Fast and Parallel Mapping Algorithms for Irregular Problems

Chao-Wei Ou, Sanjay Ranka, and Geoffrey Fox

Northeast Parallel Architectures Center
School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

Email: *cwou@npac.syr.edu, ranka@top.cis.syr.edu*

Abstract

In this paper we develop simple index-based graph partitioning techniques. We show our methods to be very fast, easily parallelizable and that they produce good quality mappings. These properties make them useful for parallelization of a number of irregular and adaptive applications.

Index Terms: Mapping, Remapping, Parallel, Merging, Sorting

1 Introduction

Parallelization of data-parallel programs on distributed-memory parallel computers requires careful attention to load balancing and reduction of communication to achieve a good performance. For most regular and synchronous problems [13], mapping can be performed at the time of compilation by giving directives to decompose the data and its corresponding computations [8]. For irregular applications, achieving a good mapping is considerably more difficult; the nature of the irregularities may not be known at the time of compilation and can be derived only at runtime [7]. These applications can be represented as computational graphs from the perspective of parallel computing. The vertices of these graphs represent tasks that can be executed concurrently, while the edges represent the interactions between them.

The key problem in efficiently executing irregular applications is partitioning the data and computation to minimize communication while balancing the load (Figure 1). Partitioning such applications can be posed as a graph-partitioning problem that is shown to be NP-complete [16]; hence exact solutions are computationally intractable for large problems. Several heuristic methods are available in the literature to perform such partitioning. These methods include recursive coordinate bisection, inertial bisection, scattered decomposition, geometry-based partitioners, simulated annealing, mean-field annealing, recursive spectral bisection, recursive spectral multisection, mincut-based methods, and genetic algorithms [1, 11, 12, 14, 15, 17, 19, 23, 24, 26, 28, 29, 36, 41]. The computational graphs derived from many applications are such that the vertices correspond to two- or three-dimensional coordinates, and the interaction between computations is limited to physically proximate vertices. Examples of such applications include molecular dynamics, static and adaptive PDE solvers [30, 38], region-growing, component labeling [10], and statistical physics simulations [6, 7, 9, 10]. Simple and fast heuristics for partitioning such graphs is to cluster physically proximate points in two or three dimensions.

Most of the above applications are iterative and the same computational graph is used for several iterations. The average time required to solve one iteration of these irregular data-parallel applications depends on the sum of the amortized cost of partitioning A, (time required for partitioning divided by the number of times the same partitioning is used), and the time required for computation and communication C. A good partitioning keeps the load balanced and the amount of communication low and has a small value of C. When the computational graph is static the cost of partitioning is typically neglected; a more expensive algorithm which produces a high-quality solution is desirable. When the computational structure can be determined only at runtime the graph-partitioning cost cannot be ignored. A cheaper algorithm that produces a good-quality solution may be preferable for such cases. Further, it is highly desirable that the partitioners should themselves be parallelizable.

For many irregular and/or adaptive applications, the computational structure changes incrementally. These changes may reflect perturbations in the physical domain (e.g., molecular dynamics applications [6]) or reflect additions/deletions to a data structure (e.g., adaptive PDE solvers [30, 38]). In such cases one option is to repartition the new computational

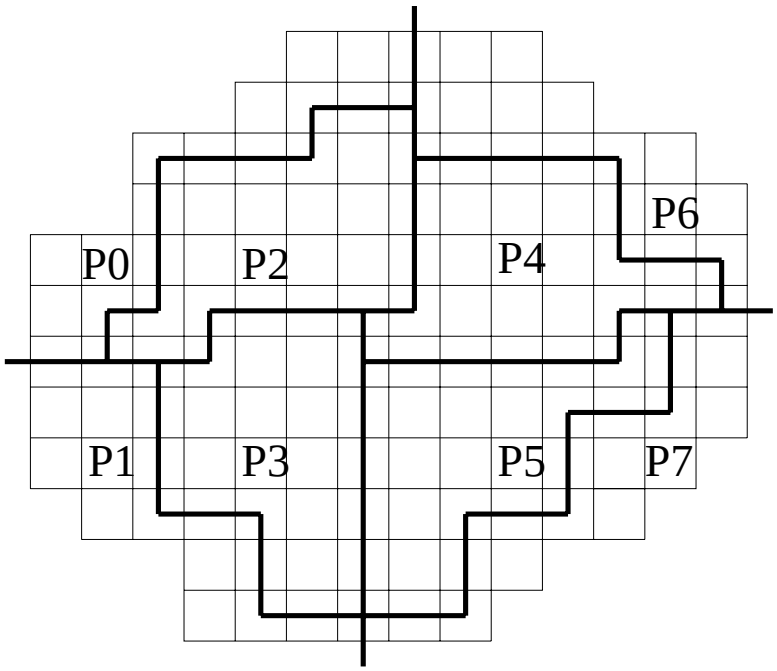


Figure 1: The partitioning of an irregular mesh

graph from scratch; however, information about partitioning from the previous phase can potentially be exploited to accelerate the partitioning for the new phase.

In this paper we describe index-based mapping schemes for partitioning two- and three-dimensional irregular and adaptive grids on parallel machines. We show that these methods are extremely fast, easy to parallelize, and produce good-quality mappings. When new vertices are added and/or old vertices deleted, the repartitioning can be achieved faster than partitioning from scratch. We believe these methods should be useful for a wide variety of irregular and adaptive problems. Index-based mappings have been used to develop efficient algorithms for sorting [37], quadtree operations [32], sparse images [33], and n-body simulations [39] on parallel machines.

The rest of the paper can be described as follows. In Section 2 we briefly describe the mapping problem; Section 3 describes the important parallel primitives required for the mapping and remapping algorithms; Section 4 gives an overview of our mapping scheme; Section 5 discusses the mapping algorithm; Section 6 shows the parallelization of the mapping algorithm, and Section 7 exploits the incremental aspects of the remapping algorithm.

2 The Mapping Problem

Let 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 the computational structure of a physical domain, these 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

$$W(q) \ = \sum_{v_i \in B(q)} w_i. \tag{1}$$

A cross-edge is defined as an edge whose end vertices belong to different partitions. The cost of all the cross-edges from a partition is given by

$$C(q) \ = \sum_{v_i \in B(q), v_j \notin B(q)} w_e(v_i, v_j). \tag{2}$$

For the purpose of mapping on parallel machines, a number of different but related cost functions have been described in the literature. Most formulations decompose the computation and communication costs separately. The total cost of mapping is given by a linear combination of the two terms. The computation cost is given by $\max_q \ W(q)$, which is typically approximated by $\sum_q (W(q) - \mu)^2$, where $\mu = \frac{\sum_q W(q)}{p}$. The communication cost is given by the sum of all the cross-edges (or the cut weight):

$$C \ = \sum_q C(q). \tag{3}$$

Some formulations also use the following metric for representing the communication cost:

$$\max_q \ C(q). \tag{4}$$

We will provide the quality of mapping achieved by using both the communication metrics, assuming that the computational loads are balanced.

3 Index-Based Mapping

This mapping is based on converting an n-dimensional coordinate into a one-dimensional index such that proximity in the multi-dimensional space is usually maintained [37, 39]. Consider a graph in which the set of vertices are arranged in a grid of size 8×8. Row-major indexing and shuffled row-major indexing are two of the several ways of indexing pixels in a two-dimensional grid (Figure 2). These two indexing schemes are shown in Figure 2 (a) and Figure 2 (b), respectively. Intuitively, it can be seen that shuffled row-major indexing maintains a two-dimensional proximity of indices better than row-major indexing.

00	01	02	03	04	05	06	07
08	09	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)

00	01	04	05	16	17	20	21
02	03	06	07	18	19	22	23
08	09	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: Different indexing schemes for an 8×8 image: (a) Row-Major and (b) Shuffled Row-Major

We assume that the vertices of the computational graph are embedded in a d-dimensional space. An algorithm for mapping such a graph using a bit-interleave-based index is given in Figure 4. The vertices are first embedded in a logical grid of size $2^{l_1} \times 2^{l_2} \times 2^{l_3} ... \times 2^{l_d}$, which is done by calculating the bounding hyperplanes along every dimension and decomposing the space along d, the dimension in 2^{l_d} bins. Thus the number of bits required for representing the bin that a vertex belongs to along the ith dimension is given by l_i. Each vertex is now represented by an integer tuple $(cord_1, cord_2, ...cord_d)$. This tuple is transformed into a one-dimensional index (Figure 3) using the bit-interleaving algorithm given in Figure 5. This algorithm assumes that $l_1 \geq l_2 \geq l_3 ... \geq l_d$ and can be easily modified for the general case. The algorithm chooses bits (right to left) of each of the dimensions one by one, starting from the dimension with the smallest number of bits. When the bits of a particular dimension are no longer available, that dimension is not considered.

Example 1: Suppose $index_1 = 001$, $index_2 = 010$, and $index_3 = 110$. The number of bits in each dimension is equal to 3. The interleaved index is 001011100.

Example 2: Suppose $l_1 = 3$, $l_2 = 2$, and $l_3 = 1$, the indices are listed as $index_1 = 101$, $index_2 = 01$, and $index_3 = 0$. The interleaved index is 100110.

Bit-interleaving is one of the many transformation functions from an n-dimensional domain to a one-dimensional domain such that proximate vertices in the original domain are generally mapped to proximate vertices in the one-dimensional domain.

Once the index of every vertex is obtained, a sorting algorithm can be used to provide a nondecreasing ordering of vertices based on their index values. This is followed by dividing the list into P consecutive sublists of nearly equal size. Each sublist represents one partition.

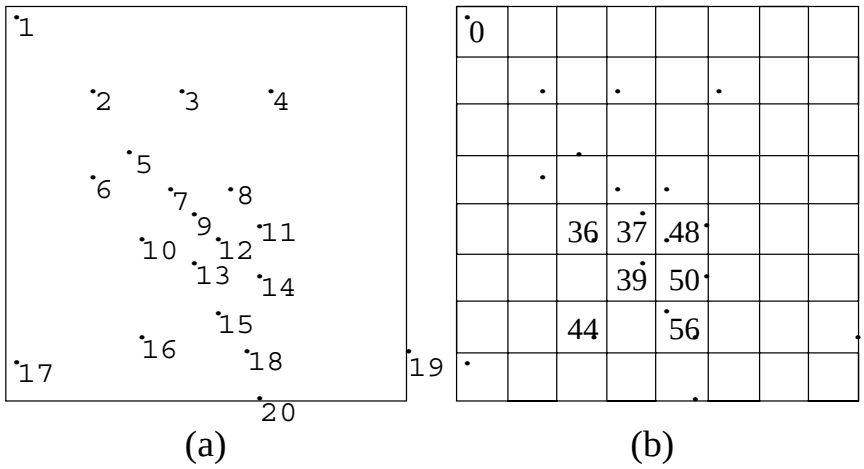


Figure 3: Mapping from a two-dimensional grid to a one-dimensional index

---

Indexing($index$,d,n)

1 **for** $j \longleftarrow 1$ **to** d **do**
2 $\quad unit_j \longleftarrow ((\max_{i=1}^{n} x_{ij}) - (\min_{i=1}^{n} x_{ij}))/2^{l_j}$
3 **for** $i \longleftarrow 1$ **to** n **do**
4 $\quad$ **for** $j \longleftarrow 1$ **to** d **do**
5 $\quad\quad cord_j \longleftarrow x_{ij}/unit_j$
6 $\quad index_i \longleftarrow$ **Interleave**($cord$,d,l)

Figure 4: Indexing algorithm

{ Assume $bits$ represent the number of bits used in each dimension i, and $bits_1 \geq bits_2 \geq bits_3 \ldots bits_{d+1} = 0$; $index$ is an array that stores the resulting index.}

Interleave($cord$,d,$bits$)

1 **for** $i \longleftarrow d$ **downto** 1 **do**
2 $\quad$ **for** $j \longleftarrow 1$ **to** $bits_i - bits_{i+1}$ **do**
3 $\quad\quad$ **for** $k \longleftarrow 1$ **to** i **do**
4 $\quad\quad\quad$ **move** the rightmost jth bit in $cord_k$ into the rightmost bit in $index$
5 $\quad\quad\quad$ **left shift** $index$
6 **return** $index$

Figure 5: Interleave

Experimental Results

In this section we present the time requirements and the quality of partitioning produced by the index-based partitioning schemes and compare them to two frequently used graph partitioning schemes—recursive coordinate bisection and recursive spectral bisection.

Recursive coordinate bisection (RCB) bisects a given graph along the longest dimension into two subgraphs with approximately equal sizes recursively until the expected number of subgraphs is achieved [3, 40]. The bisection is based on finding the median coordinate value along the dimension to be bisected. The computational time requirement is proportional to $O(N \log N)$.

Recursive spectral bisection (RSB) is derived from a graph bisection strategy based on the Fiedler vector (the second eigenvector of the Laplacian matrix of the given graph) [2, 36]. The computational complexity for RSB has been empirically observed to be $O(N\sqrt{N})$, dominated by the Lanczos iterative solver used to find the bisecting eigenvector at every recursive step. Several improvements have been proposed recently to improve on the time as well as quality of the simple spectral bisection method [2, 18, 19, 20]. One way to reduce the computational requirements, while maintaining the quality of partitioning, is based on contracting an original graph to a smaller weighted graph and then applying the eigenvector solver to the smaller graph. Several stages of contraction can be applied, hence this algorithm is popularly known as mulitlevel rescursive spectral bisection (MRSB). Since the cost of RSB is much higher than MRSB and the quality of RSB is worse than MRSB with Kernighan-Lin refinement, we present the experimental results of using the MRSB software provided in the "Chaco" package [21] instead of RSB.

Graph	Graph 1	Graph 2	Graph 3	Graph 4	Graph 5
$\|V\|$	6019	9428	10166	15606	53961
$\|E\|$	17473	59863	30471	45878	353476

Figure 6: Experimental graphs

We applied all the above partitioning schemes to several graphs on a 40-MHz SUN4 with 60 MB memory. Results for five representative graphs in Figure 6 on a SUN4 workstation are presented in Figure 7 and Figure 8. The time required for index-based partitioning is independent of the number of partitions. For large graphs as well as number of partitions, the time requirements are comparable to RCB and one to two orders of magnitude better than MRSB. The quality of partitioning produced by the index-based mapping strategy is comparable to coordinate bisection, but worse than spectral methods. We will show in the next two sections that this mapping procedure is simple to parallelize and incremental in nature. This, along with its low computational requirements, makes it suitable for a number of adaptive and irregular data-parallel applications.

Partitions	Partitioner	Graph 1	Graph 2	Graph 3	Graph 4	Graph 5
4	IBP	620	4998	837	1115	15559
	RCB	533	4789	433	785	15206
	MRSB	312	2934	366	370	9120
8	IBP	1187	7731	1360	1866	24685
	RCB	942	7799	837	1350	21005
	MRSB	477	4602	668	699	14983
16	IBP	1601	10828	2057	2703	34830
	RCB	1562	15148	1449	2254	31753
	MRSB	721	6784	1119	1161	22681
32	IBP	2184	14309	2942	3602	47285
	RCB	2117	22067	2164	3301	57655
	MRSB	1127	9711	1611	1885	31801
64	IBP	3045	19074	4372	5033	64787
	RCB	2902	27139	3251	4502	77313
	MRSB	1838	13269	2451	3024	44526

Figure 7: Quality (in terms of cross edges obtained) of IBP, RCB, and MRSB on different graphs

Partitions	Partitioner	Graph 1	Graph 2	Graph 3	Graph 4	Graph 5
4	IBP	.089	.159	.169	.249	1.009
	RCB	.116	.183	.199	.316	1.033
	MRSB	3.369	8.149	5.369	7.049	48.35
8	IBP	.089	.159	.169	.249	1.009
	RCB	.166	.249	.266	.416	1.483
	MRSB	5.319	13.09	10.97	12.18	69.81
16	IBP	.089	.159	.169	.249	1.009
	RCB	.199	.316	.349	.516	1.899
	MRSB	8.479	17.06	20.08	19.96	93.31
32	IBP	.089	.159	.169	.249	1.009
	RCB	.249	.399	.433	.649	2.333
	MRSB	14.40	26.76	30.73	30.31	120.35
64	IBP	.089	.159	.169	.249	1.009
	RCB	.299	.483	.499	.783	2.733
	MRSB	23.48	33.23	50.67	50.09	155.61

Figure 8: Execution time in seconds for IBP, RCB, and MRSB on a SUN4

4 Parallelization

We model a coarse-grained parallel machine as follows. A coarse-grained machine consists of several processors connected by an interconnection network. Rather than make specific assumptions about the underlying network, we assume a two-level model of computation. The two-level model assumes a fixed cost for an off-processor access independent of the distance between the communicating processors. A unit computation local to a processor has a cost of δ. Communication between processors has a start-up overhead of τ, while the data transfer rate is $\frac{1}{\mu}$. For our complexity analysis we assume that τ and μ are constant and independent of the link congestion and distance between two processors. With new techniques, such as wormhole routing and randomized routing, the distance between communicating processors seems to be less of a determining factor on the amount of time needed to complete the communication. This permits us to use the two-level model and view the underlying interconnection network as a virtual crossbar network connecting the processors. These assumptions closely model the behavior of the CM-5 on which our experimental results are presented. Although our algorithms are analyzed under these assumptions, most of them are architecture-independent and can be efficiently implemented on meshes and hypercubes.

For the rest of the paper, let A_j represent an element A stored in processor j. Hence $A_j[i]$ represents the ith element of an array belonging to the jth processor. We will drop the subscript j whenever it is obvious from the context.

In the following we describe some important primitives used to develop our parallel algorithms.

1. **Sending a Message**
   Assuming no vertex contention, the time taken to send a message from one processor to another is modeled as $\mathrm{O}(\tau + \mu m)$, where m is the size of the message.

2. **Global Reduction**
   Assume that each processor contains V_i. A global reduction computes an associative and commutative operation to produce a result R. The resultant R is stored in all the processors. This operation can be completed in $\tau \log p$. Special hardware is available on the CM-5 for performing this operation with a very small value of τ [5].

3. **Global Concatenation**
   Assume that each processor has n/p elements, where p is the number of processors. Each processor contains a vector $V_i[0 \cdots \frac{n}{p} - 1]$. The global concatenate operation performs a concatenation of the local vector in each of the processors. The resultant vector $R[0 \cdots n-1]$ is stored in all the processors. Assuming that n is larger than p, this operation can be completed in $O(\tau \log p + \phi_1 n)$ time. Special hardware is available on the CM-5 for performing this operation with a small value of ϕ_1 [5].

4. **Complete Exchange**
   The complete exchange primitive performs all-to-all personalized communication with

equal-sized messages. Assuming that t is the total length of all the messages sent out and received at every processor, a complete exchange can be performed in time $p\tau + \mu t$.

5. **Transportation Primitive**
   The transportation primitive performs many-to-many personalized communication with possibly high variance in message size. If the total length of the messages being sent out or received at any processor is bounded by t, the time taken for the communication is $2\mu t$ (+ lower order terms) when $t \geq O(p^2 + p\tau/\mu)$. If the outgoing and incoming traffic bounds are r and c instead, the communication takes time $2\mu(r + c)$ (+ lower order terms) when either $r \geq O(p^2 + p\tau/\mu)$ or $c \geq O(p^2 + p\tau/\mu)$ [34].

6. **Order-Maintaining Load Balance**
   Assume that processor i contains a sorted array $V_i[0 \cdots X_i - 1]$. $(0 \leq i < p)$, where p is the number of processors. Further, a concatenation of all these arrays in ascending order of the processor number is also sorted. We would like to balance the load on each processor such that the global ordering of elements does not change.

   The load-balancing algorithm which maintains the sorted order is given in Figure 9. Steps 2 and 3 calculate the prefix sum and the average number of elements. For ease of presentation we will assume $\overline{X}$ to be an integer. Let

   - prefix sum $Y_k = \sum_{i=0}^{k-1} X_i$ for $k = 1, \ldots, n-1$, and $Y_0 = 0$;
   - average number of elements $\overline{X} = \frac{1}{n}\sum_{i=0}^{n-1} X_i$. For ease of presentation we assume that $\overline{X}$ is an integer.

   Let $G_k[i]$ represent $V_k[i]'s$ corresponding global index, $G_k[i] = Y_k + i$, $0 \leq i \leq X_k - 1$.

   In Step 4 data elements are sent to appropriate destinations. Let $packet_i^k$ contain data elements that should be moved from processor P_k to P_i. Let $lb_i^k = \max\{i\overline{X}, Y_k\}$ and $ub_i^k = \min\{(i+1)\overline{X} - 1, Y_k + X_k - 1\}$, then if $lb_i^k > ub_i^k$, $packet_i^k = \phi$, otherwise $packet_i^k = \{V_k[j] \mid G_k^{-1}[lb_i^k] \leq j \leq G_k^{-1}[ub_i^k]\}$, where $G_k^{-1}[i] = i - Y_k$. The boundaries of these packets can be determined easily by calculating the leftmost processor to which data must be sent (by using a binary search for $G_k[0]$ on $Z[0..p-1]$ on processor k). Since all the data has to be sent to consecutive processors, deriving this for the rest of the processors can be achieved easily.

   The complexity of this algorithm depends on the maximum amount of data to be sent/received from any processor and the underlying communication network. Assuming that the minimum number of elements on any processor is more than $\frac{\overline{X}}{K}$ and the maximum number of elements on any processor is less than $\overline{X}K$, it can easily be shown that the maximum number of number of messages to be sent by each processor is less than or equal to K, and the maximum number of messages to be received by any processor is less than or equal to $K + 1$. Thus, assuming near load balance, i.e., $K \leq 2$, each processor will send and receive a few messages and this operation can be completed in $O(\tau + \mu * \overline{X})$.

For processor P_i, $0 \leq i < p$, **in parallel do**

1 $Z[0..p-1] =$ **Concatenate**(X_i)

2 $Y[k] = \sum_{j=0}^{k-1} Z_[j]$ for $k = 1, 2, ..., p-1$, $Y_0 = 0$

3 $\overline{X} = \frac{\sum_{j=0}^{k-1} Z[j]}{p}$

/* Processor P_k owns data from $Y[k]$ to $Y[k+1]-1$ */

/* After load balance it should have $k\overline{X}$ to $(k+1)\overline{X}$ */

4 Divide the local list into packets and send them to processors from left to right.

5 Receive messages and store them in the appropriate positions in the local array.

Figure 9: Order-Maintaining Load Balance Algorithm.

4.1 Parallel Mapping Algorithm

In this section we describe the parallelization of the mapping algorithm given in the previous section. We assume that vertices of the graph are equally distributed among all the processors. The first step is to compute the indices for all the vertices in the graph and is simple to parallelize. The algorithm is described in Figure 10. Step 1 finds the the local maximum and minimum values. Since the vertices of the graph are equally distributed, this takes the same amount of time. Step 2 finds the global maximum and minimum values in each dimension to form the indices for all vertices, which is the only communication among all processors. The interleave operation (described earlier) in step 4 assigns an index value to each vertex.

Using a sample sort the vertices are then sorted based on their index values. The algorithm (Figure 11) is similar to the randomized sorting algorithms described in [4, 35]. The subroutine **Random_choose** randomly chooses $f \times n$ $(0 < f \leq 1)$ index values from vertices and stores the values in a temporary array, $CHOOSE$. The distribution of the indices in this sample represents with high probability the distribution of the overall set. One can find this sample assuming probabilistic techniques as described in [4]. For the specific application in this paper, the integers correspond to vertices in a physical domain. A potentially smaller fraction is a good representative of the whole mesh (a sample size of 600 vertices for two dimensions, and 1000 vertices are enough for three dimensions for the geometric sampling-based scheme given in [25]). Figure 12 represents the sequence of the input vertices of the graph and the corresponding hash values for 4 processors (each processor has 4 vertices). Figure 13 gives the randomly chosen vertices for the example in Figure 12.

This sample is sorted using a deterministic parallel algorithm (parallel merge sort) and is equally divided among the processors in an ascending fashion (in the order of the processor numbers) (Figure 13). We define the boundary points of a particular processor to be the minimum and the maximum indices that are parts of a particular processor. Boundary points given by partitioning (based on the one-dimensional index) of the small sample is used as an

{processor i owns n_i vertices where $n_i \approx \frac{N}{P}$.}
for each processor i **do in parallel**
  Parallel_Indexing$(hash_i, d, n_i)$
1 **for** $j \longleftarrow 1$ **to** d **do**
        $max_i[j] := \max_{k=1}^{n_i} x_k[j]$
        $min_i[j] := \min_{k=1}^{n_i} x_k[j]$
2 **for** $j \longleftarrow 1$ **to** d **do**
        $MAX[j] :=$ **GLOBAL** $\max_{i=1}^{d} max_i[j]$
        $MIN[j] :=$ **GLOBAL** $\min_{i=1}^{d} min_i[j]$
3 **for** $j \longleftarrow 1$ **to** d **do**
        $unit[j] := (MAX[j] - MIN[j])/2^l$
4 **for** $k \longleftarrow 1$ **to** n_i **do**
    for $j \longleftarrow 1$ **to** d **do**
        $index_i[j] := \frac{x_k[j]}{unit[k]}$
    for $j \longleftarrow 1$ to d **do**
        $hash_i[k] :=$ **Interleave**$(index_i[j], d, l)$

Figure 10: Parallel Indexing Algorithm

for each processor i **do in parallel**
1 **Parallel_indexing**$(hash,d,n_i)$
2 $choose_i[1..fn_i] :=$ **Random_choose**$(hash_i, n_i)$
3 $CHOOSE[1..fN] :=$ **Parallel_merge_sort**$(choose_i[k], fn_i)$
4 $C\_BOUND[0..P] :=$ **Find_boundary**(fN, P)
5 **for** $k \longleftarrow 1$ **to** n_i **do**
        $proc :=$ **Binary_search**$(hash_i[k], C\_BOUND)$
        Add $hash_i[k]$ to $send\_list[proc]$
6 **All-to-Many** communication using $send\_list$ and store in $receive\_list_i$ of size nr_i
7 **Sort**$(nr_i, receive\_list_i, perm_i)$
8 $R\_BOUND[0..P] :=$ **Find_boundary**(N, P)
9 Perform a **Order Maintaining Load Balance** on $receive\_list_i$

Figure 11: Parallel Mapping Algorithm

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
index	0	3	7	19	12	11	15	26	37	36	48	48	39	50	56	44	42	56	61	58

Figure 12: One-dimensional indices

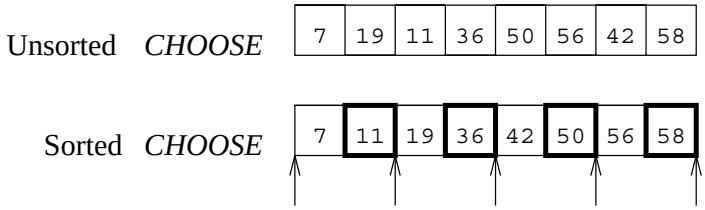


Figure 13: Sorted list of randomly chosen indices

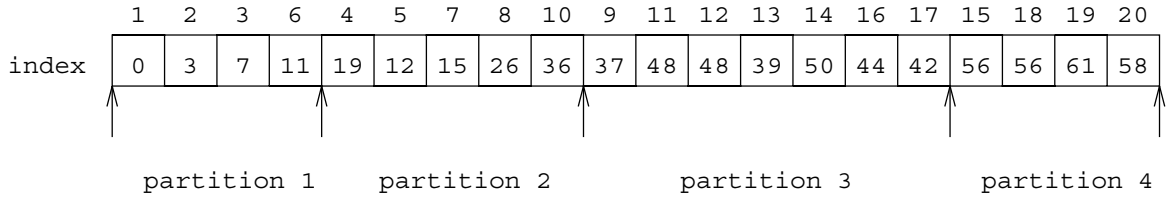


Figure 14: Move vertices according to the index values of coarse boundaries

approximation to the partitioning points (for P partitions) for all data items. For example, in Figure 14, we have $CHOOSE[C\_BOUND[1]] = 11$, $CHOOSE[C\_BOUND[2]] = 36$, $CHOOSE[C\_BOUND[3]] = 50$, $CHOOSE[C\_BOUND[4]] = 58$.

This information is used to shuffle the data items into P processors. A local sort is performed on the local vertices. This is followed by an Order-Maintaining Load Balance to ensure each processor has equal number of vertices. Figure 15 illustrates the final positions of different vertices and corresponding partitions.

Analysis

Parallel_indexing requires $O(\frac{dN}{P})$ time to assign index values to all vertices, and $2d$ global reduction operations to find the maximum and minimum values in each dimension. **Random_choose** takes $\frac{f \times N}{P}$ to choose vertices randomly. The complexity of the sorting algorithm used in Step 3 (Figure 11) can be given by

$$O(\frac{fN}{P} \log \frac{fN}{P} \ + \ \log^2 \frac{fN}{P}(\tau + \frac{\mu fN}{P} + \frac{fN}{P})),$$

where f represents the fraction of elements chosen by Step 2, the first term corresponds to a local sort, and the second term corresponds to the merging phase. Step 5 requires $O(\frac{N}{P} \log \frac{N}{P})$ amount of time. Let R represent the maximum number of indices received by any processor. The complexity of Step 6 is given by $O(\tau P \ + \ \mu R)$, and the complexity of

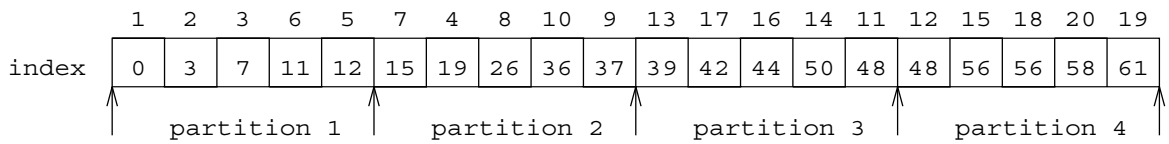


Figure 15: Sort the vertices in each coarse partition and set the refining boundaries

Step 7 is given by $O(R \log R)$. It has been shown that the value of R is $\frac{N}{P}(1+\varepsilon)$ if f is chosen properly (f is a function of ε) [4]. Further, the value of f required for the fraction value of ε ($\varepsilon < 0.05$) is reasonably small for large values of N. Under such conditions the time required for Step 7 would be bounded by $O(\frac{N \log N}{P})$.

Assuming that each processor needs to send vertices only to the left or the right, the time complexity of Step 9 can be given by $O(\tau + \mu \frac{N}{P})$. In the worst case it may require the use of the Transportation Primitive and is bounded by $O(\tau P + \mu R)$.

Experimental Results

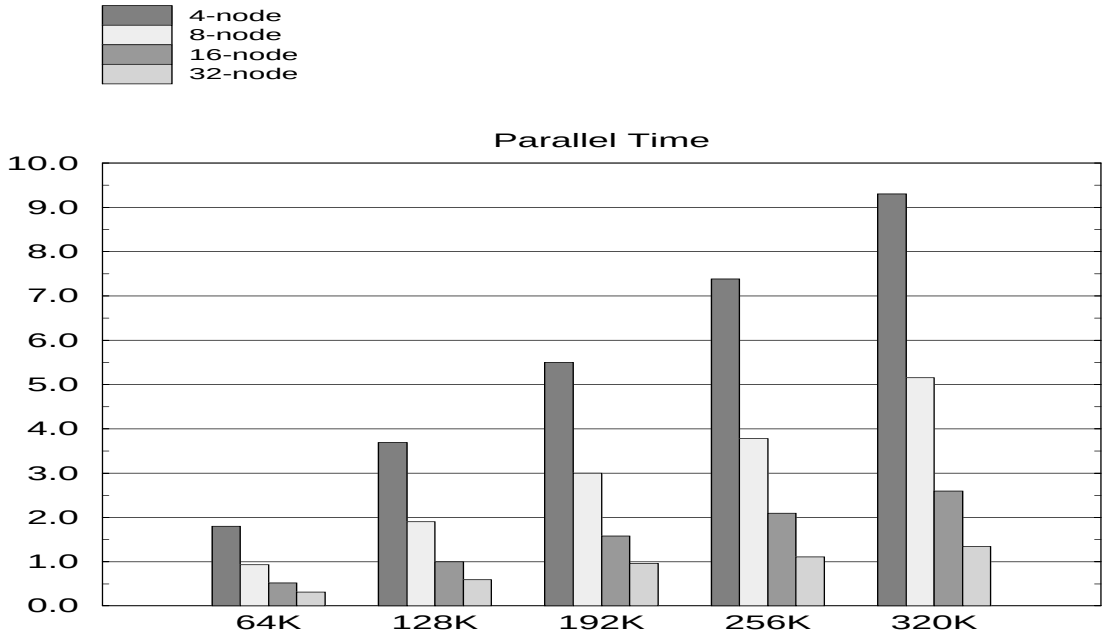


Figure 16: Timing for parallel mapping algorithm on 4, 8, 16, and 32 processors

To study the cost of parallelization for different values of N, the coordinate data was generated randomly. The algorithm was implemented in the C language with the CMMD communication library and conducted on the 32-node CM-5 available at NPAC at Syracuse University. The time shown is the median of 11 executions of the same data set. The time represents the "busy time" given by the CMMD library calls.

The computational requirements for different phases of the algorithm for Graph 4 (15606 vertices on 32 nodes) are presented in steps given in Figure 17. This demonstrates the tradeoff between the time spent on sorting the sample and the time spent in the local sort. Larger samples imply more time spent on the sample sort and less time spent on the local sorting, as the maximum number of items on a given processor is close to the average number of items ($\frac{N}{P}$). Choosing the optimal fraction would in general depend on this tradeoff. However, choosing the optimal fraction does not seem to be very important. Choosing a non-optimal fraction (close to the optimal) does not affect the total time by more than 10%.

Figure 18 gives the execution time for Graph 1, Graph 3, and Graph 4 for a 1-node CM-5. The time for different number of processors (and corresponding number of parti-

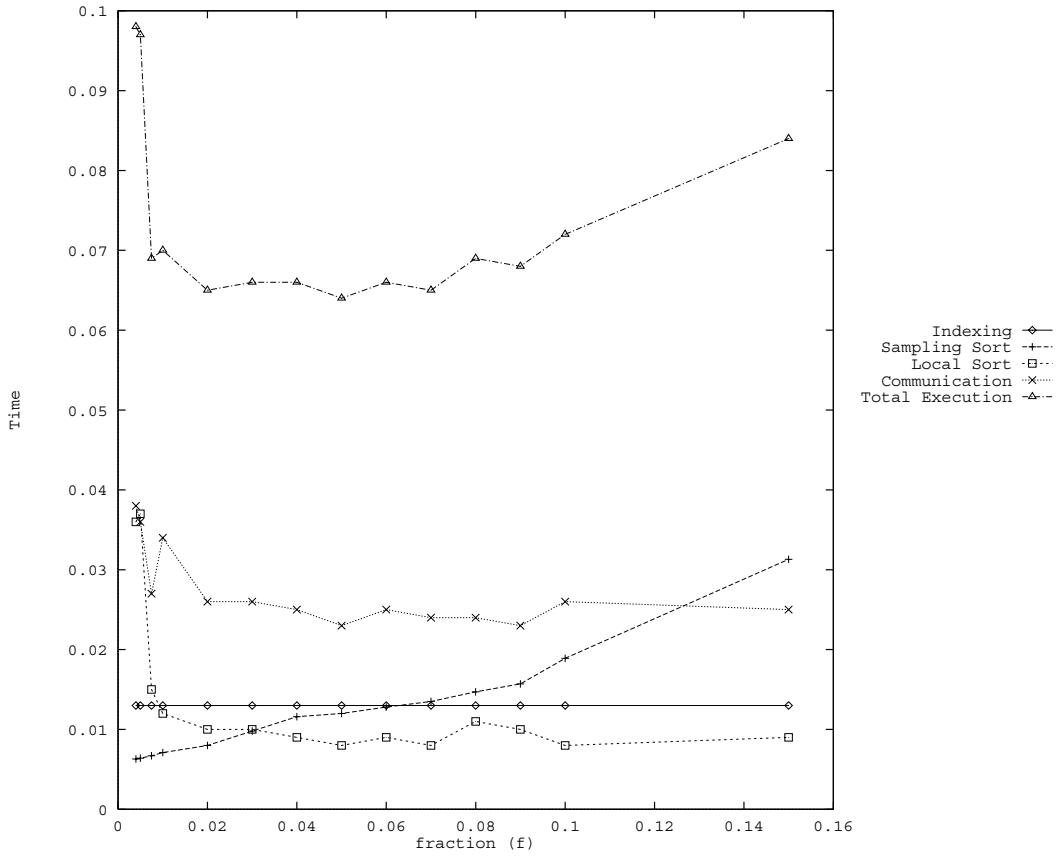


Figure 17: Partitioning times for a $\mid V \mid = 15606$ graph into 32 partitions on the CM-5 for different values of fraction f

tions) are given in Figure 19. Even for these small graphs a speedup of up to 9 is achieved on a 32-node CM-5. The time is dominated by the communication phase (Step 6 in Figure 11) of the sorting algorithm. It corresponds to 41%, 34%, and 30% of the total time for Graph 1, Graph 3, and Graph 4 for 32 partritions on 32-node CM-5, respectively. Since every processor communicates with every other processor, a major portion of this time is the setup overhead for small data sets. This communication time corresponds to moving the vertices to the appropriate destination. This is the minimum time required for remapping all the vertices based on the index-based partitioning. As the data size increases this fraction reduces significantly. Figure 19 also gives the parallel execution time for the RSB-based partitioner [22] (available in the CMSSL library) on the CM-5. The time is provided for cases when only scalar units were used as well as when vector units were used.[1] The performance of IBP-based methods (when uses the SPARC chip only) is about two to three orders of magnitudes faster than RSB-implemented which in the CMSSL library, depending on the size of the graph, when vector units are not used. Even when vector units are used, the performance is approximately two magnitudes better.

Graph 1	Graph 3	Graph 4
.276	.611	.880

Figure 18: Sequential execution time in seconds for IBP on 1-node CM-5

	Partitioner	4-node	8-node	16-node	32-node
Graph 1	IBP	.151	.094	.068	.051
	RSB	10.08	13.084	15.31	17.38
	RSB/VU	4.284	5.524	6.396	7.218
Graph 3	IBP	.254	.159	.103	.069
	RSB	14.00	22.20	28.39	33.95
	RSB/VU	5.563	9.076	10.96	13.41
Graph 4	IBP	.356	.211	.121	.096
	RSB	27.40	40.57	50.96	63.61
	RSB/VU	7.860	11.98	15.22	18.08

Figure 19: Parallel execution time in seconds for IBP, and RSB in CMSSL on CM-5

[1]The CMSSL functions are called using CM Fortran. The input graphs have to be provided in a different format (in terms of finite elements). The number of finite elements for Graph 1, Graph 3, and Graph 4 are 11451, 20232, and 30269, respectively.

4.2 Remapping Algorithm

For many applications, such as adaptive meshes, new vertices are added to the computational graph. This is typically done in a localized area to study the numerical behavior more precisely. These refinements are based on the solution of the previous phase and are available only at runtime. During a typical simulation, vertices may be added in a particular portion, only to be removed after a few phases. The following discussion is limited to the case when vertices are added to the computational graph. Deletion of vertices can be done similarly.

Remapping requires calculating the shuffled row-major indices of the new vertices, which must be combined with the indices of the previous phase. Since the previous mapping is available, this corresponds to adding an unsorted list of integers (corresponding to the indices of the new vertices that are added) to a sorted list (corresponding to the indices of the old vertices). Let A represent a sorted list of N integers, and let B represent an unsorted list of M integers. A simple sequential approach for merging list B into list A is to sort B, followed by merging the two sorted lists. The complexity of this approach is $O(M \log M + (M + N))$. For $M < O(\frac{N}{\log N})$, the complexity is $O(N)$.

A simple parallel algorithm (Figure 20) can be used for solving this problem under the assumption $M \ll N$. It assumes that list A is already sorted and distributed equally among all the processors, which corresponds to the partitioning of the previous phase. The new vertices added/deleted in the new phase are assumed to be equally distributed among all the processors for the merging algorithm. However, this is not going to be the case in general. In fact, for most practical cases the incremental vertices are added in localized portions, which would typically correspond to all the new elements belonging to a few processors. In such cases a simple load-balancing scheme can be effectively applied [31]. In most cases the cost of this load-balancing scheme is nominal compared to the cost of merging for most cases.

Analysis

The analysis provided in this section corresponds to the near worst-case scenario for this algorithm. An average case is hard to define and depends on the application to be solved. The worst-case scenario for this algorithm corresponds to one processor receiving all the merging elements from all the processors. Step 2 takes $\frac{N}{P} \log P$ amount of time. The time taken for Step 3 depends on the number of packets generated and the size of the packets. The all-to-many communication algorithm has been described in Section 3; the worst-case total cost of Step 3 is $O(p\tau + \mu M)$. Steps 4 and 5 take $O(m_i \log m_i + \frac{N}{P} + m_i)$ amount of time where m_i is the number of elements to be inserted. For the worst-case scenario this corresponds to $O(M \log M + \frac{N}{P})$, thus the total cost of Steps 1 through Step 5 is $O(\frac{M}{P} \log P + \tau P + \mu M + M \log M + \frac{N}{P})$. When N and M are large compared to P, and when $M \log M < \frac{N}{P}$ (the incremental data is much less than the data on one processor), Step 6 will be reduced to shifting to either the left or the right neighbors and can be completed in $O(\tau + \mu\gamma)$ where γ is the maximum amount of data sent/received by any processor.

{ 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 : **For** $k \longleftarrow 0$ **to** $p-1$ **do**
$send\_list[k] := nil$
Step 2 : **For** $k \longleftarrow 1$ **to** m_i **do**
$proc :=$ **Binary_search**$(B_i[k], Bound)$
Add $B_i[k]$ to $send\_list[proc]$
Step 3 : **All-to-Many** communication using $send\_list$
Step 4 : Sort all B_i the elements received in Step 3 and call it C_i
Step 5 : Merge list A_i and C_i
Step 6 : Perform **Order-Maintaining Load Balance** on A

Figure 20: Simple Merging Algorithm

Experimental Results

We generated two sets of data for performing our experiments.

- Data Set 1: Each processor generated a random number (uniform distribution) of elements such that the index values were within the boundaries of each processor. This represents the near best case for the algorithm and corresponds to the case when all processors generate approximate equal number of mesh elements.

- Data Set 2: One processor generated all the elements (M) such that the elements will reside within the boundaries of each processor. This case is followed by a load-balancing step in which data was distributed to all processors equally ($\frac{M}{P}$), which represents the worst case for the algorithm, as all the data would be sent to one processor.

Experiments were conducted for variable fractions of additional vertices as well as number of vertices. In the following we discuss only representative examples. The results in Figure 21 show the time requirements for Data Set 1 when 10% additional vertices were added. The results show that for large addition the algorithm parallelizes very well. The cost on 32 processors for 10% additional vertices (Figure 21) is approximately 0.31 seconds for 320,000 vertices. The corresponding time for mapping 320,000 vertices from scratch on 32 processors is 1.34 seconds. This shows that the incremental algorithm can be used to reduce significantly the time for repartitioning.

For Data Set 2, the algorithm does not parallelize very well with the number of processors unless the fraction is small (less than 1%) and the number of vertices is large (greater than 10,000). This is because all the messages are received by one processor. Further, all the data

is sorted on one processor in Step 4 of the remapping algorithm. Figures 22 and 23 give the timing for Data Set 2 when the number of additional vertices are 1% and 5%, respectively.

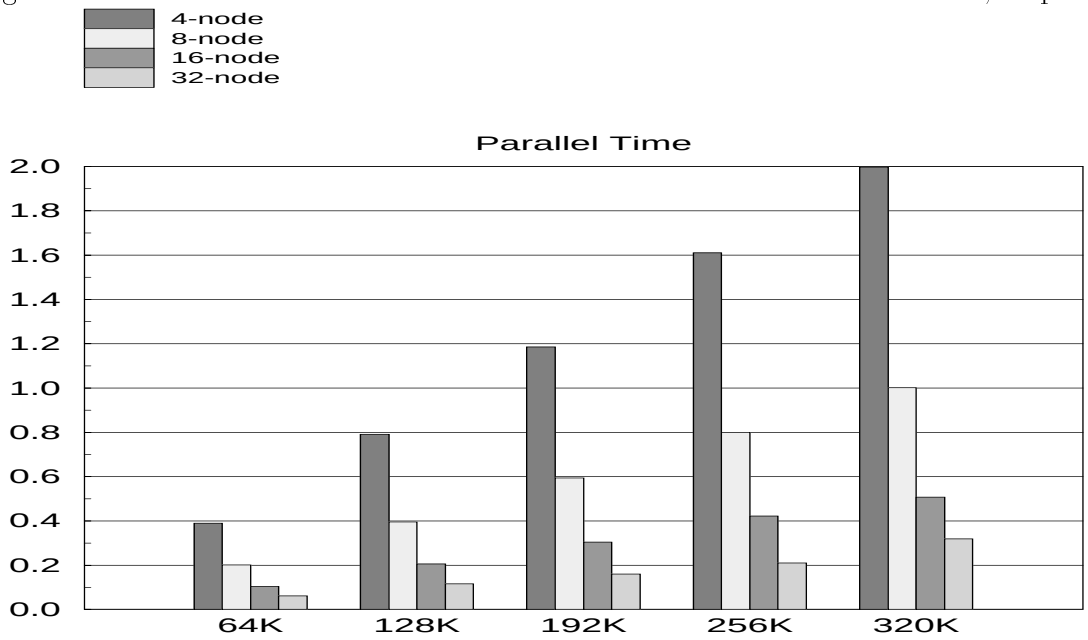


Figure 21: Time for 10% additional vertices on 4, 8, 16, 32 processors (Data Set 1)

5 Conclusions

In this paper we proposed a simple index-based mapping algorithm for mapping computational graphs. We have shown that these methods can

1. provide good solutions with a relatively low cost,
2. be parallelized, and
3. can potentially be used for problems that are incremental in nature.

The above properties should cause these algorithms to be of great importance in the parallelization of applications for which the computational structure changes frequently and/or incrementally.

The performance of the incremental mapping algorithm depends on the type of data generated. There is a big gap between the performance of our algorithm for different types of data sets. We have recently developed several algorithms for improving the worst-case performance of the merging algorithm [27]. We have also investigated the remapping of adaptive applications in which the vertices of the computational graph move reflecting perturbations in the physical domain [27].

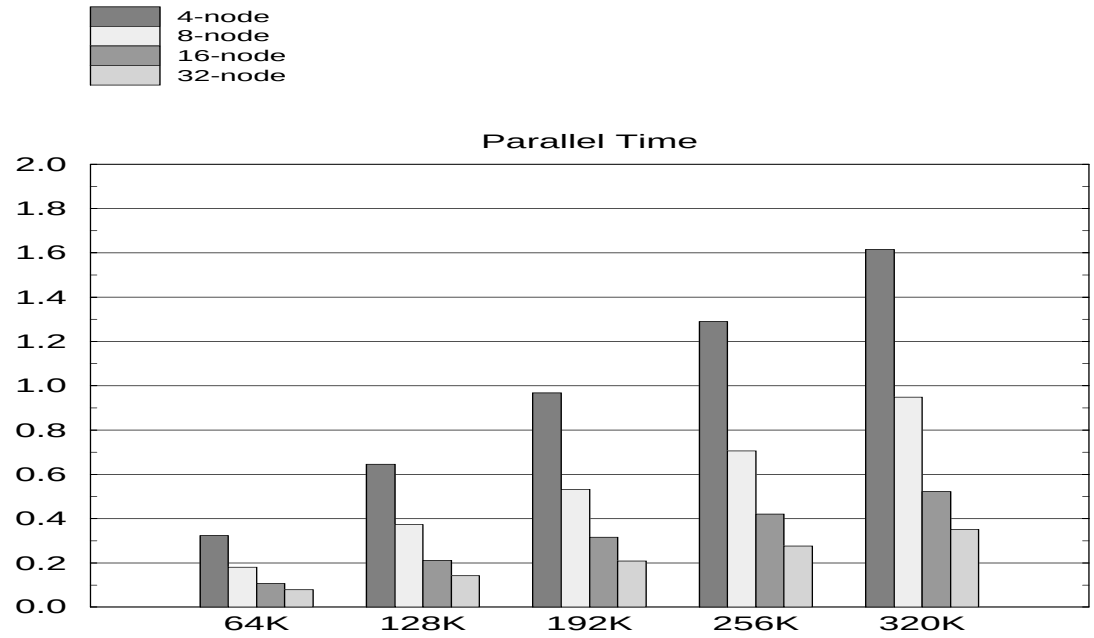


Figure 22: Time for 1% additional vertices on 4, 8, 16, 32 processors (Data Set 2)

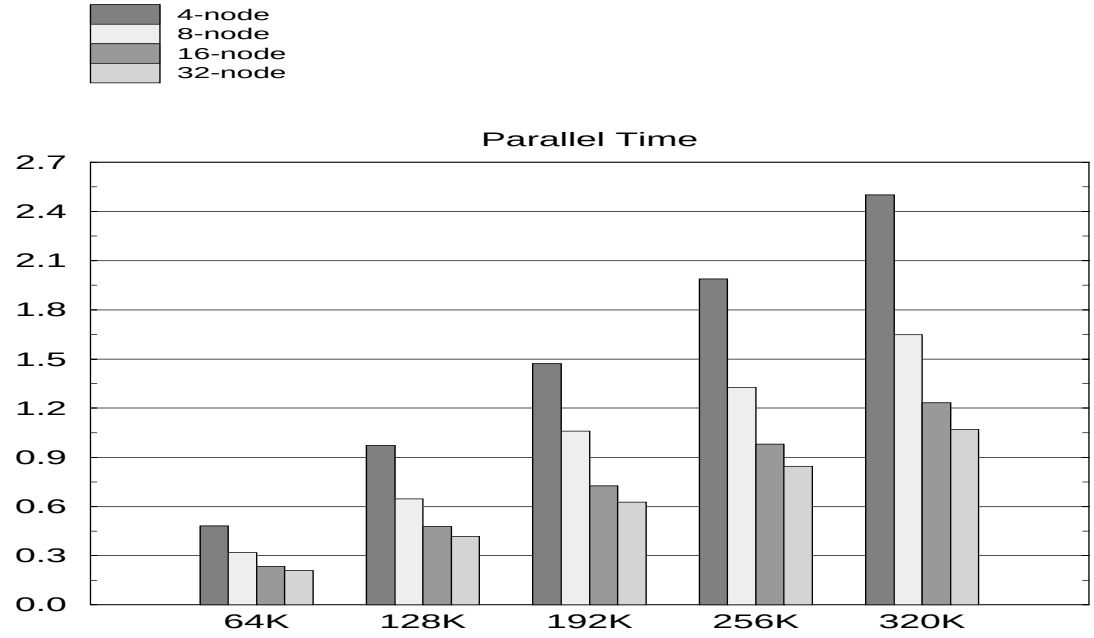


Figure 23: Time for 5% additional vertices on 4, 8, 16, 32 processors (Data Set 2)

Acknowledgments

The authors would like to thank Horst Simon for providing the meshes and the Recursive Spectral Bisection software. We would like to thank Bruce Hendrickson and Robert Leland at Sandia National Laboratories for the "Chaco" graph-partitioning software. We would like to thank Elaine Weinman for editing this paper.

This work was supported in part by NSF under CCR-9110812, NSF under ASC-9213821, and ARPA under contract #DABT63-91-C-0028, and NAG-1485. The contents do not necessarily reflect the position or the policy of the United States government and no official endorsement should be inferred.

References

[1] I. Angus, G. Fox, J. Kim, and D. Walker. *Solving Problems on Concurrent Processors*, vol. 2. Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] S. Barnard and H. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Proceedings of the 6th SIAM Conference*, pp. 711–718, 1993.

[3] M. Berger and S. Bokhari. A Partitioning Strategy for Nonuniform Problems on Multiprocessors. *IEEE Trans.*, 36:570–580, May 1987.

[4] M. Bolorforoush, N. Coleman, D. Quammen, and P. Wang. A Parallel Randomized Sorting Algorithm. *Proceedings of the International Conference on Parallel Processing*, vol. 3, pp. 293–296, 1992.

[5] Z. Bozkus, S. Ranka, and G. Fox. Benchmarking the CM-5 Multicomputer. *Proceedings of the Frontiers of Massively Parallel Computation*, 1992.

[6] B. Brooks, R. Bruccoleri, B. Olafson, D.J. States, S. Swaminathan, and M. Karplus. A Program for Macromolecular Energy, Minimization, and Dynamics Calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[7] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and J. Saltz. Software Support for Irregular and Loosely Synchronous Problems. *Proceedings of the Conference on High Performance Computing for Flight Vehicles*, 1992.

[8] A. Choudhary, G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, S. Ranka, and C. Tseng. Compiling Fortran 77D and 90D for MIMD Distributed-Memory Machines. *Proceedings of the Frontiers of Massively Parallel Computation*, 1992.

[9] P. Coddington and C. Baillie. Cluster Algorithms for Spin Models on MIMD Parallel Computers. *Proceedings of the 5th Distributed Memory Computing Conference*, pp. 384–388, Charleston, SC, April 1990.

[10] N. Copty, S. Ranka, G. Fox, and R. Shankar. SIMD and MIMD region growing algorithms on the CM-5. *International Conference on Parallel Processing*, 1994.

[11] F. Ercal. *Heuristic Approaches to Task Allocation for Parallel Computing.* Ph.D. thesis, Ohio State University, 1988.

[12] G. Fox. *Graphical Approach to Load Balancing and Sparse Matrix Vector Multiplication on the Hypercube*, 1988. Ed. M. Schultz, Springer-Verlag, Berlin.

[13] G. Fox. The Architecture of Problems and Portable Parallel Software Systems. Technical report, Syracuse University, July 1991.

[14] G. Fox and W. Furmanski. Load Balancing Loosely Synchronous Problems with a Neural Network, 1988.

[15] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, vol. 1. Prentice Hall, Englewood Cliffs, NJ, 1988.

[16] M. Garey and D. Johnson. Computers and Intractability, pp. 209–210, Freeman, New York, 1979.

[17] J. Gilbert, G. Miller, and S. Teng. A Geometric Approach to Mesh Partitioning: Implementation and Experiments. Technical report, Xerox Palo Alto Research Center, 1992.

[18] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. Technical report, Sandia National Laboratories, Albuquerque, NM 87185, 1993.

[19] B. Hendrickson and R. Leland. An Improved Spectral Load Balancing Method. *Proceedings of 6th SIAM Conference*, pp. 953–961, 1993.

[20] B. Hendrickson and R. Leland. Multidimensional Spectral Load Balancing. Technical report, Sandia National Laboratories, Albuquerque, NM 87185, 1993.

[21] B. Hendrickson and R. Leland. The Chaco User's Guide, Version 1.0. Technical report, Sandia National Laboratories, October 1993.

[22] Z. Johan, K. Mathur, S. Johnsson, and T. Hughes. An efficient communication strategy for finite element methods on c the Connection Machine CM-5 system. Technical report , TMC.

[23] H. Maini, K. Mehrotra, C. Mohan, and S. Ranka. Genetic Algorithms for Graph Partitioning and Incremental Graph Partitioning. *Proceedings of Supercomputing '94*, November 1994.

[24] N. Mansour. *Physical Optimization Algorithms for Mapping Data to Distributed-Memory Multiprocessors.* Ph.D. thesis, Syracuse University, NY, 1993.

[25] G. Miller, S. Teng, W. Thurston, and S. Vavasis. Automatic Mesh Partitioning. *Proceedings of the 1992 Workshop on Sparse Matrix Computations: Graph Theory Issues and Algorithms*, Institute for Mathematics and its Applications, 1992.

[26] B. Nour-Omid, A. Raefsky, and G. Lyzenga. Solving Finite Element Equations on Current Computers. *Parallel Computations and Their Impact on Mechanics*, pp. 209–227, 1986.

[27] C. Ou and S. Ranka. Parallel Remapping Algorithms for Adaptive Problems. *Frontiers '95*, pp. 367–374, 1995.

[28] C. Ou, S. Ranka, and G. Fox. Fast Mapping and Remapping Algorithm for Irregular and Adaptive Problems. *Proceedings of the 1993 International Conference on Parallel and Distributed Systems*, pp. 279–283, Taipei, Taiwan, December 1993.

[29] A. Pothen, H. Simon, and K. Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM Journal of Matrix Analysis and Application*, 11:430–352, July 1990.

[30] J. Quirk. *An Adaptive Grid Algorithm for Computational Shock Hydrodynamics*. Ph.D. thesis, Cranfield Institute of Technology, United Kingdom, 1991.

[31] S. Ranka, Y. Won, and S. Sahni. Programming a Hypercube Multicomputer. *IEEE Software*, pp. 69–77, September 1988.

[32] R. Shankar and S. Ranka. Hypercube algorithms for quadtree operations. *Journal of Pattern Recognition*, pp. 741–747, September 1992.

[33] R. Shankar and S. Ranka. Computer Vision Algorithms for Sparse images. *Journal of Pattern Recognition*, 26:1511–1519, October 1993.

[34] R. Shankar, S. Ranka, and K. Alsabti. Many-to-Many Personalized Communication with Bounded Traffic. *Frontiers '95*, pp. 20–27, 1995.

[35] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[36] H. Simon. Partitioning of Unstructured Mesh Problems for Parallel Processing. *Proceedings of the Conference on Parallel Methods on Large Scale Structural Analysis and Physics Applications*. Permagon Press, 1991.

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

[38] G. Warren, W. Anderson, J. Thomas, and T. Roberts. Grid Convergence for Adaptive Methods. *Proceedings of the AIAA 10th Computational Fluid Dynamics Conference*, page 1591, June 1991.

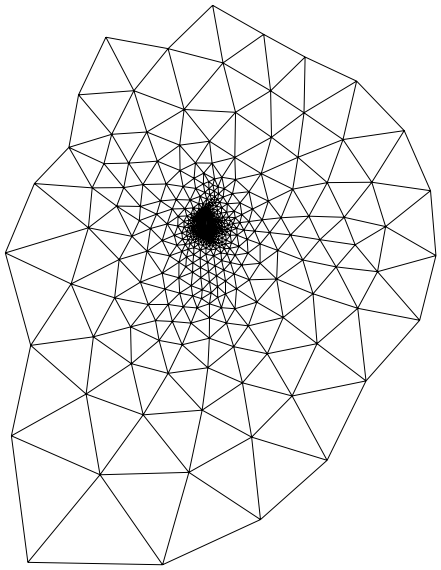
[39] M. Warren and J. Salmon. Astrophysical N-body Simulations Using Hierarchical Tree Data Structure. *Proceedings Supercomputing '92*, Minneapolis, November 1992.

[40] R. Williams. *DIME: Distributed Irregular Mesh Enviroment.* California Institute of Technology, February 1990.

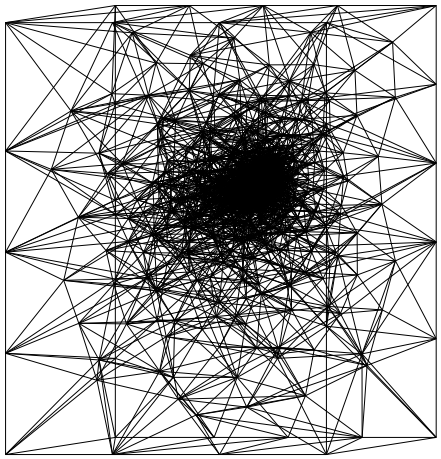
[41] R. Williams. Performance of Dynamic Load-Balancing Algorithm for Unstructured Mesh Calculations. *Concurrency*, 3:457–481, 1991.

Appendix A: Computational Graphs

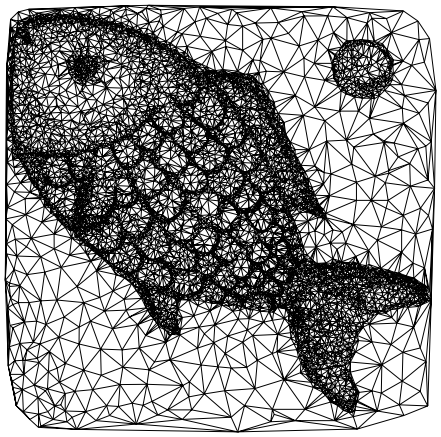
Four of the Five Graphs used for our experiments (Graph 5 is similar to Graph 2 but with larger number of nodes and edges)



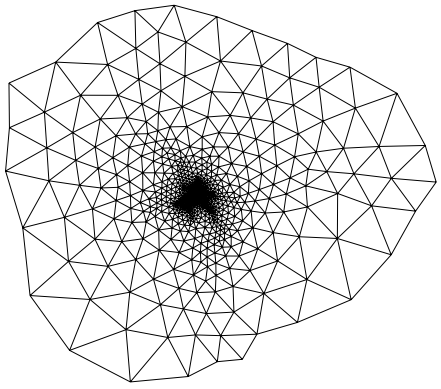
Graph 1



Graph 2



Graph 3



Graph 4