Parallel Algorithms for Dynamically Partitioning Unstructured Grids*

Pedro Diniz[†] Steve Plimpton[‡] Bruce Hendrickson[‡] Robert Leland[‡]

Abstract

Grid partitioning is the method of choice for decomposing a wide variety of computational problems into naturally parallel pieces. In problems where computational load on the grid or the grid itself changes as the simulation progresses, the ability to repartition dynamically and in parallel is attractive for achieving higher performance. We describe three algorithms suitable for parallel dynamic load–balancing which attempt to partition unstructured grids so that computational load is balanced and communication is minimized. The execution time of the algorithms and the quality of the partitions they generate are compared to results from serial partitioners for two large grids. The integration of the algorithms into a parallel particle simulation is also briefly discussed.

1 Introduction

Considerable effort has been expended to develop fast, effective algorithms that split unstructured grids into equal–sized partitions so as to minimize communication overhead on parallel machines. The typical approach of performing partitioning as a sequential pre–processing step may be unsuitable in at least three settings. The first is when the grid is so large that pre–processing on a serial machine is not feasible due to memory or time constraints. The second is in simulations that use adaptive gridding to adjust the scale of resolution as the simulation progresses. A classic example is the modeling of shock propagation where adaptive gridding ensures that the domain is sampled on a fine scale where necessary but more coarsely in other regions and hence overall computational effort is reduced. Third, in some problems computational effort in each grid cell changes over time. For example, in many codes that advect particles through a grid (PIC, DSMC) large temporal and spatial variations in particle density can induce substantial load imbalance. In all of these cases repartitioning the grid in parallel as the simulation runs can boost parallel performance. Unfortunately, many of the best sequential partitioning algorithms are difficult to parallelize and have not been designed so as to take advantage of frequent reapplication. Furthermore, since the partitioning is now embedded in the parallel simulation it must be kept economical relative to the total computation.

The parallel partitioning algorithms we describe in this paper are constructed from parallel implementations of two commonly used serial partitioners. The first component is the *Inertial* method [9] which is fast and straightforwardly parallel, but typically produces

*This work was supported by the Applied Mathematical Sciences program, U.S. DOE, Office of Energy Research, and was performed at Sandia National Labs, operated for the U.S. DOE under contract No. DE-AC04-76DP00789.

Appeared in *Proc. 7th SIAM Conf. Parallel Proc. Sci. Comput.*, February, 1995

[†]Department of Computer Science, University of California at Santa Barbara; pedro@cs.ucsb.edu

[‡]Sandia National Labs; {sjplimp, bahendr, leland}@cs.sandia.gov.

partitions of only moderate quality [7]. It is described in Section 2.1. The second component is a parallel variant of a local greedy heuristic due to Fiduccia and Mattheyses (FM) [3] which is closely related to the the well–known method of Kernighan and Lin [5]. This algorithm greedily improves an existing partition by moving grid points between sets. The motivation for using FM is that in sequential algorithms it has been observed that combining the inertial method with FM usually generates high quality partitions very quickly [7]. Unfortunately, a true parallel implementation of FM is known to be P–complete [10], which means a parallel implementation is difficult in a theoretical sense. Hence we use a more convenient but weaker variant which we describe in Section 2.2

Our parallel partitioners combine the inertial algorithm with the FM heuristic improvement in two ways. First, in an algorithm we call Inertial Interleaved FM (IIFM), the FM improvements are performed after every cut in the recursive application of Inertial. In the second algorithm, the Inertial Colored FM (ICFM), the FM refinements are postponed until all Inertial cuts are performed. The IIFM and ICFM algorithms are presented in Sections 2.2 and 2.3. Performance results for the parallel partitioning algorithms on an nCUBE 2 and an Intel Paragon for two large unstructured grids are presented in Section 3 along with a comparison of the partitioning quality to that produced by serial partitioning algorithms. Finally, in Section 4 we highlight the effect of the algorithms in a particle simulation that is a large user of CPU time on Sandia's parallel supercomputers.

2 Algorithms

We represent a computation as an undirected, weighted graph in which vertices correspond to computation and edges reflect data dependencies. Weights can be assigned to vertices or edges to encode heterogeneous computation or communication requirements. Many scientific computing problems including finite element, finite difference, and particle calculations can be phrased this way. The goal of partitioning can now be described as dividing the vertices into sets of equal weight in such a way that the weight of edges crossing between sets is minimized. This problem is known to be NP hard, so we are forced to rely on heuristics. All of our heuristics rely on a recursive bisection approach.

2.1 Parallel Inertial

The Inertial method [9] employs a physical analogy in which the grid cells are treated as point masses and the grid is cut with a plane orthogonal to the principal inertial axis (the axis about which there is a minimal moment of inertia) of the mass distribution.

At each bisection step the involved processors compute the inertial axis of the assigned subgraph. Collectively they find a point along this axis such that the weights of the graph vertices on both sides are approximately equal. The two vertex sets are then assigned to the two processor sets.

Figure 1(a) illustrates this algorithm for a four processor machine. In the first step all four processors cooperate finding the inertia axis $A1$ and the corresponding cut $C1$. Processors are also split into two sets, e.g. $\{p0, p1\}$ and $\{p2, p3\}$, where p_i owns set S_i. Processors $p0$ and $p1$ send the graph vertices they own on the left side of this cut to processors $p2$ and $p3$ respectively, who proceed conversely. We say that $p0$ pairs with $p2$ and $p1$ pairs with $p3$. In the next step $p0$ and $p1$ find the next inertia axis $A2a$, the corresponding cut $C2a$ and swap graph nodes on each side of the cut. Simultaneously $p2$ and $p3$ are finding inertia axis $A2b$, cut $C2b$, and swapping their nodes across their cut.

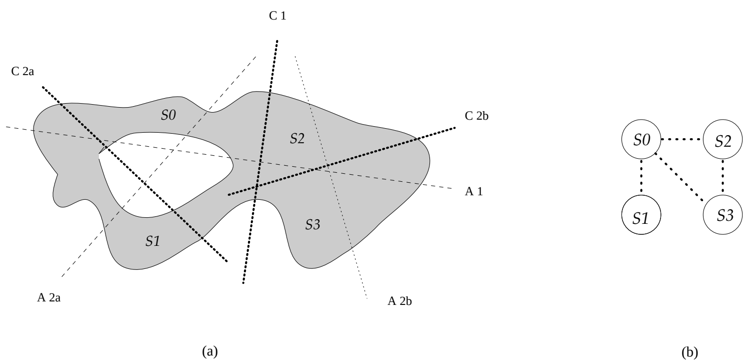

FIG. 1. *Inertial method example for 4 processors.*

2.2 Inertial Interleaved FM (IIFM)

In this algorithm each graph bisection performed by the parallel Inertial method is followed by FM heuristic refinement. This FM refinement greedily improves the bisection by moving vertices between the two partitions so as to minimize a grid metric, e.g. *cuts* – the number of edges connecting vertices on each partition. This is achieved by associating with each vertex a migration preference value corresponding to each partition and selecting at each step of the improvement the vertex with the largest contribution to the metric minimization. Partition weights can be balanced by always moving vertices from the "lightest" to the "heaviest" partition. This greedy strategy proceeds until no further improvement is possible.

Unfortunately, FM is inherently sequential and in a formal sense has been proven difficult to parallelize [10]. Our approach differs from that in standard serial implementations. We build on an idea described by Hammond [4] and apply FM in a pairwise fashion; that is, two processors perform FM on the subpieces of the partition they own. When only two processors are involved, FM can be quite efficient.

In our approach many different pairs of processors work simultaneously following each Inertial bisection. The pairing processors construct a sequence of possible vertex exchanges in an effort to reduce the number of edges cut by the partition. At each step in this construction, the vertex move with the highest migration preference among those which would not affect load balance too adversely is identified. The quality of the partition which would be observed if the moves specified by the sequence to this point were actually made is also recorded. When the sequence construction is complete we return and make the specified moves up through the point where the best partition was observed. This two-phase approach allows counter–productive moves in order to overcome local minima in the cost metric. A halting criterion is imposed on the number of counter–productive moves to terminate expensive and probably unhelpful sequences.

In the example above processors $p0$ and $p2$ (and also $p1$ and $p3$) would pair up after the first Inertial cut to improve the quality of their sets, typically by modifying the boundary between them slightly. After the second Inertial cut we would observe pairing between processors $p0$ and $p1$ as well as between $p2$ and $p3$.

2.3 Inertial Colored FM (ICFM)

This algorithm is composed of two phases. First the grid is partitioned using the parallel Inertial algorithm. Then the partition resulting from the first step is improved using the FM heuristic in a pairwise fashion, i.e. all pairs of processors whose partitions share common edges cooperate to improve the partitions they jointly own.

Note that processor pairs whose grid partitions are not contiguous may work concurrently. To exploit this we construct a *quotient graph*, $\mathcal{G}$, of the partition obtained in the first phase. Each vertex of $\mathcal{G}$ corresponds to a partition, and vertices are joined by an edge if the corresponding partitions have grid vertices joined by an edge. The quotient graph is then edge–colored so that pairwise FM can be applied to all edges of a color simultaneously. The coloring thus determines the scheduling of the pairwise improvement between partitions. In Fig. 1(b) we show the quotient graph corresponding to the partition in Fig. 1(a). A possible coloring and resulting pairwise scheduling would be $\{(0,1),(2,3)\},\{(0,2)\},\{0,3\}$. In the current implementation we perform only one sweep through the colors, and use a simple edge coloring heuristic to keep the number of colors small since the general color minimization problem is NP-complete.

3 Results

We tested the parallel algorithms described here on a variety of grids and present typical results for two of them. The first is the *WAVE* mesh from RIACS [1], a tetrahedral finite element grid of a 3D airplane with about 157K vertices and 1M edges. The second is the *CVD* mesh generated by Hennigan at Sandia, a hexahedral finite element grid of a 3D chemical vapor deposition reactor with about 158K vertices and 1.9M edges.

In addition to measuring execution time, we also evaluated the quality of the partitions produced by each method. We used two quality metrics, *cuts* – the number of edges connecting vertices in different partitions and hence a measure of the volume of communication, and *startups* – the number of partition connections (also the number of edges in the quotient graph) and hence the number of interprocessor messages. For comparison, we also partitioned the two grids with several serial algorithms using Chaco [8], a static load balancing tool. Two of the serial algorithms used were Inertial and Inertial coupled with FM (*Inertial+FM*) which are similar to their parallel counterparts; the third serial method used was a more sophisticated Multilevel method [6].

The parallel algorithms were run on an nCUBE 2 hypercube and on an Intel Paragon located at Sandia National Labs. The Chaco runs were performed on an SGI Onyx with a 125 MHz clock. Table 1 presents the results for 64 and 1024 partitions for both grids while Fig. 2 plots the parallel run times for the *CVD* grid for 64 through 1024 partitions. The parallel runs were performed on the same number of processors as partitions, typical of how they would be run when embedded in an application. Thus a run on 128 processors will not be twice as fast as one on 64 processors, since more partitions are created.

The tables show the ICFM algorithm outperforms the IIFM algorithm both in quality and partition time, attaining a reduction in cuts of about 10% over pure Inertial but with a slight increase (less than 2%) in startups. For large number of partitions, both IIFM and ICFM are an order of magnitude slower than parallel Inertial. Thus for problems where the cost of partitioning is critical, the parallel Inertial method is attractive, while in settings where the user is willing to spend more time to get a better partition, the ICFM technique would be better.

The tables also show that the Inertial+FM algorithm in Chaco is significantly more effective than its parallel counterpart – the IIFM. This is due to the pairwise parallel approach that allows exchanges only between subpieces of the full vertex set. Chaco's Multilevel FM is the most effective method, providing up to a 25% reduction in both grid

[1] Available through anonymous ftp to riacs.edu in the file /pub/grids/wave.grid

Method	64 partitions							
	WAVE				CVD			
	Time (secs)		Cuts	Startups	Time (secs)		Cuts	Startups
	nCUBE	Paragon			nCUBE	Paragon		
Inertial	7.70	1.14	121081	728	12.8	1.51	324020	674
IIFM	34.7	4.48	115827	762	72.2	8.13	317205	678
ICFM	14.4	2.23	110899	714	21.3	2.99	310266	680
Inertial+FM	41.6		98626	716	51.3		286608	618
Multilevel	102.8		96746	732	147.0		240697	588
Method	1024 partitions							
	WAVE				CVD			
	Time (secs)		Cuts	Startups	Time (secs)		Cuts	Startups
	nCUBE	Paragon			nCUBE	Paragon		
Inertial	3.92	0.67	347801	15748	4.36	0.71	842547	16978
IIFM	100	5.25	331643	15782	158	7.78	773104	17112
ICFM	14.9	1.93	315587	15808	17.3	3.36	732548	16998
Inertial+FM	96.7		285345	14478	118.0		656183	14892
Multilevel	218.4		285330	14472	297.9		650937	14774

TABLE 1
Performance results on nCUBE 2 and Paragon multiprocessors.

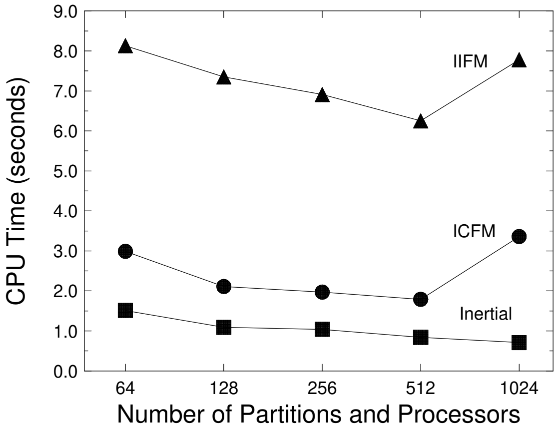

FIG. 2. *Partition time for the* CVD *grid on the Paragon.*

partition metrics considered. However, even on a very fast single processor, it is two orders of magnitude slower than parallel Inertial and 10 to 50 times slower than ICFM and IIFM respectively, running on a large parallel machine.

4 Application

We tested the parallel load–balancing algorithms in a parallel particle code used at Sandia for simulating very low–density fluid flows [1]. Because continuum approaches such as the Navier Stokes equations break down in this regime, the code uses Direct Simulation Monte

Carlo (DSMC) techniques [2] to represent the fluid as a collection of particles. Grids are used in the DSMC computation to locate nearby collision partners. The natural parallelism is to have each processor own a subset of grid cells and the particles in those cells. The computations for particle collisions and moves can then be performed independently by each processor. Inter–processor communication is performed at each timestep as particles move to new cells.

Load–imbalance is a serious concern in DSMC simulations because particle densities on the grid can vary by orders of magnitude both spatially and in time. In the past we have used a static scattered decomposition of cells to processors so that each processor, on average, owns a mix of "costly" and "cheap" cells. However this maximizes the communication since a cell's neighbors are not owned by the same processor. The parallel balancers can assign clusters of cells to each processor to minimize communication while still keeping the computational load dynamically balanced.

Using the parallel balancers in our DSMC code requires two steps. First, a new assignment of cells to processors is computed by the load–balancer using the CPU time logged in previous timesteps for particle moves and collisions in each grid cell as the cell's "weight". Then all of the appropriate particle and cell data are sent to the new processors who now own those grid cells. For a prototypical simulation of a nozzle ejecting gas into a vacuum (17300 grid cells, 1.5 million particles), we found the parallel inertial balancer worked best when run once every 200 timesteps. The total cost of one call to the balancer (balance plus data–send) was about equal to the cost of 2–4 timesteps. Using the pure Inertial balancer, the overall run time of a 10000 timestep simulation was reduced from 2116 seconds (with a static scattered decomposition) to 1642 seconds on 512 processors of the Intel Paragon, a savings of 22%. We are still working to integrate the full IIFM and ICFM balancers into the DSMC simulation and will report on their efficacy at a later time.

References

[1] T. J. BARTEL AND S. J .PLIMPTON, *DSMC Simulation of Rarefied Gas Dynamics on a Large Hypercube Supercomputer*, in Proc. AIAA 27th Thermophysics Conference, AIAA 92-2860.

[2] G. A. BIRD, *Molecular Gas Dynamics*, Clarendon Press, Oxford (1976).

[3] C. M. FIDUCCIA AND R. M. MATTHEYSES, *A linear time heuristic for improving network partitions*, in Proc. 19th IEEE Design Automation Conference, IEEE, 1982, pp. 175–181.

[4] S. HAMMOND, *Mapping unstructured grid computations to massively parallel computers*, PhD thesis, Rensselaer Polytechnic Institute, Dept. of Computer Science, Troy, NY, 1992.

[5] B. KERNIGHAN AND S. LIN, *An efficient heuristic procedure for partitioning graphs*, Bell System Technical Journal, 29 (1970), pp. 291–307.

[6] R. LELAND AND B. HENDRICKSON, *A multilevel algorithm for partitioning graphs*, Technical Report SAND93-1301, Sandia National Laboratories, Albuquerque, NM 87115, October 1993.

[7] ——,*An empirical study of static load balancing algorithms*, in Proc. Scalable High Performance Computing Conf., Knoxville, TN, June, 1994.

[8] ——, *Chaco user's guide - Version 1.0* Technical Report SAND93-2339, Sandia National Laboratories, Albuquerque, NM 87115, November 1993.

[9] B. NOUR-OMID, A. RAEFSKY, AND G. LYZENGA, *Solving finite element equations on concurrent computers*, in Parallel computations and their impact on mechanics, A. K. Noor, ed., American Soc. Mech. Eng., New York, 1986, pp. 209–227.

[10] J. SAVAGE AND M. WLOKA, *Parallelism in graph partitioning*, J. Par. Dist. Comput., 13 (1991), pp. 257–272.