1993

Fast Mapping And Remapping Algorithms For Irregular And Adaptive Problems

Chao Wei Ou
Syracuse University

Sanjay Ranka
Syracuse University

Geoffrey C. Fox
Syracuse University

Fast Mapping And Remapping Algorithms For Irregular And Adaptive Problems*

Chao-Wei Ou, Sanjay Ranka and Geoffrey Fox
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

Abstract

This paper describes the performance of locality-based mapping and remapping partitioners for unstructured grids. We show that the algorithm produces good mappings at a relatively low cost and can be easily parallelized. Further, the algorithm can provide remapping for incremental problems at a fraction of the total cost.

1 Introduction

Load-balancing and reduction of communication are two important issues for achieving good performance distributed-memory parallel computers. It is important to map the program such that the total execution time is minimized; the mapping can typically be performed statically or dynamically.

For a large class of scientific problems that are irregular in nature, achieving a good mapping is difficult [1]. The nature of the irregularities is unknown at the time of compilation and can be derived only at runtime. The handling of such irregular problems requires runtime information in order to partition the computation in such a fashion that each processor receives an approximately equal amount of computation and to minimized communication.

Partitioning for dynamic problems requires optimization methods that quickly and reliably produce reasonable, but not exact results. Partitioning such applications can be posed as a graph-partitioning problem necessarily based on the computational graph for each phase. The partitioning problem is in the class of NP-complete problems; hence exact solutions are computationally intractable for large problems.

*This work was supported in part by NSF under CCR-9110812 and in part by DARPA under contract #DABT63-91-C-0028. The contents do not necessarily reflect the position or the policy of the United States government and no official endorsement should be inferred.

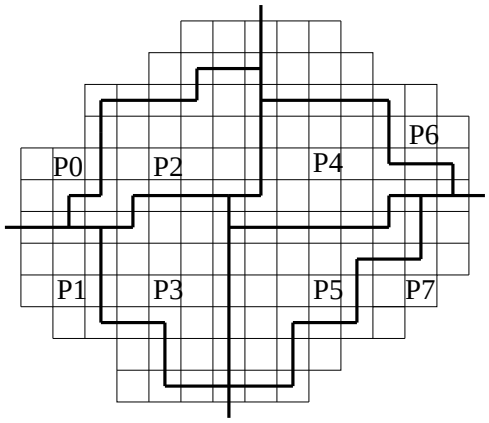

Figure 1: The partitioning of irregular mesh

However, we emphasize that good suboptimal solutions are sufficient for effective parallelization of a large class of irregular problems.

There are a large number of partitioning algorithms available in the literature [2], [6], [9]. Depending on the requirement application, one may be more useful than the other. The following are some important features of a partitioning algorithm.

1. *Cost of partitioning vs. quality*: For a given application, a cheaper algorithm generating a solution of reasonable quality may be preferable to an expensive one that yields a solution of very good quality.
2. *Direct vs. iterative*: In iterative methods (e.g., genetic algorithms) the quality of partitioning improves with the number of iterations, and thus the user can optimize between the cost vs quality of the mapping.
3. *Parallelizability*: Some methods, such as genetic algorithm-based partitioners, are inherently parallel. On the other hand, methods based on recursive spectral bisection are difficult to parallelize.
4. *Incremental updates*: For many applications, the computational structure changes from one phase to another in an incremental fashion. Thus, partitioning

of the previous phase can be used to partition the next phase at a fraction of the cost.

5. *Use of information about physical domain*: Co-ordinate information can be used if the computational graph represents a physical domain.

In this paper we present the quality of mapping produced by an index-based mapping scheme for partitioning two and three-dimensional irregular and adaptive grids on parallel machines. We show these methods to be extremely fast, easy to parallelize, that they produce good mappings, and are incremental in nature. Thus, we believe they should be useful for a large variety of irregular and adaptive problems. Index-based mapping has been used for sorting on a two-dimensional mesh [7], parallelizing quadtrees and sparse images [4] [5], and for n-body simulations on parallel machines [8].

The quality of the mappings produced by our algorithms is comparable to co-ordinate recursive bisection [6]. Although the algorithm does not perform as well as spectral bisection methods, it is easily parallelizable and should be useful for parallelizing problems that are adaptive in nature.

2 The Mapping Problem

We are given a graph $G = (V, E)$, where V represents a set of vertices, and E represents a set of edges. The number of vertices is given by $n =\mid V \mid$, and the number of edges is given by $m =\mid E \mid$. For a graph representing the computational structure of physical domain, each vertex $v_i \in V$, $1 \leq i \leq m$ corresponds to a physical coordinate in a d-dimensional space $(x_{i_1}, x_{i_2}, \ldots, x_{i_d})$. Each edge is an ordered pair (v_{i_1}, v_{i_2}). In graphs corresponding to computational structure of physical domain, edges connect physically proximate vertices.

The graph-partitioning problem can be defined as an assignment scheme $M : V \longrightarrow P$ that maps vertices to partitions. We denote by $B(q)$ the set of vertices assigned to a partition q. Thus $B(q) = \{v \in V : M(v) = q\}$. The weight w_i corresponds to the computation cost (or weight) of the vertex v_i. The cost of an edge $w_e(v_1, v_2)$ is given by the amount of interaction between vertices v_1 and v_2, thus the weight of every partition can be defined as

$$W(q) = \sum_{v_i \in B(q)} w_i.$$

The cost of all the outgoing edges from a partition represents the total amount of communication cost and is given by

$$C(q) = \sum_{v_i \in B(q),\, v_j \notin B(q)} w_e(v_i, v_j).$$

We would like to make an assignment such that the time spent by every node is minimized, i.e.,

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

(a)

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

(b)

Figure 2: (a) Row-Major and (b) Shuffled Row-Major Indexing for an 8×8 image

$$\max_q (W(q) + \beta C(q)),$$

where β represents the cost of unit computation/cost of unit communication on a machine. It is more convenient to minimize

$$\sum_q C(q)$$

because: 1) The computational load is typically balanced by most algorithms and thus the first term is close to $\sum_q W(q) \ / \ P$ for each partition and can be factored out; 2) the max function is not differentiable; most optimization methods are gradient descent methods and hence require minimization of a differentiable function.

2.1 Incremental problems

An adaptive irregular computation consists of a loosely synchronous computation executed repeatedly in which the data access pattern changes between iterations [1]. The changes may be gradual, reflecting adiabatic changes in the physical domain (e.g., molecular dynamics), or large-scale reflecting additions to a data structure (e.g., adaptive PDE solvers). The physical and numerical properties of these algorithms typically guarantee that large-scale restructuring of data is needed infrequently. Thus, from the perspective of the incremental mapping problem, the following scenarios may arise:

1) All the coordinates may perturb.

2) New points may be added and/or old points deleted.

This paper is limited to the latter case.

3 The Mapping Scheme

Mapping is based on converting an n-dimensional co-ordinate into a one-dimensional index such that proximity in the multi-dimensional space is usually

```
    Indexing(hash,d,n)
1    for j ←− 1 to d do
2        unit_j ←− ((max_{i=1}^{n} x_ij) − (min_{i=1}^{n} x_ij))/2^lj
3    for i ←− 1 to n do
4        for j ←− 1 to d do
5            index_j ←− x_ij/unit_j
6        hash_i ←− Interleave(index,d,l)
```

Figure 3: Indexing algorithm

```
/* Sorted array A is distributed using block distribution */
/* Unsorted array B is distributed using block distribution */
/* Bound[i] is the largest key of A stored in processor i */
For each processor i do in parallel
Step 1 : VAL := Global_concatenate(Bound[i])
Step 2 : For k ←− 1 to p do
             SEND_LIST[k] := nil
Step 3 : For k ←− 1 to m_i do
             proc := Binary_search(m_i, VAL)
             Add B[k] to SEND_LIST[proc]
Step 4 : All-to-Many communication using SEND_LIST
Step 5 : Sort all the points received in Step 4 and call it C
Step 6 : Merge list A and C
Step 7 : For k ←− 1 to p do
             Refined_Bound[k] := k(n+m)/p
Step 8 : Perform a locality-maintaining Load_Balance
             according to Refined_Bound[k]
```

Figure 4: Parallel Merging Algorithm

maintained [8]. Consider a graph in which the set of vertices are arranged an 8×8 grid. Row-major indexing and shuffled row-major indexing are two of the several ways to index pixels in a two-dimensional grid. These two indexing schemes are shown in Figure 2. Intuitively, one would expect that shuffled row-major mapping maintains the two-dimensional proximity of indices better than row-major indexing does. With no loss of generality, we assume the vertices in the physical space are all mapped onto a logical grid of size $2^{l_1} \times 2^{l_2} \times 2^{l_3}$ such that $l_1 \geq l_2 \geq l_3$. The indexing algorithm is given in Figure 3.

A simple example of interleaving indices is as follows. Suppose

$$index_1 = 101 \; index_2 = 01 \; index_3 = 0.$$

The interleaved index would be 100110; this is done by choosing bits (right to left) of each of the dimensions one by one, starting from dimension 3 (the dimension with the smallest number of bits). When the bits of a particular dimension are no longer available, that dimension is not considered.

The main purpose of a mapping algorithm is to determine the partitions by dividing the sorted index list. The algorithm assumes the input is a d-dimensional array. Once the index of every point is obtained, a simple sorting algorithm can be employed to provide the required mapping. We have used a sample-based sorting algorithm for our implementation (it is omitted in this paper due to space limitations).

For the incremental problem, all index values of incremental points are inserted into the sorted list. A simple merging algorithm can be used for repartitioning data when new nodes are added or deleted. This problem can be described as merging m numbers (no ordering between them) with a sorted list of n numbers to give another sorted list. The sequential complexity of this algorithm is $O(m \log m + n)$.

The parallel merging algorithm applies the indexing algorithm to the new vertices and moves them to processors, based on the previous boundaries. A sequential merge algorithm in each processor forms a sorted list of size $\frac{n}{p} + m_i$ where m_i is the number of vertices to be inserted in the processor i. One can use a parallel prefix algorithm to find the new (refined) boundaries. This is followed by a locality-maintaining **Load_Balance** algorithm [3] that balances the load. It can be shown that the worst case complexity of the merging algorithm is $O(m \log m + \frac{n}{p})$.

4 Experimental Results

The results presented in Table 1 were obtained by applying the index-based mapping algorithm to a large number of meshes. We can make the following observations about the index-based mapping: 1) The quality of partitioning for a small number of partitions is not very good. 2) The quality of partitioning degrades if the mesh is highly irregular. 3) For large meshes the quality of mapping is comparable/better than co-ordinate recursive bisection. 4) The time required for partitioning is independent of the number of partitions. 5) For large mesh sizes with a reasonable number of partitions, the algorithm gives better performance than CRB at less then half the cost. 6) The quality of partitioning is always worse than SRB. However, the time required is two to three magnitudes better.

We thus see that this algorithm is comparable/better than CRB for large meshes and a reasonable number of partitions. Clearly, the performance is always inferior to that of SRB, but at a much lower cost.

To study the time for parallelization for different values of N, the co-ordinate data was generated randomly. The algorithm was implemented on a CM-5. Figure 5 shows the timing on 4, 8, 16, and 32 nodes. For 128,000 vertices, the time taken is of the order of 0.69 seconds on a 32-node CM-5. The time taken for

$\mid V \mid = 2800, \mid E \mid = 17377$			
Partition	Partitioner	Time	Cutset
16	SORT	.399960	4785
	CRB	.329967	4501
	SRB	43.696	3421
64	SORT	.399960	8172
	CRB	.479952	8563
	SRB	56.154	6385
256	SORT	.389961	12226
	CRB	.669933	13078
	SRB	64.194	10566
$\mid V \mid = 2851, \mid E \mid = 15093$			
Partition	Partitioner	Time	Cutset
16	SORT	.379962	2840
	CRB	.309969	2176
	SRB	72.213	1455
64	SORT	.389961	5918
	CRB	.439965	4806
	SRB	82.272	3395
256	SORT	.379962	10108
	CRB	.659934	8452
	SRB	91.081	7238
$\mid V \mid = 9428, \mid E \mid = 59863$			
Partition	Partitioner	Time	Cutset
16	SORT	1.32987	10936
	CRB	1.14988	9731
	SRB	203.820	7236
64	SORT	1.29987	19165
	CRB	1.71983	20147
	SRB	247.695	14310
256	SORT	1.33987	30799
	CRB	2.27977	37272
	SRB	280.712	25073
$\mid V \mid = 53961, \mid E \mid = 353476$			
Partition	Partitioner	Time	Cutset
16	SORT	7.76922	36128
	CRB	7.22928	31753
	SRB	1719.768	49374
64	SORT	7.74923	65958
	CRB	10.9389	77313
	SRB	2234.786	66596
256	SORT	7.84921	108692
	CRB	14.4486	151359
	SRB	2523.358	95612

Table 1: Comparison of SORT, CRB, SRB algorithms (time is in seconds)

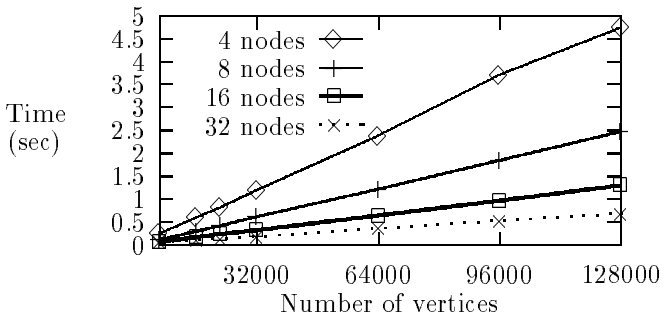

Figure 5: Parallelization of mapping algorithm on 4, 8, 16, and 32 nodes

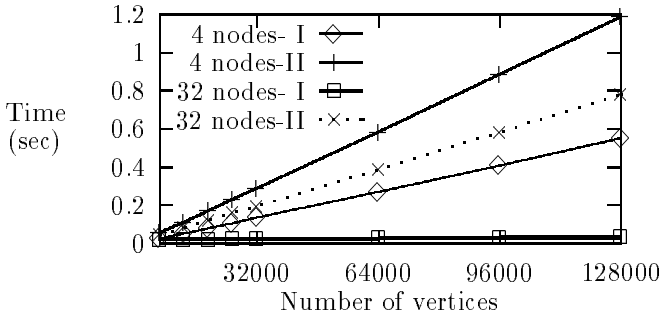

Figure 6: 10% incremental problem on 4, 8, 16, 32 processors (I and II represent cases 1 and 2, respectively)

the algorithm on 4 nodes was 4.75 seconds. Thus, the algorithm scales well.

For the incremental case, we generated two sets of data for performing our experiments.

1. Each node generated an approximately equal number of random points such that the index values were within the boundaries of each processor.

2. One node generated all the points (m) such that the points were within the boundaries of a processor. This case was followed by a load- balancing step in which the data was distributed to all processors equally ($\frac{m}{p}$).

The results(Figure 6) show that for case 1 the algorithm parallelizes very well. The cost on 32 nodes in the incremental case of 10% new vertices is approximately 0.03 second for 128,000 vertices. This shows

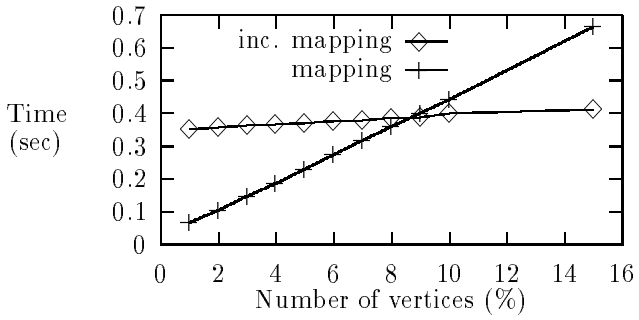

Figure 7: Comparison between incremental mapping and mapping algorithms on case 2 ($|V| = 64,000$ and $p = 32$)

that the incremental mapping algorithm can be used to reduce the time for repartitioning (the corresponding time for mapping 128,000 vertices on 32 nodes is 0.69). For case 2 result(Figure 6), the algorithm does not scale very well with the number of processors unless the fraction is small and the number of vertices are large (greater than 10,000). This is because a large number of messages (p) are received by one processor. Further, all the data is sorted in one processor in Step 5.

Figure 7 shows the comparison of incremental mapping (using the merging algorithm) with the mapping algorithm (sorting algorithm) for 64,000 vertices for the worst case data. This result shows that it is better to perform incremental mapping rather than mapping when the fraction is less than 9%.

5 Conclusions

In this paper we have described a simplex index-based algorithm for graph partitioning. It is shown that an index-based algorithm should be useful for partitioning unstructured and adaptive problems for the following reasons:
1. They provide good solutions with a relatively low cost, which is a necessary requirement due to the adaptive nature of problems.
2. They can be parallelized.
3. They can be used for problems that are incremental in nature.

The performance of our parallel incremental mapping depends on the type of data generated. There is a big gap between the performance of the best case and the worst case of our algorithm. We are currently conducting further research in this area to improve the worst case performance of the incremental algorithm.

References

[1] Alok Choudhary, Geoffrey C. Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Sanjay Ranka, and Joel Saltz. Software support for irregular and loosely synchronous problems. In *Proceedings of the Conference on High Performance Computing for Flight Vehicles*, 1992. to appear.

[2] Nashat Mansour. *Parallel Genetic Algorithms with Application to Load Balancing for Parallel Computing*. PhD thesis, Syracuse University, Syracuse, NY 13244, 1992.

[3] Kishan Mehrotra, Sanjay Ranka, and Jhy-Chun Wang. A probabilistic analysis of a locality maintaining load balancing algorithm. In *7th International Parallel Processing Symposium*, Newport Beach, CA, April 1993.

[4] R. Shankar and S. Ranka. Hypercube algorithms for quadtree operations. *Journal of Pattern Recognition*, September 1992.

[5] R. Shankar and S. Ranka. Computer vision algorithms for sparse images. *Journal of Pattern Recognition*, October 1993.

[6] H. Simon. Partitioning of unstructured mesh problems for parallel processing. In *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Permagon Press, 1991.

[7] C.D. Thompson and H.T. Kung. Sorting on a mesh-connected parallel computer. *comm. ACM*, 20:263–271, 1977.

[8] Michael S. Warren and John K. Salmon. Astrophysical n-body simulations using hierarchical tree data structure. In *Proceedings Supercomputing '92*, Minneapolis, November 1992.

[9] R.D. Williams. Performance of dynamic load-balancing algorithm for unstructured mesh calculations. *Concurrency*, 3:457–481, 1991.