Parallel Remapping Algorithms for Adaptive Problems[1]

Chao-Wei Ou and Sanjay Ranka

School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

Email: *cwou@top.syr.edu, ranka@top.cis.syr.edu*
Phone: (315) 443-4890, (315) 443-4457
FAX: (315) 443-1122

[1]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.

Abstract

In this paper we present fast parallel algorithms for remapping a class of irregular and adaptive problems on coarse-grained distributed memory machines. We show that the remapping of these applications, using simple index-based mapping algorithm, can be reduced to sorting a nearly sorted list of integers or merging an unsorted list of integers with a sorted list of integers. By using the algorithms we have developed, the remapping of these problems can be achieved at a fraction of the cost of mapping from scratch. Experimental results are presented on the CM-5.

1 Introduction

The key problem in efficiently executing data parallel applications is that of partitioning the data among the processors such that the computation load on each node is balanced, while communication is minimized. Partitioning such applications can be posed as a graph-partitioning problem. The nodes of a graph constitute a set of computations that can be executed concurrently, and the edges comprise the interaction between the various computations. Graph-partitioning problems belong to the class of NP-complete problems [9]; hence exact solutions are computationally intractable for large problems. However, good suboptimal solutions are sufficient for effective parallelization of most applications. There are a number of partitioning algorithms available in the literature [1, 10, 14, 18, 19, 28]. This list is by no means complete.

In a large number of such problems the computational structure (or dependencies) can be constructed only during execution [5]. For such cases these graphs must be constructed at runtime; thus it is important that the partitioning of data be done at runtime. Achieving this in parallel is clearly necessary, else partitioning itself would become a bottleneck.

This paper is focused on a subclass of applications in which the computational graph is such that the vertices correspond to two- or three-dimensional coordinates, and the interaction between computations is limited to vertices that are physically proximate. Examples of such applications include finite element calculations [16], molecular dynamics [4], particle dynamics [23], particle-in-a-cell [13, 27], region growing [7], and statistical physics [6]. A list of other such applications is given in [5]. For these applications, partitioning can be achieved by exploiting the above property. Essentially, proximate points are clustered together and form a partition such that the number of points attached to each partition are approximately equal. Most of the interactions are local and the amount of interprocessor communication is low if proximate points are clustered together. Many such algorithms have been described in the literature, including recursive co-ordinate bisection [28], and inertial bisection [11]. We have discussed an index-based indexing scheme in [25] and shown that it produces good mappings for computational structures satisfying the above property.

The index-based transformation (to be described later) converts a two-dimensional (or three-dimensional) coordinate corresponding to a particular node of the computational graph to a one-dimensional index (an integer). The index has the property that proximity in two (or three) dimensions is generally maintained. This reduces the problem of mapping to one of sorting a list of integers. Sorting in parallel is a well-studied problem in the literature. Several algorithms are available for sorting a list of integers [2, 26]. We have shown that a sample-based sorting scheme can be efficiently used for performing the mapping by using an index-based method [18].

For a large class of irregular and adaptive data parallel applications [5], the computational structure changes from one phase to another in an incremental fashion. Thus, the partitioning information of the previous phase can be effectively utilized to give the partitioning for a new phase. The changes are typically gradual, reflecting adiabatic changes in the physical domain, or large-scale, reflecting additions to a data structure. Molecular dynamics applications often exhibit the former behavior because interactions between particles are implemented by neighboring lists that change as the atoms move [4]. Adaptive PDE solvers are examples of the second behavior. Other examples with which we are familiar include some vision algorithms, including region-growing and labeling [7], statistical physics simulations near critical points and the particle-sorting phase of a direct Monte Carlo simulation [8]. The key problem in efficient parallelization of these applications is reacting quickly to minor modifications in the data structure. The physical and numerical properties of these algorithms typically guarantee that large-scale restructuring of data is needed infrequently. Thus, for effective parallelization, the partitioning of the graph needs to be updated as the graph changes over time. The following scenarios may arise (Figure 1):

- Perturbation: All the coordinates may perturb (within some small distance), e.g., particle-dynamics problems [24].
- Node Additions: New points may be added and/or old points deleted. This happens in the case of adaptive grids [5].

One option is to repartition the new graph without using previous information. In this paper, we show that this can be done in an incremental fashion with a much lower cost. Using the index-based mapping scheme, the remapping for perturbation can be reduced to sorting a nearly sorted list. For the case of additions, the remapping algorithms can be reduced to merging a list of unsorted integers in a sorted list. In this paper we study the parallelization of these two problems. We have developed several parallel algorithms for the above problems; each of these algorithms has the best "worst case" performance for different ranges of parameters (the size of the list of integers m, the size of sorted list n, and the number of processors p, etc.). Experimental results of these algorithms, provided on a 32-node CM-5, show that they can be effectively used for incremental remapping at a fraction of the cost of mapping.

The rest of the paper is organized as follows. Section 2 gives a brief outline of the graph-partitioning problem and the index-based approach for performing the partitioning. Section 3 gives the important features of the CM-5. Section 4 describes the important parallel primitives required for the remapping algorithms in Section 5. These primitives provide a level of architectural independence for our algorithms. Section 6 describes the results for a number of data sets on the CM-5 for the different algorithms. We present our conclusions in Section 7.

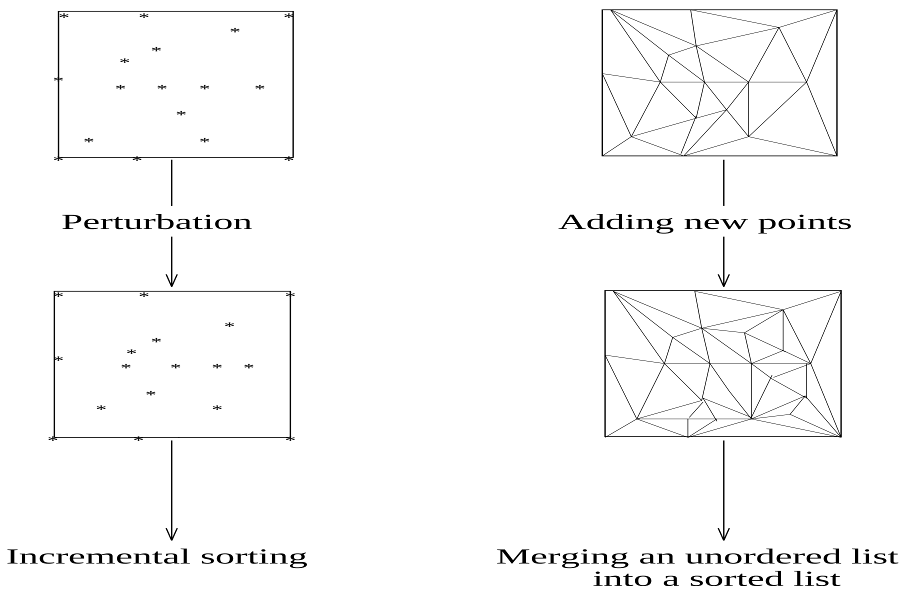


Figure 1: Incremental Aspects

2 Graph Partitioning

Consider a graph $G = (V, E)$, where V represents a set of vertices, E represents a set of undirected edges, the number of vertices is given by $n = |V|$, and the number of edges is given by $e = |E|$. 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, i.e., $B(q) = \{v \in V : M(v) = q\}$. For graphs representing the computational structure of a physical domain, each vertex $v_i \in V$, $1 \leq i \leq n$ corresponds to a physical coordinate in a d-dimensional space $(x_{i_1}, x_{i_2}, \ldots, x_{i_d})$, and each edge is an ordered pair (v_{i_1}, v_{i_2}). For such graphs, these edges connect physically proximate vertices.

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

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

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

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

We would like to make an assignment such that the time spent by every node is minimized, i.e., $\min_q \ (W(q) + \beta C(q))$, where β represents the cost of unit computation/cost of unit communication on a machine. However, for most cases the value $\sum\limits_q C(q)$ is minimized. This is because most programs perform local computation followed by collective communication of non-local data. This requires the computation loads to be as close to balanced as possible, i.e., $w(0) \approx w(1) \approx w(2) \ldots \approx w(p-1)$. This removes the first terms from the minimization function. Further, minimization is replaced by summation because that makes it a continuous function. Since many of the methods proposed in the literature have been gradient descent (or require quadratic minimization function), this approximation makes graph-partitioning problems more amenable to these methods.

2.1 Index-Based Partitioning

The computational graphs which we consider in this paper assume that most interactions occur between points that are physically proximate in two or more dimensions. Row-major indexing and shuffled row-major indexing are two of the several ways of indexing pixels in a two-dimensional grid. These two indexing schemes are shown in Figure 2 (a) and Figure 2 (b) for a graph in which the set of vertices are arranged in a grid of size 8×8. Row-major indexing orders vertices such that if two points are along the same row, then their indices are close to each other. However, the same property is not maintained along the other dimension. On the other hand, shuffled row-major indexing maintains the above property along both dimensions. This indexing scheme can be generalized to n-dimensions and used to convert an n-dimensional index into one-dimensional index such that proximity in the n-dimensions is generally maintained. Index-based algorithms for partitioning graphs have been described in [18]. An IBP algorithm includes three phases—indexing, sorting, and coloring. The indexing scheme is based on converting an N-dimensional coordinate into a one-dimensional index such that proximity in the multi-dimensional space is maintained.

The shuffled row-major index can be easily derived by interleaving the indices. A simple example of interleaving indices is as follows. Suppose $index_1 = 001$, $index_2 = 010$, and $index_3 = 110$. Then the interleaved index would be 001011100. In the above case the number of bits in each dimension are equal. This could easily be generalized to cases where the sizes are different. For example, if $index_1 = 101$, $index_2 = 01$, and $index_3 = 0$, then 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. When the bits of a particular dimension are no longer available, that dimension is not considered.

Another indexing scheme which maintains proximity in multiple dimensions is based on Hilbert space filling curves (Figure 2 (c)). We have performed experiments with this indexing and the quality of partition obtained is similar to the shuffled row-major indexing. However, most of the algorithms discussed in this paper are in general, independent of the indexing method used.

After indexing is done, an efficient sorting algorithm can be applied to sort these vertices

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)

00	03	04	05	58	59	60	63
01	02	07	06	57	56	61	62
14	13	08	09	54	55	50	49
15	12	11	10	53	52	51	48
16	17	30	31	32	33	46	47
19	18	29	28	35	34	45	44
20	23	24	27	36	39	40	43
21	22	25	26	37	38	41	42

(c)

Figure 2: Different Indexing Schemes for an 8×8 image: (a) Row-Major, (b) Shuffled Row-Major, and (c) Hilbert Space Filling Curve.

according to their indices. Finally, this sorted list is divided into P equal sublists. Figure 2 (c) shows the nodes of a computational graph and corresponding shuffled row-major indices (Figure 2 (d)). A partitioning can be achieved by sorting the list of indices and dividing it into equal parts. We have shown that the quality of the solutions produced using these methods are comparable to other coordinate-based partitioning schemes for a large number of graphs derived from actual applications [18]. Since these methods do not directly utilize the edge information available, the number of cross edges is larger than the spectral methods [10]. However, for large graphs, the total sequential time required is at least two to three orders of magnitude smaller as compared to spectral methods. Further, these methods can be easily parallelized.

3 CM-5 System Overview

This section gives a brief overview of the CM-5 system that we used to conduct our experiments. The CM-5 is available in configurations of 32 to 1024 processing nodes, each node being a SPARC microprocessor with 32M bytes of memory and optional vector units. The node operates at 33 MHz and is rated at 22 Mips and 5 MFlops. When equipped with vector units, each node of the machine is rated at 128 Mips (peak) and 128 MFlops (peak).

The CM-5 internal networks include two major components, a data network and a control network. The CM-5 has a separate diagnostics network to detect and isolate errors throughout the system. The data network provides high-performance data communications among all system components. The network has a peak bandwidth of 5M bytes/sec for node-to-

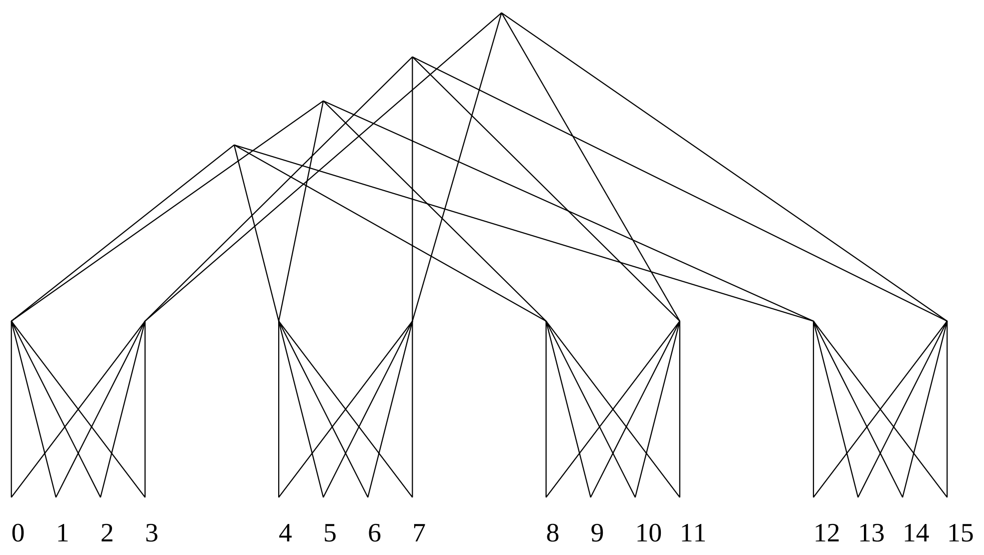


Figure 3: CM-5 data network with 16 nodes

node communication. However, if the destination is within the same cluster of 4 or 16, it can give a peak bandwidth of 20M bytes/sec and 10M bytes/sec, respectively [3]. Figure 3 shows the data network with 16 nodes. The control network handles operations requiring the cooperation of many or all processors. It accelerates collective operations such as broadcast and integer reduction, and system management operations such as error reporting.

4 Basic Operations

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

1. **Sending a Message**
   Sending a message from one node to another can be modeled as $O(\tau + \mu B)$, where τ is the overhead, μ is the transfer rate and B is the size of the message. As discussed in the previous section, the value of μ epends on whether the destination belongs to a specific subgroup and whether other nodes are sending messages. For our complexity analysis we will assume that τ and μ are constant, independent of the congestion and distance between two nodes.

2. **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 computes a concatenation of the local list in each of the processors. The resultant

```
For all processors $P_i$, $0 \leq i \leq n-1$, do in parallel
for $k$ = 1 to n-1 do
    $j = i \oplus k$
    if $COM(i,j) > 0$ then $p_i$ sends a message to $p_j$
    if $COM(j,i) > 0$ then $p_i$ receives a message from $p_j$
```

Figure 4: All-to-Many Personalized communication

vector $R[0 \cdots n-1]$ is stored in all the processors.

$$R[j] = V_{j\ div\ p}[j \bmod p]$$

This operation can be completed in $O(\phi_1 n)$ time on the CM-5, where ϕ_1 is a constant [3].

3. **Global Combine**

Assume that each processor contains a vector $V_i[0 \cdots n-1]$. Let p be the number of processors. The global concatenate operation computes an element-wise sum of the local list in each processor. The resultant vector $R[0 \cdots n-1]$ is stored in all the processors.

$$R[j] = \sum_{i=0}^{p-1} V_i[j]$$

This operation can be completed in $O(\phi_2 n)$ time on the CM-5, where ϕ_2 is a constant [3].

4. **All-to-Many Personalized Communication** [22]

In all-to-many personalized communication, each processor needs to send a different message (potentially of a different size) to a subset of all the processors. A simple strategy is to use multiple send-receive operations to perform all-to-many communication primitives. Each processor P_i sends a message to processor $P_{(i \oplus k)}$ and receives a message from $P_{(i \oplus k)}$, where $0 < k < p$. When $COM(i,j) = 0$, processor P_i will not send a message to processor P_j, but will receive a message from P_j if $COM(j,i) > 0$. The entire communication uses pairwise exchange ($j = i \oplus k \Leftrightarrow i = j \oplus k$). The cost of this collective communication depends on the structure of the communication matrix and the sizes of different messages.

Several other algorithms have been developed for scheduling such unstructured collective communication so as to reduce node contention (a node receiving messages from two nodes at the same time), as well as like-link contention (two messages with different source and destination passing through the same link) [15, 21]. However, many of these methods require the same communication structure to be utilized several times and are not appropriate in this case.

5. **Order Maintaining Load Balance**

Assume that each 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 the increasing order of the processor number, is also sorted. We would like to balance the load on each processor such that the global ordering of the elements does not change.

The load-balancing algorithm which maintains the sorted order is given in Figure 5. Steps 2 and 3 calculate the prefix sum and the average number of elements. We will assume $\overline{X}$ to be an integer for ease of presentation. 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 $\overline{X} = \frac{1}{n} \sum_{i=0}^{n-1} X_i$. We assume that $\overline{X}$ is an integer for ease of presentation.

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 easily determined 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 easily achieved.

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 any processor has is more than $\frac{\overline{X}}{K}$ and the maximum number of elements any processor has is less than $\overline{X}K$, it can be easily 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 node will send and receive a few messages. Since link contention is not a major problem on the CM-5, this operation can be completed in $O(\tau + \mu * \overline{X})$ time when the loads are nearly balanced.

```
For processor $P_i$, $0 \leq i < p$, in parallel do
    Step 1: $Z[0..p-1] = \textbf{Concatenate}(X_i)$
    Step 2: $Y[k] = \sum_{j=0}^{k-1} Z[j]$ for $k = 1, 2, ..., p-1$, $Y_0 = 0$
    Step 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 $(k-1)\overline{X}$ to $k\overline{X}$ */
    Step 4: Divide the local list into packets and send them to processors from left to right.
    Step 5: Receive messages and store them in the appropriate positions in the local array.
```

Figure 5: Order Maintaining Load Balance Algorithm.

6. **Order Maintaining Data Movement**

   Assume that each processor, i $(0 \leq i < p)$, contains a sorted array $V_i[0 \cdots X_i - 1]$. Further, a concatenation of all these arrays in the increasing order of the processor numbers is also sorted. In the Order Maintaining Data Movement operation, we would like to move the elements on each processor such that the global order of the elements does not change. However, unlike the previous procedure, load balancing is not required. The movement is decided by another sorted array $PART$ with p elements. All the local elements between $PART[i]$ and $PART[i+1]$ are to be moved to processor i $(0 \leq i < p)$. The algorithm is similar to the Order-Maintaining-Load-Balancing algorithm. Unlike the previous algorithm, the leftmost processor to which data has to be sent by a given processor is decided by a binary search on $PART$. The complexity of this algorithm depends on the maximum amount of data to be sent/received by any processor and the underlying communication network.

7. **Sort**

   Sorting a list of keys reorders them in a non-decreasing (or a non-increasing) order. There are several algorithms available in the literature for parallel sorting [12]. For distributed memory machines, a number of sorting schemes have been shown to be effective [2, 26]. We have used a parallel sampling-based sort for our problems. A detailed description of our method used is discussed in [18]. For the rest of this paper we will assume that sorting n elements on p processors requires $O(\frac{n \log n}{p})$ amount of time. This is true when n is $O(P^{1+\epsilon})$, $\epsilon > 0$.

5 Remapping for Applications Requiring Perturbations

In applications such as molecular dynamics, particle-in-a-cell methods, particle dynamics, etc., the interaction between several particles is simulated. These particles are dispersed in a two- or three-dimensional space, and the simulation is performed for a large number of time steps. At each time step, the numerical approximation techniques used for simulation dictate that the amount of particle movement be small. Further, most of the important interactions in the simulation are limited to points that are physically proximate. Assume that index-based mapping is used for partitioning these points. Corresponding index is expected to change by a small amount, but this is not always the case (e.g., the index of (3, 3) is 15 while the index for (3, 4) is 26).

Thus the remapping of the particles, after a few time steps, can be reduced to the following problem. There is a sorted list A of size n. A nearly sorted list B is derived from list A by perturbing each element by a small amount (on an average) by adding or subtracting a small random number.

We would like to develop an algorithm for sorting the new list B efficiently by utilizing information of the previously sorted list A. The basic principle used by the algorithm is the fact that A is close to B and sorted, and the partitions of A can be used to distribute the elements of B into lists of approximately equal size. Each of these lists is such that all the elements of the list are greater then the previous list. Sorting each of the sublists implies a globally sorted list. An incremental sorting algorithm, given in Figure 6, assumes the presence of a sorted list A, of size n, divided equally among all the processors in a contiguous fashion. The local list A is divided into l buckets to get approximate boundaries of l buckets of B on each processor. The maximum element of A is used to create an l's size *local_bound* array in each processor; $local\_bound_i[j] := A_i[\frac{jn}{pl}]$, $1 \leq j \leq l$. For each element of B an appropriate bucket must be obtained.

Each element of B can be classified into three categories, depending on whether the key belongs to the same bucket as the corresponding entry of A, or a different bucket on the same processor, or a bucket of another processor. These categories are termed Type 0, Type 1, and Type 2, respectively. For small perturbation, it is expected that most of the keys will be of Type 0, a small fraction of the keys will be of Type 1, and an even smaller fraction of the keys will be of Type 2. For this reason, using a search algorithm that is biased towards the current position of the key is worth considering. The algorithm described in Figure 6 reflects this bias. Checks are made for keys of Type 0 before searching for Type 1 followed by Type 2. We opted to choose interpolation search over binary search for the same reasons. Type 0 and Type 1 keys are added to local buffers representing the different buckets on the same

```
/* $Bound[i]$ is the largest key of $A_i$. */
/* $local\_bound[j]$ is the largest key of $A_i$ in $[\frac{(j-1)n}{l}, \frac{jn}{l})$, where $l$ is the number of buckets in a processor */
For each processor $i$ do in parallel
Step 1 :   $Bound$ := Global_concatenate( $local\_bound_i[l]$)
Step 2 :   For $j \longleftarrow 1$ to $m_i$ do
               Case Key of $B[j]$ of
               Case 0 : $B[j]$ in $[local\_bound[\frac{j}{l}], local\_bound[\frac{j}{l}+1]]$
                             Add $B[j]$ to $tmp\_local_i[\frac{j}{l}]$
               Case 1 : out of $[local\_bound_{[}\frac{j}{l}], local\_bound_{[}\frac{j}{l}+1]]$ but still within
                     $[Bound[i-1], Bound[i]]$
                             $dest$ := Interpolation_search($B[j]$, $local\_bound$)
                             Add $B[j]$ to $tmp\_merge_i[dest]$
               Case 2 : out of processor
                             $proc$ := Interpolation_search($B[j]$, $Bound$)
                             Add $B[j]$ to $send\_list[proc]$
Step 3 :   Apply All-to-Many communication using $send\_list$ and receive elements in $received_i$
Step 4 :   For those elements received in Step 3
               $dest$ := Interpolation_search($received_i[j]$, $local\_bound$)
               Add $received_i[j]$ to $tmp\_merge_i[dest]$
Step 5 :   Sort each bucket $B_i$. ($B_i$ has all the elements in $tmp\_local_i$ and $tmp\_merge_i$.)
Step 6 :   Perform a Order Maintaining Load Balance on list $B$
```

Figure 6: Bucket Based Incremental Sorting Algorithm

processor. Type 2 keys have to be moved to a different processor and added to appropriate buffers.

Step 3 completes the data transmission for the out-of-processor keys. Step 4 searches for the buckets for nonlocal keys. Step 5 performs the sorting in each of the buckets. This is followed by a load-balancing step.

The exact complexity of these algorithms is hard to derive as they are data dependent. The following provides an approximate analysis. Assuming that n keys are distributed equally in each of the processors ($\frac{n}{p}$ in each processor), the time required for Step 1 on the CM-5 is $O(\phi p)$. With l buckets (intervals) in each processor ($l \leq \frac{n}{p}$), Step 2 requires $O(\beta \frac{n}{p} + \gamma \frac{n}{p} \log_a l + \eta \frac{n}{p} \log_a p)$, where

$$\beta = \frac{no.\ of\ keys\ of\ type\ 0}{\frac{n}{p}},$$

$$\gamma = \frac{no.\ of\ keys\ of\ type\ 1}{\frac{n}{p}},$$

$$\eta = \frac{no.\ of\ keys\ of\ type\ 2}{\frac{n}{p}},$$

$$\beta + \gamma + \eta = 1.$$

This analysis assume that a is the corresponding base, in the sense of binary search, for interpolation search. For the case when the keys have uniform distribution, the time required for interpolation searche is $O(\log\log n)$ on an average. In many practical case this assumption is true. The cost of Steps 3 and 4 in the worst case is $O(p\tau + \eta\frac{n}{p}\mu)$ and $O(\frac{\rho}{p}\log_a l))$, respectively, where $\rho = \frac{no.\ of\ received\ keys\ of\ type\ 2}{\frac{n}{p}}$. Step 5 requires $O(\frac{n}{p})$ to copy keys back to list B. Assuming that the amount of perturbation is not large, the values for γ, η, and ρ are small, as compared to β. So the complexity of Steps 2 and 4 can be approximated by $O(\frac{n}{p})$. The complexity of Step 5 will depend on l and the size of each local bucket. The time required is $\sum_{i=0}^{l-1} O(m_i \log m_i)$, where m_i is the number of keys in local bucket i. Assuming all buckets are of approximately the same size, this can be approximated by $O(\frac{n}{p}\log\frac{n}{pl})$.

From practical perspectives, a simple optimization step can be added to reduce the cost of sorting in Step 5. Since the list B is almost sorted, all the elements of B in type 0 will be in a nearly ascending order. A sorted sublist can be obtained by removing the elements that are less than the largest found so far. This preprocessing step requires $O(b)$ amount of time where b is the size of the list. Assuming that α is the fraction of such keys, the sorting time can be reduced to $O((1-\alpha)b\log(1-\alpha) + b)$ in Step 5.

There is a clear tradeoff between the time spent in Step 5 and the searching cost in Steps 2 and 4. The complexity of above algorithm depends on the size of each bucket, (l). If l is small, the cost of Step 2 and Step 4 are low. However, the cost of Step 5 is large. For large l, the values of β, γ, and η can no longer be assumed to be small. Thus an appropriate value of l must be determined experimentally. Even assuming that the cost of search and data movement is negligible in the remapping algorithm, the maximum speedup that can be achieved over sorting from scratch is limited to

$$\frac{O(\frac{n}{p}\log n)}{O(\frac{n}{p}\log\frac{n}{pl})} \approx C\frac{\log\frac{n}{p}}{\log\frac{n}{pl}},$$

where C is expected to be greater than 1.

5.1 Experimental Results

The above algorithms were executed on a 32-node CM-5. To study the behavior of these algorithms for varying values of n, artificial elements were generated in a three-dimensional space using a uniform random number generator. Each of the coordinate values were between 0 and 20, and the number of bits attached to each dimension was 10. This corresponds to 1024 bins along each dimension with the size of the index key being 30 bits. Thus each bin represented a value of approximately $.02 \times .02 \times .02$ units. The perturbation was limited to a sphere of radius r and was accomplished as follows. For each data point we generated 3 random numbers λ, θ, and ϕ where λr is the length of radius and θ and ϕ are within [0, 2 π). Using these three random numbers, the values of the three coordinates were calculated in the following fashion.

$$\begin{aligned} &x_{i,j} = x_{i,j} + \triangle x_{i,j} \text{ where} \\ &\triangle x_{i,0} = \lambda_i r \sin(\theta_i), \\ &\triangle x_{i,1} = \lambda_i r \cos(\theta_i) \sin(\phi_i), \text{ and} \\ &\triangle x_{i,2} = \lambda_i r \cos(\theta_i) \cos(\phi_i). \end{aligned}$$

Figure 7 shows the experimental results for the perturbation sorting on a 32-node CM-5 for different radii of perturbations. A comparison is made with sorting from scratch. These results show that for 64K data elements, the speedup achieved over sorting from scratch is about a factor of 4 when the perturbation radius is 0.01 and the optimization mentioned in the earlier section is performed. This improvement reduces as the amount of perturbation increases. Further, the overhead of optimization is more than the gains when perturbation is large.

The experimental results in Figure 8 include the indexing time for graph remapping. Indexing represents a major fraction of time spent on mapping. Since all the coordinates are assumed to have been perturbed, a new value of index has to be recalculated. Thus, the relative performance gain of mapping versus remapping is smaller (as compared to sorting vs. perturbation sorting) although the absolute performance gains are the same. In practical cases, it is necessary only to recalculate the indices for vertices that have moved sufficiently (such that their indices might have changed) to reduce the cost. The cost of the indexing can also be substantially reduced by using table look-up methods.

Figure 9 shows the tradeoffs for different bucket sizes. If the bucket size is small, the cost of searching is higher. When the bucket sizes are large, the cost of sorting goes up. The optimal bucket size for this particular case was equal to 10.

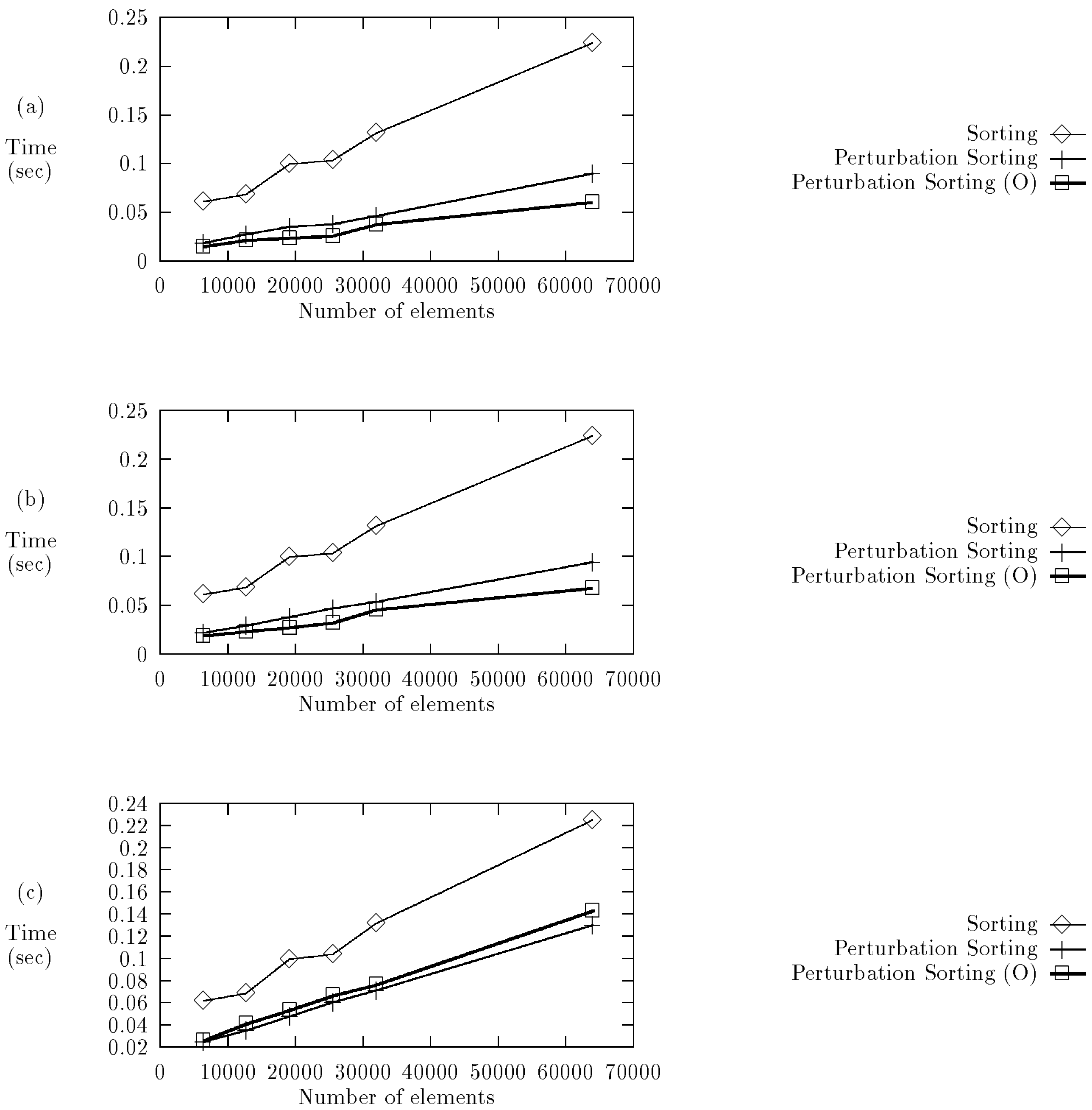


Figure 7: The performance of different sorting algorithms for a nearly sorted list compared with sorting an unsorted list. (a) radius of perturbation = 0.01, (b) radius of perturbation = 0.1, and (c) radius of perturbation = 1.

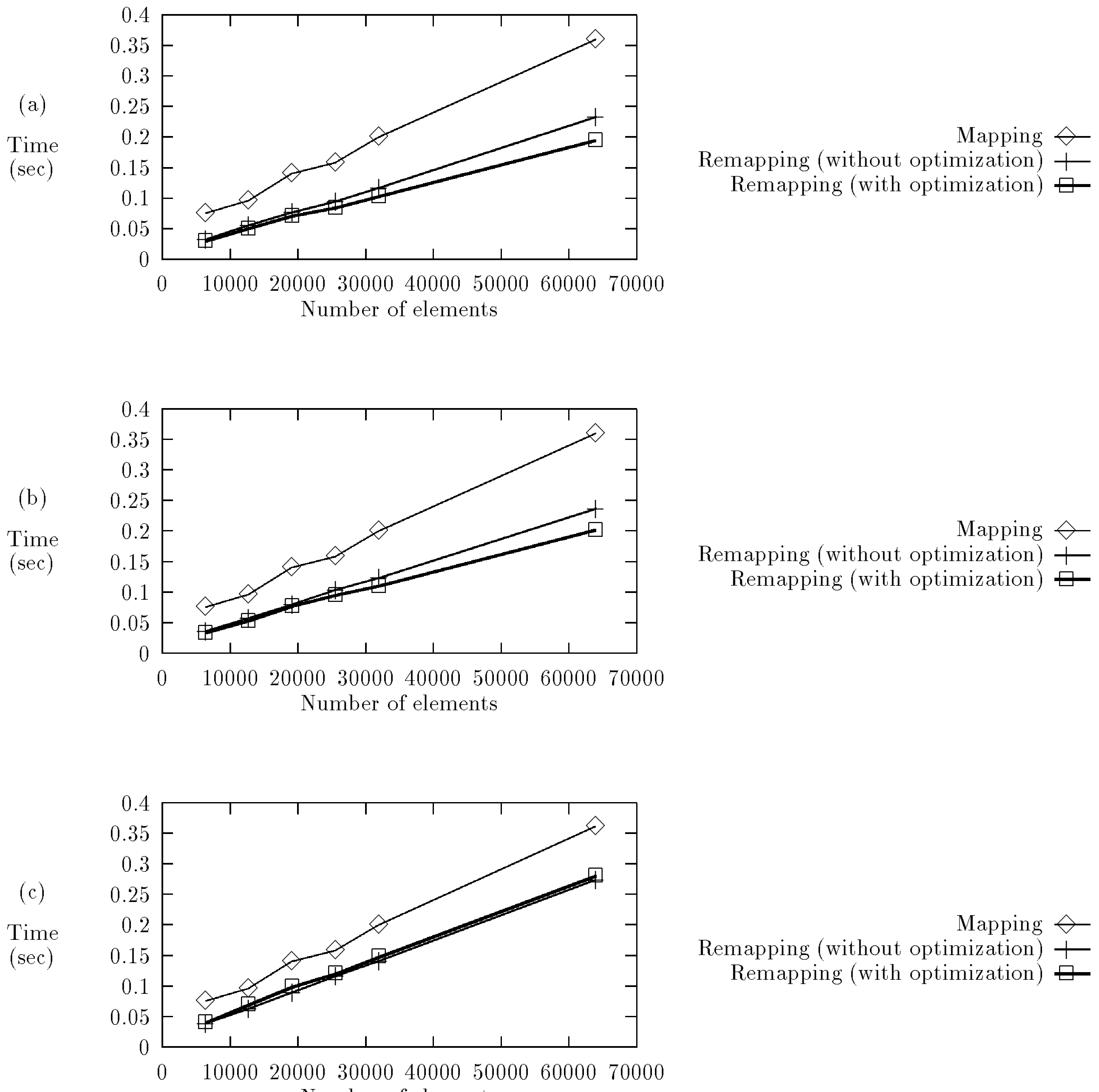


Figure 8: The performance of different remapping algorithms compared with mapping using index-based scheme (including indexing time). (a) radius of perturbation = 0.01, (b) radius of perturbation = 0.1, and (c) radius of perturbation = 1.

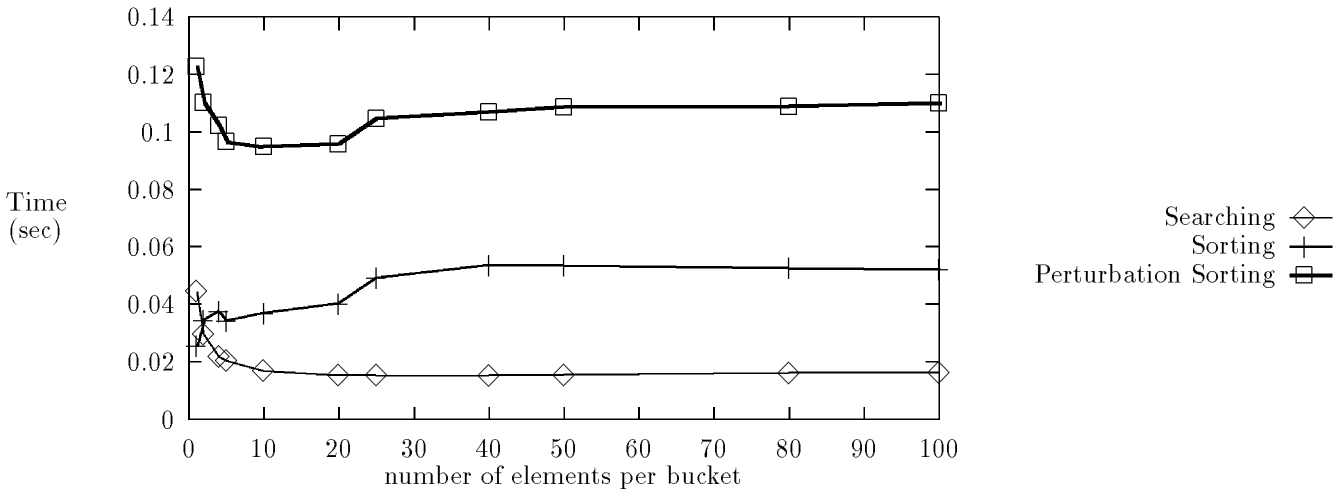


Figure 9: Time for searching phase and sorting phase of the perturbation sorting algorithm for different bucket sizes. The size of data set is 64,000 and $| r |= 0.1$.

6 Remapping for Applications Requiring Addition/Deletion of Nodes

For many applications, such as adaptive meshes, new nodes are added to the computational graph. Typically, this is 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 nodes may be added in a particular portion, only to be removed after a few phases. An example of an initial mesh and a refined mesh is given in Figure 17. The following discussion is limited to the case when nodes are added to the computational graph. All of these algorithms can be easily modified when this is not the case.

Remapping requires calculating the shuffled row-major indices of the new nodes, 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 nodes that are added) to a sorted list (corresponding to the indices of the old nodes). The case of node deletion is similar.

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 $O(n)$. In the following subsection we describe three different parallel algorithms for solving this problem under the assumption $m \ll n$.

```
/* 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 $A_i$ the elements received in Step 4 and call it $C_i$
Step 5 :    Merge list $A_i$ and $C_i$
Step 6 :    Perform Order Maintaining Load Balance on $A$
```

Figure 10: Simple Merging Algorithm

We assume that the list A is already sorted and divided equally among all the processors. This corresponds to the partitioning of the previous phase. The new points added/deleted in the new phase are assumed to be equally divided among all the processors for the following algorithms. However, this is not going to be the case in general. In fact, for most practical cases the incremental nodes are added in localized portions. This 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 [20]. The cost of this load-balancing scheme is nominal compared to the cost of merging for most cases.

Most of the analysis provided in the following section corresponds to near worst-case scenarios for each of the algorithms. An average case is hard to define and depends on the application to be solved and the particular algorithm used for merging. The performance of these algorithms would be much better on typical cases than the results described in the following sections. Our emphasis is to show that the worst-case cost of remapping is a small fraction of the total cost of remapping from scratch.

6.1 Simple Merging Algorithm

A simple merging algorithm is given in Figure 10. 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

```
For each processor $i$ do in parallel
/* Stage 1 : Preprocessing phase */
Step 1 :   Create $q$ buckets for list $A_i$ and record the boundary values in $bucket_i[1..q]$
Step 2 :   $BUCKET[1..pq]$ := Global_Concatenate($bucket_i[1..q]$)
Step 3 :   For $k \longleftarrow 1$ to $m_i$ do
               $p$ :=Binary_search($B_i[k], BUCKET$)
               $count_i[p] := count_i[p]+1$
Step 4 :   $SUM[1..pq]$ := Global_Combine($count[1..pq]$)
Step 5 :   For $k \longleftarrow 0$ to $p-1$ do
               $Coarse\_Bound[k] := \min_{j=1}^{pq}$ abs($SUM[j] - \frac{i(n+m)}{p}$)
Step 6 :   Perform Order Maintaining Data Movement using $Coarse\_Bound$
/* Stage 2 : Merging phase */
Step 7 :   Apply Simple Merge /* See Figure 10 */
```

Figure 11: Bucket Merging Algorithm

generated and the size of the packets. All-to-many communication algorithms have been described in Section 4. Assuming link contention does not affect total communication time, 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 Step 1 through Step 5 is $O(\frac{m}{p} \log p + p\tau + \mu m + m \log m + \frac{n}{p})$. When n and m are compared to p, and when $m \log m < \frac{n}{p}$ (the incremental data is much less than the data on one processor), this corresponds to $O(\frac{n}{p})$[1]. For this case the **Order Maintaining Load Balance** will be reduced to shifting to either the left or the right neighbors. The total time required for this step under the above assumptions is $O(\tau + \mu\gamma)$ where γ is the maximum amount of data moving out of any processor.

6.2 Bucket-Merging Algorithm

The worst-case scenario for the previous algorithm is when all the new points to be added fall into the boundaries of one processor. When all or most of the elements to be added lie between the minimum and the maximum element of the sublist of one processor, it will have the effect of that processor receiving all the messages as well as sorting the elements received.

[1]The time required for mapping from scratch would be $O(\frac{n}{p} \log n)$, where n is the number of vertices.

This algorithm attempts to improve the worst-case scenario of the previous algorithm (Figure 10). A high-level description of the bucket-merging algorithm is given in Figure 11. This algorithm balances the load for the merging step more effectively by a preprocessing step that adjusts the global array A before the merging phase. Each processor divides the local lists into buckets. Essentially, each bucket represents a virtual processor. Step 4 determines the number of elements that will be received by each bucket. Step 6 calculates the approximate number of buckets that should belong to the different processors. Assuming $m < \frac{n}{p}$, the complexity of Phase 1 would be $O(pq + \frac{m}{p} \log pq)$. For phase 2, the complexity analysis would be different then the one discussed in the previous section. The worst-case cost of this phase again is $O(m \log m + (p\tau + \mu m) + \frac{n}{p})$. This corresponds to all the data going to one bucket. However, if all the indices are equally distributed among all the buckets belonging to a processor at the beginning of phase 1, some of these buckets will move toward the left or the right.

An appropriate choice of q is required to minimize the sum of the cost of the preprocessing phase and the merging phase. This will reduce the load on the processor that has received all the elements in the simple merge algorithm. Larger values of q increase the cost of preprocessing but lead to potentially better performance in the second stage.

6.3 Sort-Based Merging Algorithm

The algorithm first sorts all the keys in the unordered list, thus reducing the problem to one of merging two sorted lists. Although merging of sorted lists has been a widely studied problem in the literature, most of the algorithms thus developed have been for cases when the two lists are equal in size. To the best of our knowledge, we have not seen any algorithms for coarse-grained machines for such cases.

A high-level overview of the algorithm for merging two lists is given in Figure 12. The first step divides A and B into buckets of equal size (say δ). The number of buckets of A and B on each processor is q_a and q_b, respectively. For the sake of presentation, we will assume that the number of local elements of A and B (n/p and m/p, respectively) are divisible by δ; we will describe the required changes if this is not the case. A simple example is used to explain the details of the algorithm.

The basic principle of our method is to find partitions that will divide the merged list into approximately equal sections. The first step is to find the boundaries of the buckets. This can be achieved by storing the maximum element of every bucket. We also keep track of the size of each bucket, which is useful for making the required modifications when the number of elements are not multiples of the bucket size. Two new lists are formed on each of the processors by concatenating those boundary elements corresponding to the buckets of list A and B, respectively. The sizes of these lists are $q_a\ p$ and $q_b\ p$ for A and B, respectively.

A similar operation is performed for the size list (which maintains the size of each bucket). The time required for this operation on the CM-5 is $O(q_a\ p + q_b\ p)$. These two lists are merged to give another list, C, of size $O((q_a + q_b)p)$. The time required for this merging is $O(p(q_a + q_b))$.

Since the sizes of all the buckets are equal, and the total number of buckets is $(q_a + q_b)p$, the number of buckets belonging to each processor should be approximately $q_a + q_b$. This can be achieved by counting the number of buckets from left to right. Processor i gets all the data corresponding to $C[i \cdot (q_a + q_b)]$ to $C[(i+1)((q_a + q_b) - 1]$. It can easily be shown that the maximum difference of the number of elements between any two partitions is less than the size of two buckets.

The next step is to apply an **Order Maintaining Data Movement** operation on A using the partitions obtained in the processors. This is followed by a **Order Maintaining Data Movement** operation on B. The cost of these operations is difficult to analyze and depend on the amount of data movement. However, the fact that the partitioning provided by C is nearly accurate implies that the data of A and B, which has to be moved to processor i, will be required by any merging algorithm that assumes these lists are distributed. In that sense, the amount of data movement is close to optimal.

Once the data of A and B is moved to the correct processor in Step 4, it is merged locally in Step 6. The cost of the operation is the sum of the sizes of the local lists. Because the result array would be nearly equally divided (within 2 buckets), the time required for this operation is $O((\frac{m+n}{p}) + 2\delta)$. The last step would typically result in a shift to the left or right processor. Thus, the total amount of time required for merging is $O((q_a + q_b) \cdot p + \frac{m+n}{p} + 2\delta)$ and the cost of Data Movement in Step 5 and Step 7. When m and $n \gg p$ and δ is reasonable, the algorithm is close to optimal.

For the case when all the buckets are not of the same size, the arrays Length_A and Length_B can be used to find the partitions in Step 4. Assuming a large number of buckets, this is practically always possible. A practical optimization can be added to have larger bucket sizes for A than B, as B is much smaller than A.

Let PRED$[i] = \sum_{j=0}^{i}$ (length of the bucket ending at $C[i]$.) This requires keeping extra information about every element of C (which list it belongs to and what bucket it represents). PRED$[i]$ represents the approximate number of elements (within the maximum size of any bucket) that is less than $C[i]$. A simple algorithm can be used to divide PART into approximately equal parts.

```
/* $A$ is sorted list of size $n$ distributed equally among all the processors */
/* $B$ is sorted list of size $m$ distributed equally among all the processors */
  **For** each processor $i$ **do in parallel**
  Step 1: Divide local list of $A$ and $B$ into buckets of size $\delta$. Let the number of buckets be $q_A$ and $q_B$, respectively.
          Let the maximum of each bucket and the size of the buckets of $A$ be given by $\max A_i[1 \ldots q_a]$ and size $A_i[1 \ldots q_a]$,
          respectively. The corresponding arrays for $B$ are $\max B_i[1 \ldots q_b]$ size $B_i[1 \ldots q_b]$.
  Step 2: $BOUND\_A[1 \ldots pq_a] \longleftarrow$ CONCATENATE $(\max A_i)$
          $BOUND\_B[1 \ldots pq_b] \longleftarrow$ CONCATENATE $(\max A_i)$
          $LENGTH\_A[1 \ldots pq_a] \longleftarrow$ CONCATENATE (size $A_i$)
          $LENGTH\_B[1 \ldots pq_a] \longleftarrow$ CONCATENATE (size $B_i$)
  Step 3: $C \longleftarrow$ MERGE $(BOUND\_A, BOUND\_B)$
  Step 4: Divide $C$ into $p$ approximately equal partitions. Let it be $PART[0 \ldots p-1]$
  Step 5: Order Maintaining Data_Movement $(A, PART)$
          Order Maintaining Data_Movement $(B, PART)$
  Step 6: $D_i \longleftarrow$ Merge $(A_i, B_i)$
  Step 7: **Order Maintaining Load Balancing** $(D)$
```

Figure 12: Parallel algorithm for merging two sorted lists

6.4 Experimental Results

We generated two types of data sets in order to study the behavior of different algorithms on the CM-5.

1. *Data Set 1:* Each processor generated a random number (uniform distribution) of elements such that the index values were within the boundaries of that processor. This is expected to be to the near best case for all the merging algorithms.

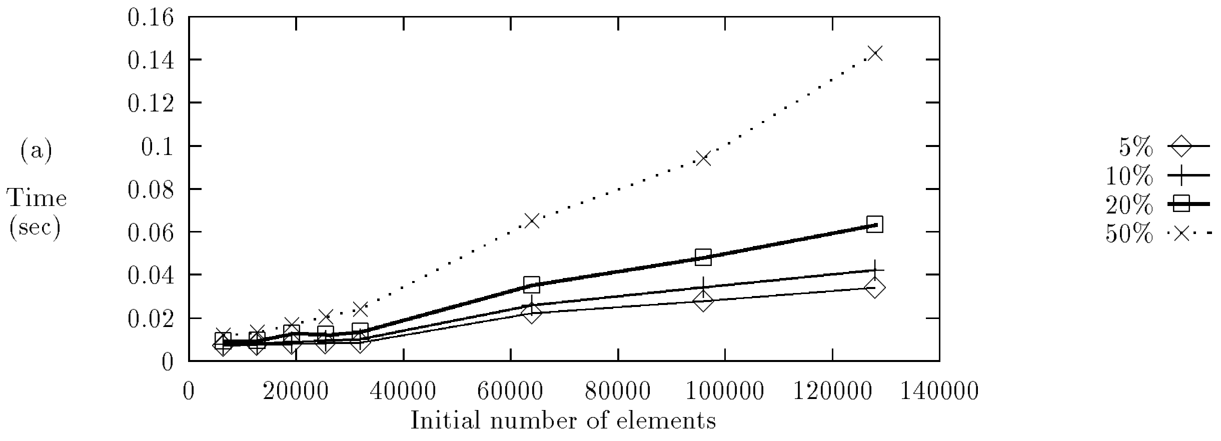
2. *Data Set 2:* One processor generated all the elements such that all the elements were within the smallest and largest elements of that processor. This was followed by a load-balancing step in which the elements were distributed to all processors equally. This represents the near worst-case scenario for the merging algorithms.

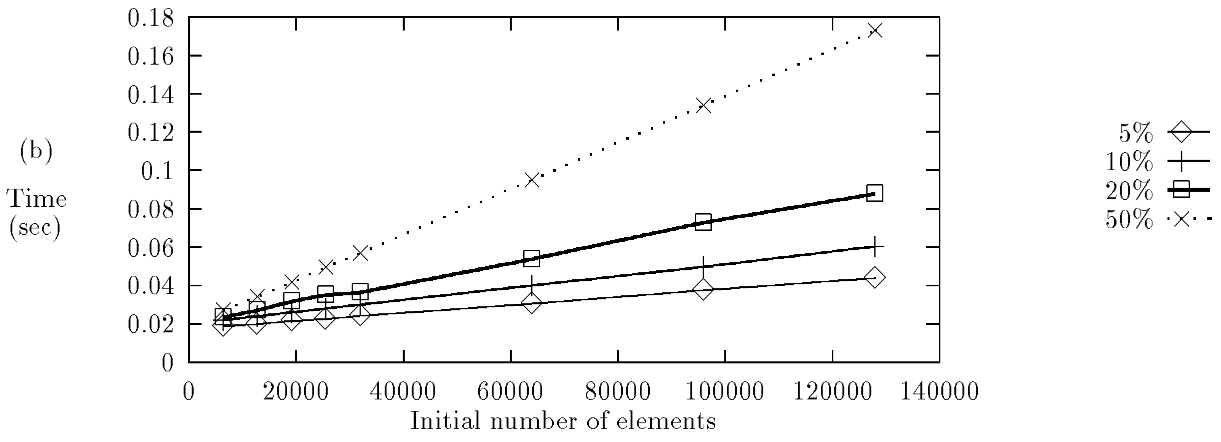
For Data Set 1 (Figure 13), the simple merge algorithm performs the best. This is expected as the amount of data movement is minimal and the elements to be merged are nearly equally distributed. For initial array size of 64,000 and the number of additional elements equal to 20%, the cost of simple merge, bucket merge and sort-based merge are 0.035, 0.04 and 0.07 seconds respectively. This compares favorably with the corresponding cost of sorting which is 0.195 seconds.

For Data Set 2 (Figure 14), the situation is totally different. The time required for the simple merge is much more than the time required for the other two algorithms when the number of additional elements are larger than a small fraction of the total number of nodes. Further, the time required for simple merge is much worse as compared to Data Set 1. The time required for bucket merge is considerably lower than for simple merge. The extra cost of adding a preprocessing phase is offset in the next phase when the number of additional nodes are larger than a small fraction. The time required for the sort-based algorithm does not deteriorate much, compared to Data Set 1. It is the algorithm of choice if the number of additional nodes is large.

For Data Set 2, comparison between different algorithms is given in Figure 15. The merging algorithms perform better than sorting from scratch. By choosing the algorithm with the best worst case performance, depending on the different quantity of additional nodes added, a combined merge algorithm can be derived. The comparison of the combined merge algorithm is given in Figure 15 (c), which shows that performance gains of factors of 2 to 4 can be achieved when the number of additional elements added is less than 10%. We believe that for most typical cases the performance of the combined merge algorithms will be an order of magnitude better, a significant improvement since the mapping algorithm based on sorting is itself very fast as well as parallel.

Figure 16 includes the cost of indexing in the combined merge algorithm to give the worst-case cost of remapping. It assumes that, for mapping, the index calculation is done





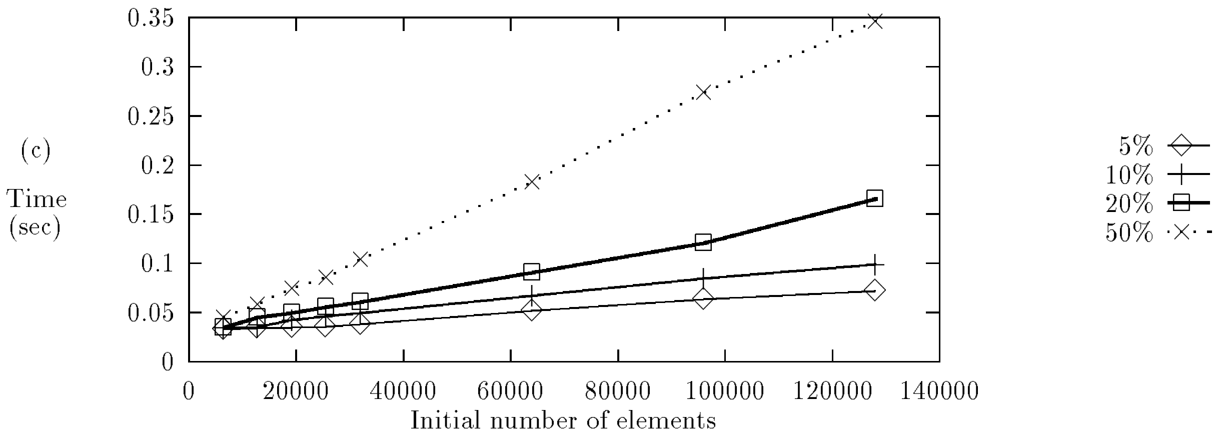
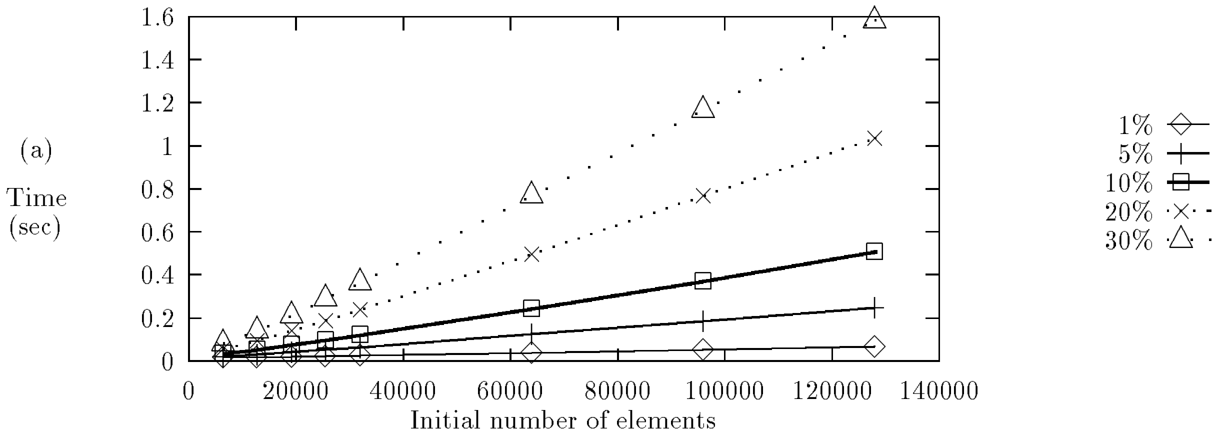
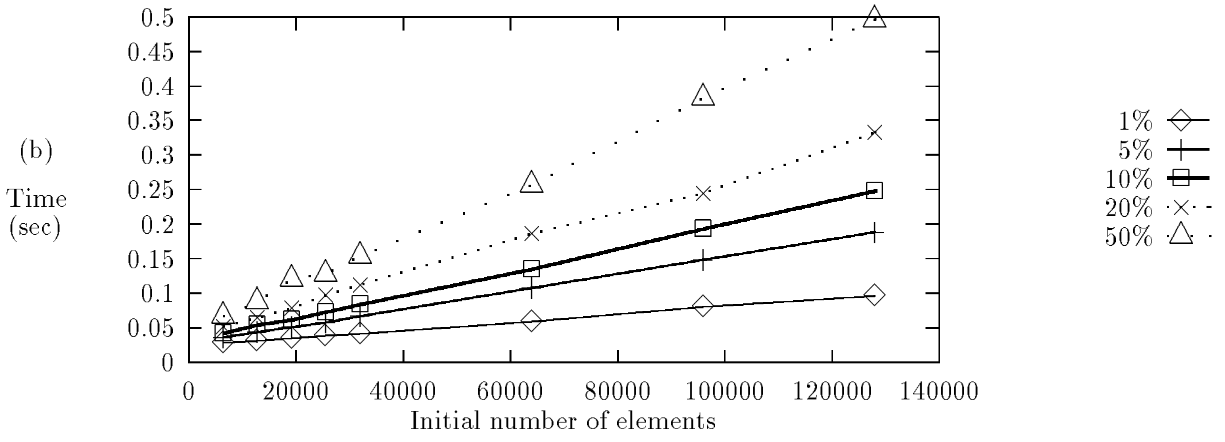


Figure 13: Performance comparison of different merging algorithms on 32-node CM-5 for different fraction of additional elements for Data Set 1. (a) simple merging algorithm, (b) bucket merging algorithm, and (c) sort-based merging algorithm





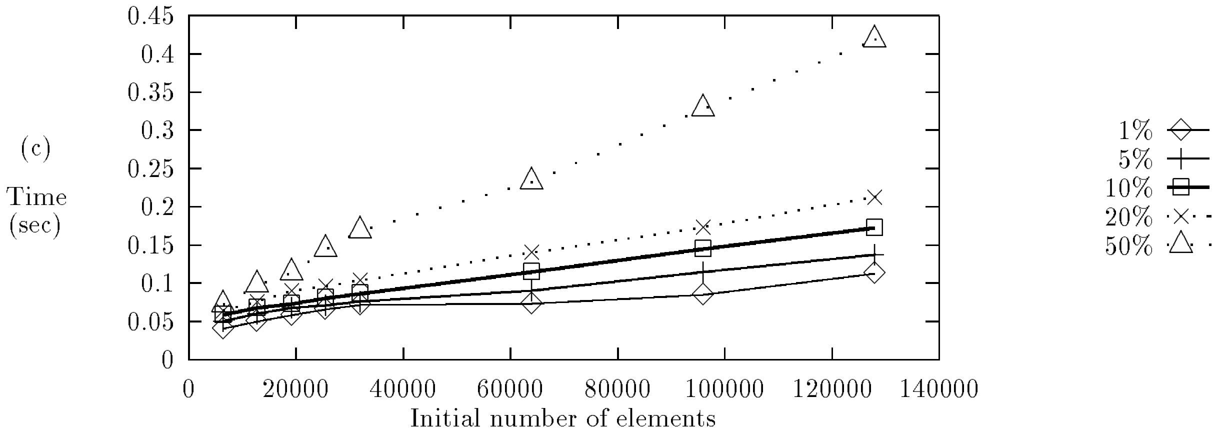


Figure 14: Performance comparison of different merging algorithms on 32-node CM-5 for different fraction of additional elements for Data Set 2. (a) simple merging algorithm, (b) bucket merging algorithm, and (c) sort-based merging algorithm

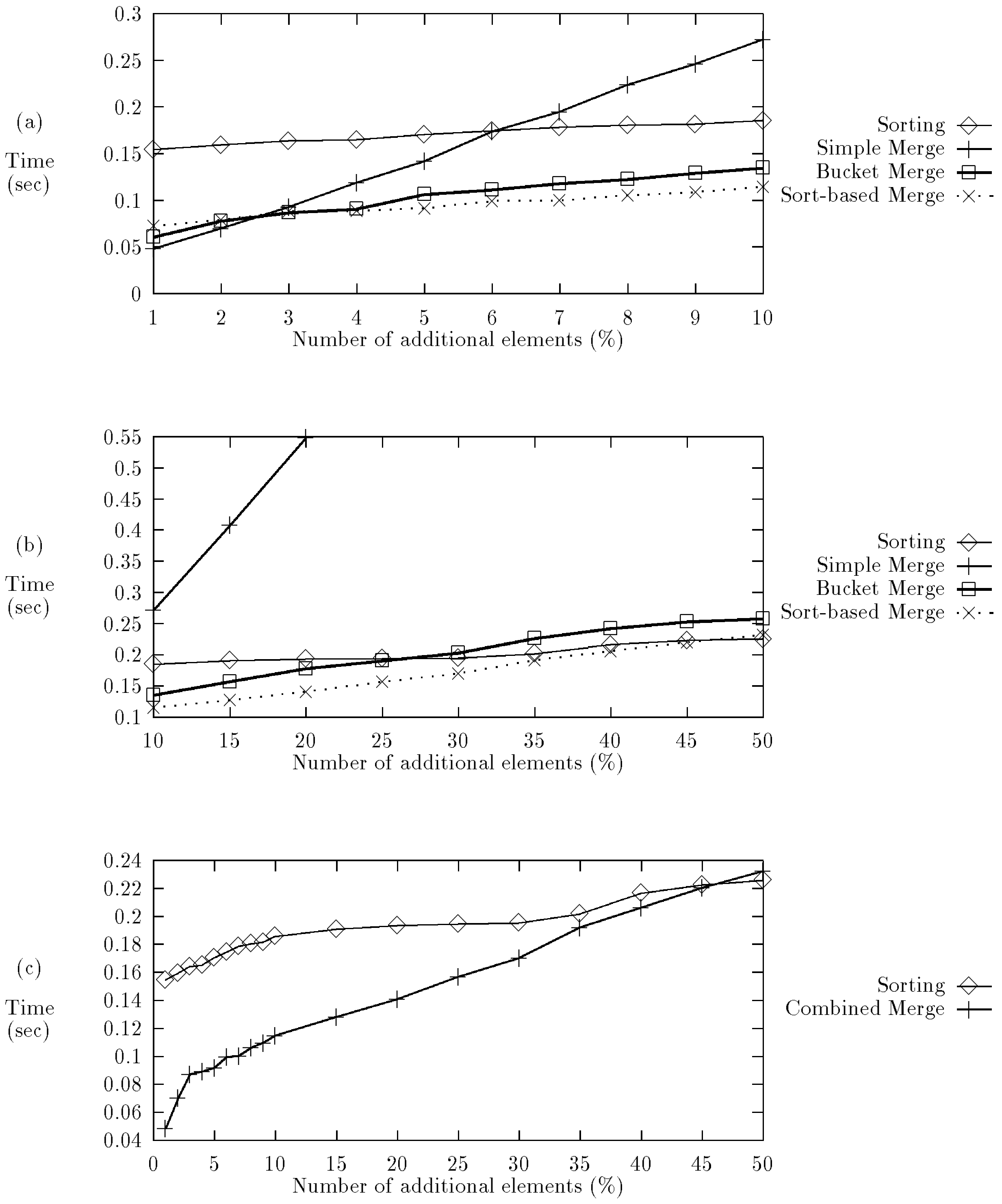


Figure 15: Performance comparison of different algorithms for Data Set 2 (initial number of elements = $64,000$) (a) The fraction of additional elements between 1% and 10%. (b) The fraction of additional elements between 10% and 50%. (c) Comparison of the best merging algorithm and sorting from scratch.

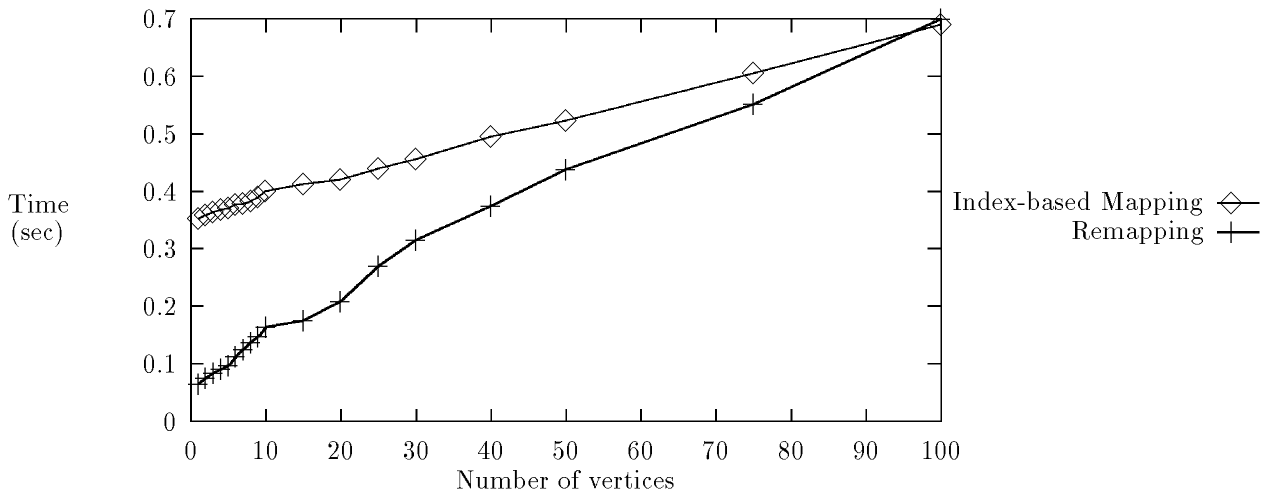


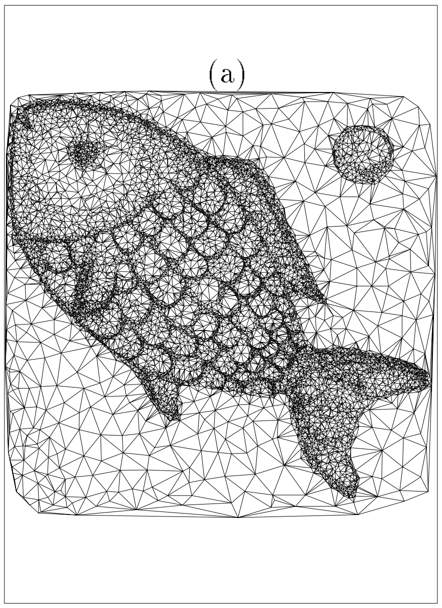
Figure 16: Comparison for Data set 2 index-based mapping and incremental mapping algorithms on $| V |= 64,000$

for the vertices, while for remapping the index calculation is done for additional vertices only. The results show that for small fractions of additional vertices, the cost of remapping is an order of magnitude better.

Figure 17 (a) shows a highly irregular graph with 10,166 nodes and 30,417 edges. Additional nodes and edges were added in small localized areas. The new graph is given in Figure 17 (b). It has 10,838 vertices and 32,487 edges.

Figure 18 gives the cost of performing the mapping the initial graph and the cost of remapping with the additional nodes/edges suing the index based partitioner on a 32 processor CM-5. All the timings except the index based repartitioning for the new graph assume that mapping is performed from scratch. The timings for recursive coordinate bisection and recursive spectral bisection are sequential times on a SUN4. Even assuming perfect parallelization of these two methods, the time required for indexed based partitioner is better than coordinate bisection and much better than spectral bisection. Further, the number of cross edges generated is close to the cross edges generated by coordinate bisetion and slightly worse than the cross edges generated by spectral bisection.

The cost of remapping is less than 10% of the time required for mapping from scratch. It should be noted that most of the time (Figure 17) for such a small number of vertices is spent in communication. This is because our software currently uses a send/receive style of message passing which have large overheads. Since the size of the messages is small these timings can be improved considerably by using active message based communication primitives that have considerably lower set up costs on the CM-5.



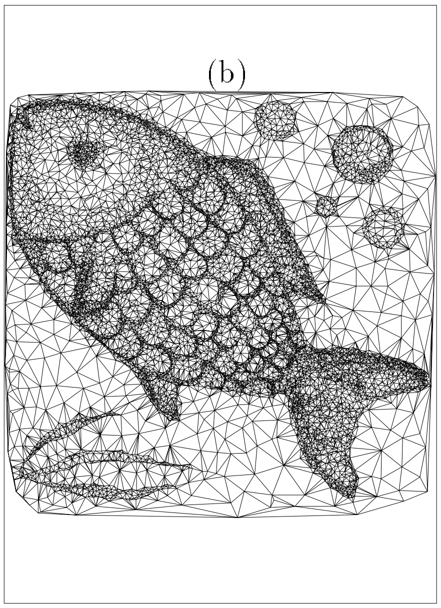


Figure 17: (a) Initial Graph (b) New Graph

Graph	Initial Graph $\lvert V\rvert = 10,166$ $\lvert E\rvert = 30,471$		New Graph $\lvert V\rvert = 10,838$ $\lvert E\rvert = 32487$	
Partitioner	Time	Cross Edges	Time	Cross Edges
Recursive Coordinate Bisection	3.133 s	2585	3.339 s	2550
Recursive Spectral Bisection	800.05 s	2118	904.81 s	2158
Index-Based Mapping (S)	0.087 p	2687	0.096 p	2973
Index-Based Mapping (H)	0.089 p	2825	0.098 p	2907
Index-Based Remapping (S)	—	—	0.008 p	2973
Index-Based Remapping (H)	—	—	0.008 p	2907

Time unit in seconds.
(S) - using Shuffled Row-Major indexing scheme.
(H) - using Hilbert Space Filling Curve.
p - parallel timing on a 32-node CM-5.
s - sequential timing on SUN4.

Figure 18: Cost of performing mapping and remapping using different partitioners

7 Conclusions

In this paper we have presented parallel remapping algorithms for a class of incremental and adaptive data parallel applications. The index-based mapping scheme has been shown to be extremely fast and to produce good quality mappings [18]. We have shown that by using the methods developed in this paper, remapping can be achieved at a fraction of the time required for mapping. Experimental results for these algorithms on a 32 node CM-5 support our conclusions.

We believe our methods would be crucial in the parallelization of the class of incremental and adaptive data parallel applications targeted in this paper. One drawback of these methods is that they do not take the edge information into account for partitioning. We are currently developing remapping algorithms that exploit this information in order to reduce the number of cross edges [17]. However, the computational cost of these methods are considerably higher.

References

[1] Rahul Bhargava, Virinder Singh, and Sanjay Ranka. A Modified Mean Field Annealing Algorithm for Task Graph Partitioning. Technical report, Syracuse University. Under preparation.

[2] Masood Bolorforoush, Nastaran S. Coleman, Donna Quammen, and Pearl Wang. A Parallel Randomized Sorting Algorithm. In *Proceedings of the International Conference on Parallel Processing*, August 1992.

[3] Zeki Bozkus, Sanjay Ranka, and Geoffrey C. Fox. Benchmarking the CM-5 Multicomputer. In *Proceedings of the Frontiers of Massively Parallel Computation*, 1992.

[4] B.R. Brooks, R.E. Bruccoleri, B.D. 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.

[5] 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.

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

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

[8] Leonardo Dagum. Data Parallel Sorting for Particle Simulation. *Concurrency*, 4(3):241–255, May 1992.

[9] M. R. Garey and D. S. Johnson. Computers and Intractability.

[10] Bruce Hendrickson and Robert Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. Technical Report SAND92-1460, Sandia National Laboratories, Albuquerque, NM 87185, 1992.

[11] Bruce Hendrickson and Robert Leland. The Chaco User's Guide, Version 1.0. Technical Report SAND93-2339, Sandia National Laboratories, October 1993.

[12] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin-Cummings, 1994.

[13] P.C. Liewer and V.K. Decyk. A General Concurrent Algorithm for Plasma Particle-in-Cell Simulation Codes. *Journal of Computational Physics*, 2:302–322, 1985.

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

[15] Lionel M. Ni and Philip K. McKinley. A Survey of Wormhole Routing Techniques in Direct Networks. *IEEE Computer*, 26(2):62–76, February 1993.

[16] S. Nolting. Nonlinear Adaptive Finite Element Systems on Distributed Memory Computers. In *Proceedings of European Distributed Memory Computing Conference*, April 1991.

[17] Chao-Wei Ou and Sanjay Ranka. Parallel Incremental Graph Partitioning Using Linear Programming. Technical report, Northeast Parallel Architectures Center at Syracuse University, April 1994.

[18] Chao-Wei Ou, Sanjay Ranka, and Geoffrey Fox. Fast Mapping And Remapping Algorithm For Irregular and Adaptive Problems. In *Proceedings of the 1993 International Conference on Parallel and Distributed Systems*, Taipei, Taiwan, December 1993.

[19] A. Pothen, H. Simon, and K-P Liou. Partitioning Sparse Matrices with Eigenvectors of Graphs. *SIAM Journal of Matrix Analysis and Application*, 11(3), July 1990.

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

[21] Sanjay Ranka and Jhy-Chun Wang. Static and Runtime Scheduling of Unstructured Communication. In *Proceedings of the 2nd Symposium on Parallel Computational Methods for Large Scale Structure Analysis and Design*, Norfolk, VA, February 1993.

[22] Sanjay Ranka, Jhy-Chun Wang, and Manoj Kumar. All-to-Many Personalized Communication on Distributed Memory Machines. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[23] J. K. Salmon. Parallel Hierarchical N-Body Method. Technical Report CRPC-90-14, Center for Research in Parallel Computing, Caltech, Pasadena, CA, 1990.

[24] John Salmon and David Warren. personal communication.

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

[26] Hanmao Shi and Jonathan Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.

[27] D.W. Walker. Characterizing the Parallel Performance of a Large-Scale, Particle-in-Cell Plasma Simulation Code. *Concurrency: Practice and Experience*, 1990.

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