectively with 122 vertices, 75 vertices, and 30 vertices. The upper left element of vertex similarity, $A_{0,0}$=122, indicates that $S_{0,\text{old}}$ and $S_{0,\text{new}}$ have 122 vertices in common after partitioning. A new set with the largest vertices of an old set is called a *dominant* set. Those shaded squares indicate dominant new sets. $S_{0,\text{new}}$ is the dominant set of $S_{0,\text{old}}$, $S_{3,\text{new}}$ is the dominant set of $S_{1,\text{old}}$, etc. It is these dominant sets which will help decide how data to be moved. A similar argument applies to weights. The weights of old sets are again distributed over new sets after partition. Shaded squares of Figure 5(b) again represent dominant sets.

Comparing the dominant sets of vertex similarity and weight similarity, we find that two dominant sets do not agree with each other. For example, $\boldsymbol{A}_{0,0}$ of vertex similarity indicates that $S_{0,\text{old}}$ will likely become $S_{0,\text{new}}$ since it is a dominant one. However, $\boldsymbol{B}_{0,1}$ of weight similarity indicates otherwise. The disagreement between the two metric precisely indicates that yet another metric is needed to determine how data movement should take place. The third metric, called *absolute similarity* $\boldsymbol{C}$, is therefore constructed, which combines both vertex similarity and weight similarity based on the following rule:

$$\boldsymbol{C} = \boldsymbol{A} * cpn + \boldsymbol{B} * cpw$$

where *cpn* is a number of clocks per node (vertex) and *cpw* is a number of clocks per weight, i.e., computational intensity. The two coefficients, *cpn* and *cpw*, are determined by an application program and machine architecture. Using $cpn = cpw = 1$, we find an absolute similarity $\boldsymbol{C}$ shown in below.

Set#	0	1	2	3	4	5	6	7	
0	511	**585**		150					0
1		657	**554**						1
2	332	**449**			390		53		2
3	173	146		172	184	54	**504**		3
4							552	**555**	0
5								**1112**	1
6						**1110**			2
7					**519**	502	104		3
	0	1	2	3	0	1	2	3	PE#

Figure 5: Absolute similarity $\boldsymbol{C} = \boldsymbol{A}*cpn + \boldsymbol{B}*cpw$.

Shaded squares again indicate dominant sets. Absolute similarity now indicates that old S_0 becomes S_1, old S_1 becomes new S_2, etc. It should be noted that the absolute similarity $\boldsymbol{C}$ matches neither the vertex similarity $\boldsymbol{A}$, nor the weight similarity $\boldsymbol{B}$. This is exactly why the two similarity metrics are merged into a single homogenous metric which is to be used for evaluation and decision. The absolute similarity, however, has two serious drawbacks. A close inspection of the absolute similarity indicates that first, processors do not have an equal number of dominant sets assigned. Second, more than one set is assigned to the same set number.

Figure 6 shows that there is only one dominant set assigned to processor 0 (P_0), while there are three dominant sets assigned to P_1. This is against the purpose of this report, load balancing; processors with different number of dominant sets will certainly cause load imbalance. The dominant sets of absolute similarity, therefore, need to be rearranged so that each processor will have an equal number of dominant sets. Processor reassignment, however, is not straightforward since it itself constitutes another optimization problem. We certainly do not want to solve another optimization problem to solve an optimization problem. Our approach to processor reassignment is therefore designed to find a suboptimal solution due to runtime constraints. The following algorithm reassigns dominant sets to processors.

```
for (i=0; i<npe; i++) {                      /* npe = number of processors */
      cnt = nds[i] – nppp;                   /* nppp = number of partitions per processor */
      for (j=0; j<cnt; j++) {                /* nds[i] = number of dominant sets for Pi */
1.          Find from a processor Pi a minimum dominant set. /* Pi = Processor */
2.          Mark it as a regular non-dominant set of Pi.
3.          Find a processor Pk which has less than nppp dominant sets.
4.          Find a maximum non-dominant set from Pk.
5.          Mark it as a dominant set for Pk.
      }
}
```

Figure 6: A simplified version of *processor* reassignment.

Applying the above reassignment algorithm to the absolute similarity $\boldsymbol{C}$ gives a new absolute similarity $\boldsymbol{C}'$ shown in Figure 8(a).

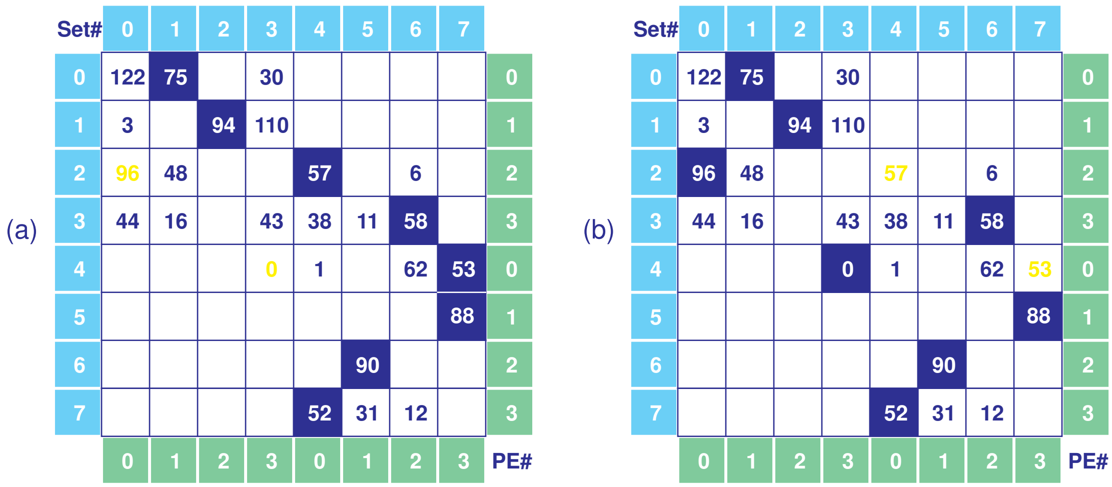

Figure 7: Absolute similarity. (a) $\boldsymbol{C}'$ after processor reassignment, (b) $\boldsymbol{C}''$ after set reassignment.

Processor 0 had only one dominant set before reassignment. After dominant sets are reassigned, P_0 now has two dominant sets, including the one taken from P_1. All processors now have *two* dominant sets. The modified absolute similarity $\boldsymbol{C}'$, however, is still problematic. A close inspection of Figure 8(a) reveals that no set is assigned to S_0 while two sets are assigned to S_7. This uneven assignment requires yet another reassignment. A set reassignment algorithm listed below ensures that all sets each receive a unique set number:

```
for (i=0; i<npart; i++) {              /* npart = number of partitions */
    cnt = nds[i] – 1;                  /* nds[i] = number of dominant sets for column i */
    for (j=0; j<cnt; j++) {
1.      Find from column i a dominant set C_k,i which has the second maximum value.
2.      Mark C_k,i as a regular non-dominant set.
3.      Find from row k a maximum non-dominant set C_k,l, such that (i mod npe) ≠ (l mod npe)
4.      Mark C_k,l as a dominant set for column i.
    }
}
```

Figure 8: A simplified version of *set* reassignment.

Applying the set reassignment algorithm above to the absolute similarity $\boldsymbol{C}'$ gives a new absolute similarity $\boldsymbol{C}''$ shown in Figure 7(b). Now, all four processors have two sets assigned and each set has a unique set number. The reassignment step is now complete. The reassignment algorithms shown above are certainly not the best algorithms which would yield an optimal solution. While the reassignment can be formulated as a search problem to find an optimal solution, this effort is probably not necessary. Our main objective is to quickly reassign sets and processors in every iteration such that the evaluation and decision steps can proceed while an application CFD program is in progress. Using the modified absolute similarity $\boldsymbol{C''}$ of Figure 8(b), we now evaluate computational gain and communication loss of the new assignment.

6 Evaluate and Decision

The evaluation step computes two values, computational *gain* and communication *loss*. The two values are computed based on an absolute similarity. Computational gain can be easily obtained by simply overlapping an absolute similarity $\boldsymbol{C}$ and a weight similarity $\boldsymbol{B}$. Similarly, communication loss is obtained by simply overlapping an absolute similarity $\boldsymbol{C}$ and a vertex similarity $\boldsymbol{A}$. Figure 10(a) shows the result of overlapping an absolute similarity and a weight similarity. Each row now has two dominant sets, indicating that they belong to the same processor. A sum of non-dominant entries (not shaded squares) in each row gives a computational gain, which is also tabulated in the last column.

Communication loss requires a little explanation as it involves more than just summation. Two parameters are considered for communication cost, remote memory latency and message setup time. Remote memory latency refers to memory-to-memory copying time for a word (or integer unit) while message setup time refers to preparing message heading, copying to message buffer, etc. Latency applies to *every* vertex while setup time applies only to *each* set. Using the machine dependent parameters of latency=40µsec and setup time=50µsec (which are estimates for the IBM SP-2 and need to be refined further), we find computational gain and communication loss, as shown in Figure 10.

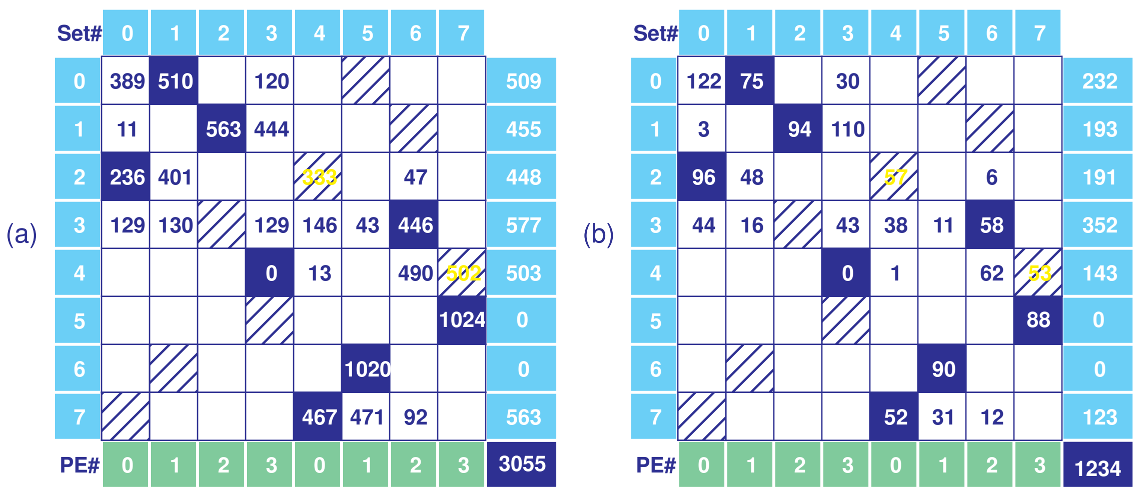

Figure 9: Cost metric. (a) computational gain, (b) communication loss.

A final decision on accepting/rejecting a newly partitioned and assigned grid is made by comparing computational gain and communication loss. If the computational gain is greater than the communication loss, Jove accepts the new data set. This comparison can be controlled by introducing a coefficient such that *gain* > α**loss*, where α can be determined based on experiments and problem characteristics. For the running example shown in Figure 10, the new data is accepted because computational gain of 3055 is greater than communication loss of 1234.

7 Implementation Details and Preliminary Results

The foregoing approach has been implemented on an SP-2 multiprocessor installed at NASA Ames Research Center. Jove, written in C with approximately 3000 lines of C code, is running on a single SP2 processor. The implementations of CFD application programs are currently being completed and are expected to be available in the immediate future to interface to Jove. We list below various assumptions and parameters used in the experiments:

- Number of partitions > number of processors
- Number of partitions per processor, *nppp*   1
- All processors have the same number of partitions.
- Time to generate a new grid by CFD application > Time to decide by Jove
- Processor to processor logical distance is the same.
- Setup time = 50 μsec
- CPN: clocks per vertex communication (remote memory latency)
- CPW: clocks per weight computation
- Cost/Vertex = setup time + CPN

The first four items are fundamental assumptions and are machine *independent*. The last five items are machine *dependent* and should be modified for different machines.

Two unstructured grid problems are used for experiments. The first grid, PARC, has 1240 vertices and 3355 edges while the second grid, STRUT, has 14504 vertices and 57387 edges. All the snap shots (intermediate grids, and their weights) are prepared as files which are read by Jove for the decision process. Figure 11 illustrates a typical behavior of Jove on an SP-2 processor for the two grids. The *x*-axis plots iteration numbers while the *y*-axis shows execution time for each iteration. Note that the *y*-axis is plotted with a logarithmic scale and that the two plots have different scales.

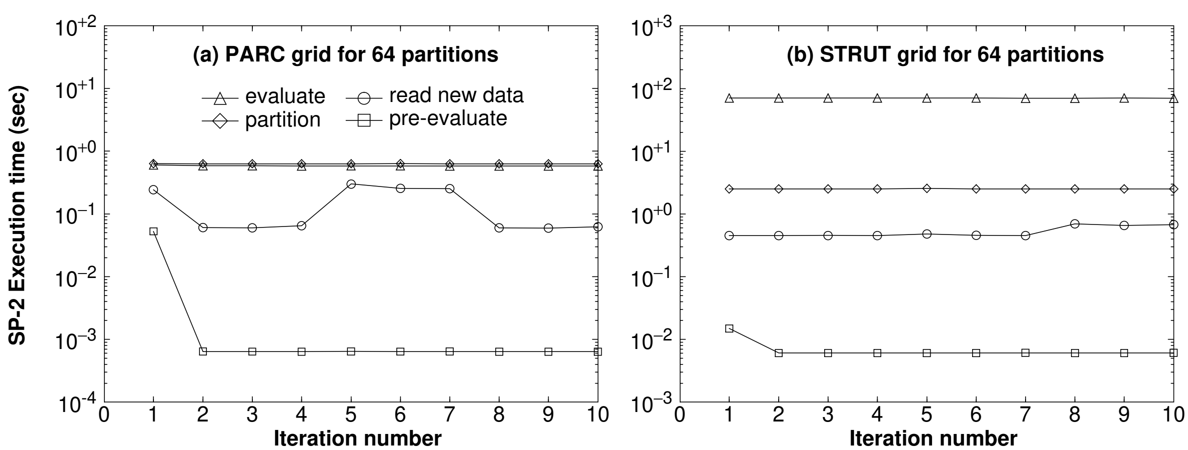

Figure 10: Execution time (sec) in iterations on an SP-2 processor.

Figure 11(a) shows that partition and evaluate time are large and dominate the whole process. Pre-evaluate step is essentially negligible as it is a fraction of evaluate time. The read time fluctuates due to accessing files on disk (file access will be replaced by receiving data via message passing from the CFD applications after the completion of their implementation). When the number of vertices increases, the evaluate time starts dominating the whole process. Figure 11(b) shows that the evaluate time entirely dominates the whole process. We stated earlier in section 3 that we wish to quickly partition a at runtime. The objective is certainly achieved as Figure 11(b) indicates. However, the results also indicate that evaluate time is too large compared to the partition time. This clearly indicates that finding an optimal processor assignment will be difficult because the whole process will depend on the evaluation step.

The two plots show that the execution time of individual steps do not drastically change over iterations. In fact, they remain almost constant, which is an encouraging news since an actual CFD would take hundreds or thousands of iterations before termination. The two plots also reflect the relation between the number of vertices and execution time. We find that the execution time for STRUT is an order of magnitude greater than that for PARC. STRUT has indeed 10 times more vertices than PARC. The relationship between number of vertices and execution time is an important factor in performing CFD on a multiprocessor since it will help identify the behavior of Jove, which in turn will affect the decision on data movement. Note that the above plots do not take data movement into consideration as this report focuses mainly on establishing a framework for dynamic load balancing.

Table 2 below summarizes our preliminary experimental results, where various individual execution time are listed in terms of number of partitions.

	PARC Grid								STRUT Grid							
	8-set		16-set		32-set		64-set		8-set		16-set		32-set		64-set	
	sec	%	sec	%	sec	%	sec	%	sec	%	sec	%	sec	%	sec	%
Read new data	0.687	8.4	2.865	25.3	1.144	10.3	1.417	10.4	4.791	0.7	7.2	1.0	4.613	0.6	5.232	0.7
Pre-evaluate	0.014	0.2	0.014	0.1	0.019	0.2	0.058	0.4	0.069	0.0	0.081	0.0	0.069	0.0	0.070	0.0
Partition	2.241	27.5	3.192	28.1	4.454	40.3	6.310	46.3	9.152	1.3	12.8	1.8	20.1	2.8	25.18	3.4
Evaluate	5.2	63.9	5.270	46.5	5.439	49.2	5.841	42.9	699.5	98.0	699.9	97.2	701.6	96.6	703.6	95.9
Total	8.142		11.3		11.1		13.6		713.5		720.1		726.5		734.0	

Table 2: Execution time for individual steps.

The table clearly indicates that increasing the number of partitions does not necessarily affect the overall execution time. Note in the last row of the table that when the number of partitions increases to 64, the overall execution time gives merely 3% increase for STRUT. The change for PARC is larger than that for STRUT as it becomes approximately 68%. Considering that full scale CFD applications typically contain hundreds of thousands of vertices, we believe the change will be negligible for vertices on the order of millions. Once full scale CFD applications are running on the SP-2, we will be able to precisely measure the performance of the dynamic load balancer Jove.

8 Conclusions

Parallel computational fluid dynamics is a challenging task. The main difficulty of parallel CFD is the change in computational behavior. As computation progresses, the initial data distribution and processor mapping become invalid as some processors spend much time reading data from remote processors while others have much work to do. This change in computational intensity requires data and computational load be realigned at runtime to improve performance, hence dynamic load balancing. This paper has presented a framework for dynamic load balancing for CFD applications on a 160-processor SP2 multiprocessor. We have developed various algorithms to perform dynamic load balancing.

Our approach is centered around a quick partitioning method based on inertial value of vertices. Two key metrics of vertex similarity and weight similarity are constructed to help decide if the new grid generated by a CFD application should be taken. Vertex similarity was constructed to find how the sets of old grid are distributed over the sets of new grid. Similarly, weight similarity was constructed to find the distribution of computational intensity of the new grid over the sets of old grid. The two metrics are merged into one consistent metric, called absolute similarity with various machine dependent and independent parameters. To ensure processors get the same number of sets, an extensive algorithm has been developed to properly assign new sets to processors. While some sets have a unique number assigned to them, some do not. This observation lead us to develop a set reassignment algorithm. Evaluation step has been performed along the computation of computational gain and communication loss. Decision on a new grid acceptance/ rejection is made based on the comparison of the computational gain and communication loss.

We have completely implemented all the steps described in the report. The whole system, called Jove, is currently running on a single node SP2 multiprocessor. Preliminary experimental results on several grids indicate that the proposed approach is indeed feasible as Jove can correctly characterize the behavior of a complete load balancing system for CFD applications. We anticipate that the proposed approach will be an efficient method for solving grand challenge problems on MIMD message passing machines.

Acknowledgments

The authors would like to thank Eric Barczsz of NASA Ames Research Center for providing various comments on the proposed approach. Special thanks go to Eddy Pramono for creating grids for testing algorithms presented in the report.

References

1. S. Barnard and H. Simon, "Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems," Concurrency: Practice and Experience, 16 (1994), pp. 101 - 117, 1993.
2. T. J. Barth and H. E. Deconinck, *Unstructured Grid Methods for Advection Dominated Flows*, AGARD, 7 Rue Ancelle, 92200 Neuilly-sur-Seine, France, 1992.
3. K. Devine, J. Flaherty, S. Wheat, and A. Maccabe, "A Massively Parallel Adaptive Finite Element Method with Dynamic Load Balancing," in *Proc. IEEE Supercomputing 93*, Portland, Oregon, November 1993, pp.2-11.
4. B. Ghosh and S. Muthukrishnan, "Dynamic Load Balancing in Parallel and Distributed Networks by Random Matchings," in *Proc. ACM Symposium on Parallel Algorithms and Architectures*, June 1994, Cape May, NJ, pp.226-235.
5. B. Hamizadeh and D. Lilja, "Self-Adjusting Scheduling: An On-Line Optimization Technique for Locality Management and Load Balancing," in *Proc. International Conference on Parallel Processing*, St. Charles, IL, August 1994, pp.II-39-46.
6. P. Mehrota, J. Saltz, and R. E. Voigt, Unstructured Scientific Computation on Scalable Multiprocessors, MIT Press, Cambridge, Mass., 1992.
7. E. Pramono, H. Simon, and A. Sohn, "Dynamic Load Balancing for Finite Element Calculations on Parallel Computers", to appear in *Proc. 7th SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.
8. W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, Numerical Recipes in C, Cambridge University Press, 1992.
9. H. Simon, "Partitioning of Unstructured Problems for Parallel Processing," *Computing Systems in Engineering 2*, 1991, pp.135-148.
10. H. Simon (Ed.), Parallel Computational Fluid Dynamics, MIT Press, 1992.

11. H. Simon and A. Sohn, “Dynamic Spectral Partitioning”, Tech. Rep. RNR-94-xxx, NASA Ames Research Center, Moffett Field, CA 94035, October 1994 (in preparation).

12. V. Venkatakrishnan, H. Simon, and T. Barth, “A MIMD Implementation of a Parallel Euler Solver for Unstructured Grids,” *Journal of Supercomputing* 6, 1992, pp.117-137.