Partitioning and Incremental Partitioning for Adaptive Irregular Problems

by
Chao-Wei Ou

Abstract of Dissertation
April 29, 1996

Parallelization of several scientific and engineering applications requires the partitioning of the underlying computational graph among processors such that the computational load on each node is roughly the same, while communication is minimized. Obtaining exact solutions is computationally intractable, since graph partitioning is an NP-complete problem.

For a large class of irregular and adaptive data parallel applications (such as adaptive graphs), the computational structure changes from one phase to another in an incremental fashion. In incremental graph-partitioning problems the partitioning needs to be updated as the graph changes over time; a small number of nodes or edges may be added or deleted at any given instant.

In this thesis we present new methods for partitioning and incremental partitioning of computational graphs, all of which are inherently parallel. We describe implementation results for our methods on several commercially parallel machines.

We have developed a parallel graph-partitioning software (SPRINT) which incorporates several of the techniques discussed here. It can be used either individually or in the runtime library for high performance languages. We describe the functionality of the SPRINT software.

Partitioning and Incremental Partitioning for Adaptive Irregular Problems

by
Chao-Wei Ou
Bachelor of Science, National Chiao-Tung University, Taiwan, May 1987
Master of Science, Syracuse University, Syracuse, NY, USA, August 1992

Dissertation

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer and Information Science
in the Graduate School of Syracuse University

April 29, 1996

Approved ____________________

Date ____________________

Acknowledgements

I would like to thank my thesis advisor, Dr. Sanjay Ranka, for his guidance and encouragement, Dr. Geoffrey Fox, Director of Northeast Parallel Architectures Center, for his insightful suggestions and support, and Dr. Kishan Mehrotra for his help. I would also like to thank Dr. Salim Hariri, Dr. Xiaoming Li, and Dr. Srinivas Aluru for their suggestions and comments proved valuable to my research.

I am grateful to Tseng-Hui Lin, from whom I have learned much about the art of system development the designing of software for parallel machines. The year that I spent working with Dr. Tom Haupt helped me learn the runtime aspects of a data parallel compiler.

In addition, I owe a lot to my family. This dissertation would not have been possible without my parents' support and advice. I appreciate the great support from my wife Su-Wan and my daughter Jessica. I would like to thank all my friends in Syracuse for their support during my long journey to my degree.

This work was supported in part by NSF under ASC-9213821, ARPA under contract #DABT63-91-C-0028, NAG-1485, and ONR under contract N00014-93-1-0158. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

From the perspective of parallel computing, a large number of applications can be represented as computational graphs. The nodes of a graph represent tasks that can be executed concurrently, while the edges represent the interaction between them. Informally, the parallelization of these applications requires partitioning the nodes of the graph such that each partition has equal computational load and the cross-edges are minimized. Graph partitioning has many important real-world applications, including domain-decomposition techniques for solving Partial Differential Equations (PDE), laying out circuits in VLSI, etc. In this thesis our main concentration is on the use of graph partitioning for mapping data-parallel applications on a coarse-grained parallel machine.

For a large class of irregular and adaptive data parallel applications [11], 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 change, 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 [8]. 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 [15], statistical physics simulations near critical points [14], the particle-sorting phase of a direct Monte Carlo simulation [16], unstructured multigrid [49], parallelized sparse triangular solver [3], particle-in-cell codes [44, 45, 68], adaptive mesh method [17] and shock profiling [4]. The key to efficient parallelization of these applications is in 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, partitioning 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 [8].
- Node Additions: New points may be added and/or old points deleted. This happens in the case of adaptive grids [11, 55].

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 vertices that are physically proximate. Examples of such applications include molecular dynamics [8], static and adaptive PDE solvers [11, 13], region-growing [15], component labeling [15], and statistical physics simulations [14]. A simple and fast heuristic for achieving partitioning for same graphs is to cluster physically proximate points in two or three dimensions. Important heuristics include recursive coordinate bisection, inertial bisection, scattered decomposition, geometry-based partitioners, and index-based partitioners [12, 18, 24, 29, 51, 52, 64, 71]. There are a number of methods that use explicit edge information to achieve better partitioning. Important heuristics include simulated annealing, mean-field annealing, recursive spectral bisection, recursive spectral multisection, mincut-based methods, and genetic algorithms [1, 26, 27, 31, 32, 46, 48, 59]. Although these heuristics can be applied to arbitrary graphs, empirical evidence of the quality of the partitioning has generally been limited to graphs from the physical domain.

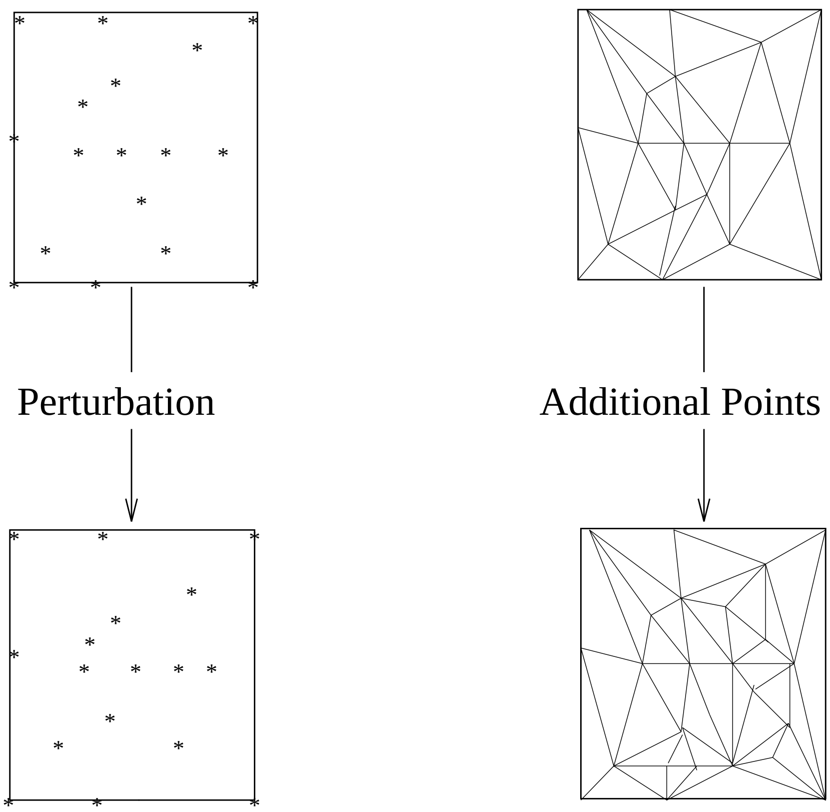


Figure 1: Incremental Aspects

The rest of this chapter is organized as follows. The graph partitioning problem is defined in Section 1. Section 2 describes two commonly used coordinate-based partitioning methods (recursive coordinate bisection and recursive inertial bisection), eigen-based partitioning methods (recursive spectral bisection, recursive spectral mutisection, and multilevel recursive bisection), and Neural-Network-based partitioning methods (simulated annealing and mean field annealing); Section 3 describes the model of parallel machine used in this thesis; and Section 4 shows the outline of the thesis.

1.1 Mapping Computational Graphs

Generally, the mapping process can be separated into two sub-processes, partitioning the graph and mapping the partitions onto the underlying architectures. The first process, graph partitioning, partitions the graph into partitions in which the interactions between two partitions is minimized. The second process assigns each partition to a processor such that the hop-weight is minimized. The detailed definitions of these two processes are described as follow.

1.1.1 Graph Partitioning

Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ represents a set of vertices, $\mathcal{E}$ represents a set of undirected edges, the number of vertices is given by $N = |\mathcal{V}|$, and the number of edges is given by $E = |\mathcal{E}|$. The weight w_i corresponds to the computation cost of the vertex v_i; $< v_i, v_j >$ denotes the edge, and the cost (or weight) of an edge $w_e(v_i, v_j)$ is given by the amount of interaction between vertices v_i and v_j.

The graph-partitioning problem can be defined as an assignment scheme $\mathcal{M} : \mathcal{V} \longrightarrow \mathcal{P}$, $\mathcal{P} = \{1, 2, \cdots, P\}$ that maps vertices to partitions. We denote by $\mathcal{B}(q)$ the set of vertices assigned to a partition q. Thus $\mathcal{B}(q) = \{v \in \mathcal{V} : \mathcal{M}(v) = q\}$. The weight of a partition can be defined as

$$\mathcal{W}(q) \;=\; \sum_{v_i \in \mathcal{B}(q)} w_i. \quad q = 1, 2, ..., P. \tag{1}$$

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

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

For the purpose of mapping on parallel machines, a number of different but related cost functions have been described in the literature [2, 33, 48]. Following are some cost functions that are minimized:

- 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 $MAX_q\ \mathcal{W}(q)$, which is typically approximated by $\sum_q (\mathcal{W}(q) - \mu)^2$, where $\mu = \frac{\sum_q \mathcal{W}(q)}{p}$. The communication cost is given by the sum of all the cross-edges (or the cut weight).

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

  Thus, the total cost is

$$\alpha W(q) + C. \tag{4}$$

  The above metric fails to take congestion into account [31]. Computational graphs generate communication in which several messages simultaneously compete for the same wire (the effect of this is small for reasons discussed in Chapter 5). Thus a potentially better metric is one that weighs the communication cost by the number of hops the message needs to traverse. Let h_{qr} denote the number of wires from processor q to processor r. The communication cost that takes the hop weight into account is given by

$$\sum_q \sum_r h_{qr} * \sum_{v_i \in \mathcal{B}(q)} \sum_{v_j \in \mathcal{B}(r)} w_e(v_i, v_j). \tag{5}$$

- The time required to execute the computational graph on a parallel machine is given by the processor that requires the maximum time. Some formulations approximate
$$\max_q \left(\mathcal{W}(q) + \beta\mathcal{C}(q)\right), \tag{6}$$
where β represents the ratio of cost of unit computation to the cost of unit communication on a machine.

We present results for a wide range of metrics for communication cost assuming the computational loads are nearly balanced. This allows for an easy comparison of schemes based on any of the above formulations.

1.1.2 Incremental Graph Partitioning

In incremental graph-partitioning problems, assume that a solution is available for a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ by using one of the many available methods in the literature; e.g., the mapping function $\mathcal{M}$ is available such that

$$|\mathcal{B}(0)| \approx |\mathcal{B}(1)| \approx |\mathcal{B}(2)| \approx \cdots \approx |\mathcal{B}(q-1)|, \tag{7}$$

and the communication cost is close to optimal. Let $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ be an incremental graph of $\mathcal{G}(\mathcal{V}, \mathcal{E})$
$$\mathcal{V}' = \mathcal{V} \cup \mathcal{V}_1 - \mathcal{V}_2 \quad \text{where} \quad \mathcal{V}_2 \subseteq \mathcal{V}, \tag{8}$$
i.e., some vertices are added ($\mathcal{V}_1$) and some vertices are deleted ($\mathcal{V}_2$). Similarly,

$$\mathcal{E}' = \mathcal{E} \cup \mathcal{E}_1 - \mathcal{E}_2 \quad \text{where} \quad \mathcal{E}_2 \subseteq \mathcal{E}, \tag{9}$$

i.e., some edges are added ($\mathcal{E}_1$) and some are deleted ($\mathcal{E}_2$). We would like to find a new mapping $\mathcal{M}' : \mathcal{V}' \longrightarrow \mathcal{P}$ such that the new partitioning is as load balanced as possible and the communication cost is minimized.

The methods described in this dissertation assume that $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$ is sufficiently similar to $\mathcal{G}(\mathcal{V}, \mathcal{E})$, i.e., the number of vertices and edges added/deleted are a small fraction of the original number of vertices and edges.

1.1.3 Architecture-Dependent Mapping of the Partitions

This process maps the partitions to the processors. Formally, $\mathcal{Z} : \mathcal{S}_\mathcal{V} \longrightarrow \mathcal{A}_\mathcal{V}$ represents a mapping from the supergraph $\mathcal{S}(\mathcal{S}_\mathcal{V}, \mathcal{S}_\mathcal{E})$ to the processor graph $\mathcal{A}(\mathcal{A}_\mathcal{V}, \mathcal{A}_\mathcal{E})$. When the communication cost can be estimated by the total number of cross-edges, this mapping can be performed easily by an arbitrary assignment that ensures that each processor gets one partition (For details, see Section 5).

When the hop weight is a better formulation of the communication cost, one of the following can be minimized:

- The average hop weight of the mapping given by

$$AveHopWeight = \frac{\sum_{<x,y> \in \mathcal{S}_\mathcal{E}} h_{\mathcal{Z}(x),\mathcal{Z}(y)} w_e(x,y)}{|\mathcal{E}|}. \tag{10}$$

- The maximum contribution to the hop weight by any processor given by

$$MaxHopWeight = \max_x \frac{\exists_{<x,y> \in \mathcal{S}_\mathcal{E}} h_{\mathcal{Z}(x),\mathcal{Z}(y)} w_e(x,y)}{|\mathcal{E}_x|}. \tag{11}$$

1.1.4 Communication and Crossedges

The communication cost on coarse grain parallel machines mainly consists of the latency overhead (start-up time) and the communication bandwidth. For small message sizes, the latency overhead is significant while for large message sizes, the total time is dominated by the size of the message and transmission rate (inverse of the bandwidth). Effective parallelization of graph based applications requires careful attention to both these parameters to reduce the communication overhead.

The traditional graph-partitioning attempts to partition the graph such that the number of vertices in each partition is approximately equal and the number of cross-edges is minimized. Although the latency overhead is ignored in the cost metric, reducing the number of cross-edges generally leads to "well-rounded" partitions. This generally leads to a small number of neighboring partitions for each partition. Thus, only a small number of messages are generated, reducing the latency overhead. Further, the size of the message generated is also limited to the number of boundary elements (Figure 2). Again, reduction in the number of cross-edges indirectly reduces the number of boundary elements.

A precise cost function for the communication cost can be defined for a given architecture which includes the latency overhead, the actual message size generated and congestion overhead (due to messages having paths which have one or more shared links) for a given architecture. However, such a metric is architecture dependent and does not directly allow for well known partitioning methods such as spectral methods which assume a simple (quadratic) minimization function. In this thesis we restrict the number of cross-edges as the main metric for comparison of our methods with the other well known methods. Although comparison based on this metric has limitations as described above, it is representative of more precise metrics for most cases.

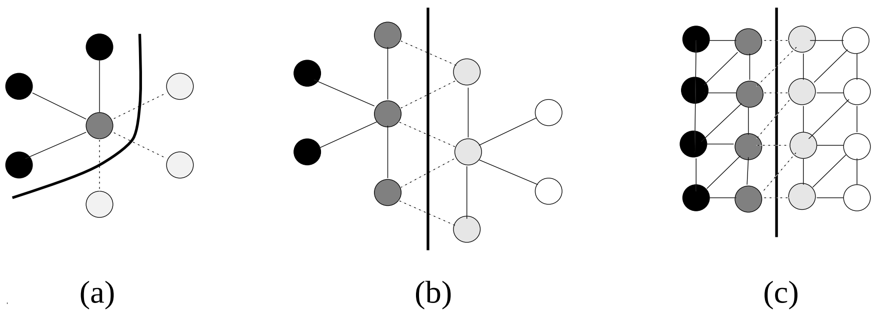


Figure 2: Scenarios of relation between cross-edges and communication data: a) 4 boundary vertices and 3 crossegdes; (b) 6 boundary vertices and 5 cross-edges; (c) 8 boundary vertices and 7 cross-edges.

1.2 Survey of Graph Partitioning Techniques

1.2.1 Coordinate-Based Partitioning

In a computational graph an interaction exists between two physically proximate nodes. Therefore, physical information can be used to explore the computational graphs. There are several coordinate-based methods available in the parallel computing community, including recursive coordinate bisection, recursive inertial bisection [64], recursive geometry bisection [29], etc. In this section we describe recursive coordinate bisection and recursive inertial bisection, which are commonly used in the parallel computing community.

Recursive Coordinate Bisection The recursive coordinate bisection orthogonally bisects the graph along the largest dimension into two almost equally weighted subgraphs and recursively bisects these two subgraphs continuously along the longest dimension. The key implementation of this scheme is to find the median vertex quickly at the current dimension. The common usage of recursive coordinate bisection is in finding the median vertex recursively. The median vertex divides the graph into two subgraphs based on the geometry of the graph and calculates the number of vertices or the weight for both subgraphs. The subgraph with the larger number of vertices or greater weight will be divided into another two subgraphs until the median vertex is found. If the graph contains some vertices with the same coordinates at the current dimension, the tie can be broken by arbitrarily or randomly assigning those vertices to one of two subgraphs. An example is shown in Figure 3 to partition the graph into 4 partitions.

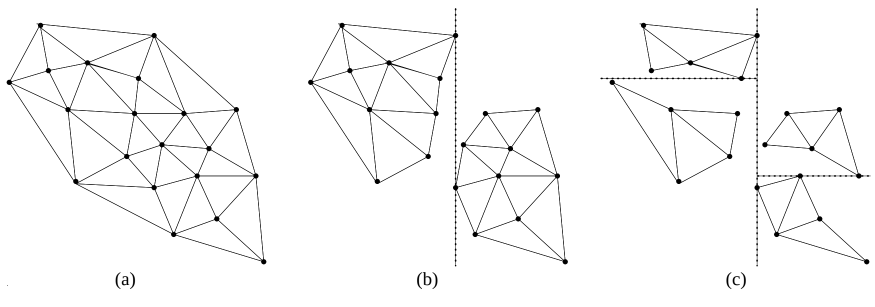


Figure 3: An example of partitioning the graph into 4 partitions by RCB: (a) the initial graph; (b) partitions by the first recursive step; (c) the partitioned graph after RCB.

Recursive Inertial Bisection Recursive inertial bisection divides the graph along the longest extension (the principle axis) into two subgraphs to avoid long and thin subgraphs. RIB first calculates and diagonalizes the initial tensor of centroid (the center of mass) to find the principle axis, which is the longest extension. All vertices project to this axis to form a linear vector. This vector is bisected at the median to two sub-vectors, such that the element values in the lower vector are less than those in the higher vector. According to these two sub-vectors, two subgraphs can be formed. The same process is applied to the subgraphs recursively until the desired number of partitions is achieved.

To find the principle axis corresponds to solve the eigen value of the tensor, then to calculate the eigen vector according to the largest eigen value. In parallel implementation this step can be done synchronously, since the size of tensor is small (corresponding to the number of dimensions). Finding the median of the vector will be the same as the method in recursive coordinate bisection. An example is shown in Figure 4 to partition the graph into 4 partitions.

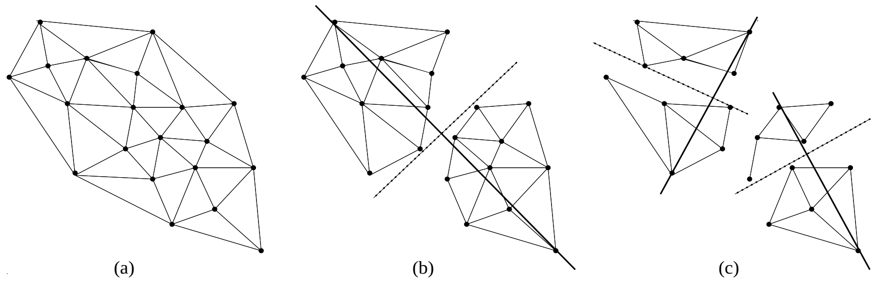


Figure 4: An example of partitioning the graph into 4 partitions by RIB: (a) the initial graph; (b) partitions of the first recursive step (c) the partitioned graph after RIB. The solid-thick lines represent the principle axis.

1.2.2 Eigenvector-Based Partitioning

There are several recently developed important heuristic methods that use connectivity information to partition graphs. This set of partitioning methods generates better quality partitioning than do coordinated-based partitioning methods. The computational graph can be represented as a connectivity matrix (usually the matrix is symmetric). There are several important theorems of the eigenvector of this matrix that can be applied to solve the graph-partitioning problem [50]. We describe several methods that are based on the properties of the eigenvector.

Recursive Spectral Bisection This heuristic graph-partitioning method was recently proposed by Simon [64], and the ideas have been revived and further developed by several researchers [5, 2, 31, 32, 33, 59, 60].

Assuming the size of the vertex set is even, Spectral bisection can be defined as follows. The graph is defined as in Section 1.1, with x_i assigned to each v_i such that $x_i = \pm 1$ and $\sum_{v_i} x_i = 0$. This condition represents $\mathcal{V}$ as divided into two equal-sized partitions. Function $f(x) = \frac{1}{4}\sum_{<v_i,v_j>\in\mathcal{E}}(x_i - x_j)^2$ counts the number of edges crossing the partitioned sets, if x_i and x_j have the same sign (x_i and x_j belong to the same partition), nothing is contributed in $f(x)$; otherwise, 4 is contributed. The objective function $f(x)$ to be minimized can be transformed into a matrix representation as follows. The adjacency matrix A of a graph can defined as

$$A_{ij} = \begin{cases} 1 & : \quad (v_i, v_j) \in \mathcal{E}, \\ 0 & : \quad \text{otherwise.} \end{cases} \tag{12}$$

The matrix degree of graph can be defined as $D = diag(d_i)$, where d_i is the degree of v_i.

$$\sum_{<v_i,v_j>\in\mathcal{E}} (x_i - x_j)^2 = \sum_{<v_i,v_j>\in\mathcal{E}} (x_i^2 + x_j^2) - \sum_{<v_i,v_j>\in\mathcal{E}} 2x_i x_j \tag{13}$$

can be recast as the terms

$$\sum_{<v_i,v_j>\in\mathcal{E}} (x_i^2 + x_j^2) = \sum_{<v_i,v_j>\in\mathcal{E}} 2 = 2|\mathcal{E}| = \sum_{v_i} x_i^2 d_i = x^T D x \tag{14}$$

and

$$\sum_{<v_i,v_j>\in\mathcal{E}} 2x_i x_j = \sum_{v_i}\sum_{v_j} x_i A_{ij} x_j = \sum_{v_i} x_i \sum_{v_j} A_{ij} x_j = x^T A x. \tag{15}$$

The *Laplacian* of the graph can be defined as $L(G) = D - A$, where

$$L_{ij} = \begin{cases} -1 & : \quad (v_i, v_j) \in \mathcal{E}, \\ d_i & : \quad i = j, \\ 0 & : \quad \text{otherwise.} \end{cases} \tag{16}$$

The objective function can be rewritten as

$$\text{Minimize} \quad : \quad \frac{1}{4} x^T L x \tag{17}$$

$$\text{Subject to} \quad : \quad x^T \mathbf{1} = 0, \quad x^T x = n, \tag{18}$$

where $\mathbf{1}$ is the n-vector $(1, 1, 1, ...)^T$.

The Laplacian matrix is positive semidefinite. Its smallest eigenvalue is zero and the associated eigenvector is all ones. If Graph G is connected, the second smallest eigenvalue λ_2 is positive. The eigenvector associated with λ_2 (the Fiedler vector) contains important directional information about the graph [21, 23, 22]. The information of the components of x_2 explores the distances between vertices in the graph. The normalized eigenvectors u_1, u_2, ..., u_n with corresponding eigenvalues $\lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$ are pairwise orthogonal. It can be shown that $f(x) \geq \frac{1}{4} n \lambda_2$. By choosing $x = \sqrt{n} u_2$, $f(x) = \frac{1}{4} n \lambda_2$. The Lanczos method is commonly used to calculate the Fiedler vector of Laplacian matrix. Graph bisection can be achieved by mapping the vertices to two subsets according to the median of the x_i. The entire process is repeated until the desired number of partitions is achieved. Several extensions of the spectral method are described in the next sections.

Spectral Quadrisection To partition the graph into four sets, we need another vector, y, so that the graph can be bisected on each vector. The objective function can be written as $f(x, y) = \frac{1}{4}(x^T L x + y^T L y)$. The constraints $x^T \mathbf{1} = 0$ and $y^T \mathbf{1} = 0$ ensure the load balance in the x vector and y vector, respectively. In quadrisection more constraints need to be added and are shown as follows.

$$\text{Minimize} \quad : \quad \frac{1}{4}(x^T L x + y^T L y) \tag{19}$$

$$\text{Subject to} \quad : \quad x^T \mathbf{1} = y^T \mathbf{1} = x^T y = 0 \quad x^T x = y^T y = n. \tag{20}$$

The solution can be acquired easily by choosing x and y to be eigenvectors of L. The minimum possible value of $f(x, y) = \frac{1}{4}n(\lambda_2 + \lambda_3)$ is obtained by choosing $x = \sqrt{n}u_2$ and $y = \sqrt{n}u_3$.

Hendrickson and Leland [33] choose $x = \sqrt{n}u_2 \cos\theta + \sqrt{n}u_3 \sin\theta$ and $y = \sqrt{n}u_2 \sin\theta + \sqrt{n}u_3 \cos\theta$ where θ can be chosen by minimizing $g(\theta) = \sum_{v_i}(1 - x_i^2)^2 + (1 - y_i^2)^2$ over $\theta \in (0, 2\pi)$. They claim that the choice of θ can reach global optimization but needs a low cost of local optimizations to improve the quality of the partitions.

After x and y vectors are obtained, the simple median-finding method is no longer useful in the quadrisection. A new technique was introduced by Tokuyama and Nakano [67] that assigns one fourth of the vertices to each partition.

Spectral Octasection The idea of spectral octasection is close to that of the spectral quadrisection. The objective function and constraints can be described as follows.

$$\text{Minimize} \quad : \quad \frac{1}{4}(x^T Lx + y^T Ly + z^T Lz) \tag{21}$$

$$\begin{aligned}\text{Subject to} \quad : \quad & x^T\mathbf{1} = y^T\mathbf{1} = z^T\mathbf{1} = x^T y = x^T z = y^T z = \sum_{v_i} x_i y_i z_i = 0, \\ & x^T x = y^T y = z^T z = n.\end{aligned} \tag{22}$$

One solution is that $x = \sqrt{n}u_2$, $y = \sqrt{n}u_3$, and $z = \sqrt{n}u_4$ with the corresponding minimum of $f(x, y, z) = \frac{1}{4}n(\lambda_2 + \lambda_3 + \lambda_4)$. There are three degrees of freedom involved in the computation since it is a 3-dimensional problem. The process of assigning one eighth of vertices to each partition is similar to the assigning process of quadrisection. This method can be generalized to higher dimensional spectral bisection.

Multilevel Recursive Spectral Bisection Spectral graph-partitioning methods involve the solving of the eigenvalues and eigenvectors on a symmetric Laplacian matrix. Since the cost of solving eigen-based problems is at least one to two orders of magnitude higher than the cost of coordinate-based partitioning methods, a new technique is introduced to improve the performance of eigen-based methods by using multigrid implementations. This method tries to contract the original huge graphs to smaller working graphs (coarse graphs) to reduce the cost of solving eigen-based problems. When the mapping of the coarse graph is obtained, a local refinement technique is applied to map it back to the original graph. The multilevel recursive spectral bisection algorithm can be separated into four phases, contraction, Fiedler vector, interpolation, and refinement, in each recursion step. The algorithm is described as follows.

1. **Contraction** In order to create the contracted graph $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ from the graph $\mathcal{G}$, $\mathcal{V}'$ is chosen to be the maximum independent set with respect to $\mathcal{G}$ such that $\forall v_i \in \mathcal{V}'\ (v_i, v_j) \in \mathcal{E} \Longrightarrow v_j \notin \mathcal{V}'$. A maximum independent set can efficiently be found by applying a simple greedy algorithm. A sparse matrix can be created from the contracted graph $\mathcal{G}'$. During recursion the subgraphs can be disconnected, but the contraction process will preserve the connectivity of the sub-contracted graphs. This will affect the correctness of the Fiedler vector.

2. **Fiedler vector** The major cost of spectral bisection is dominated by the Lanczos iterations because of the size of the initial graph $\mathcal{G}$. The Lanczos algorithm is used in this implementation to solve the Fiedler vector of the contracted graph $\mathcal{G}'$ instead of $\mathcal{G}$. Since the size of the contracted graph is much less than that of the initial graph, the cost of finding the Fiedler vector can be greatly reduced.

3. **Interpolation** Given a Fiedler vector $F' =< f'_i >$, $i = 1..|\mathcal{V}'|$ of a contracted graph $\mathcal{G}'$, the interpolation process constructs an expansion vector, $F^0 =< f^0_i >$, $i = 1..|\mathcal{V}|$ that can be used as an approximation of the Fiedler vector of $\mathcal{G}$. The first step in the interpolation process is injection to place the components of F' to the first seeds (the starting seeds in the contraction process) in F^0. The second step is to set the rest of the elements of F^0 by averaging the elements of their neighbors that have been set by injection.

4. **Refinement** The Rayleigh quotient iteration (RQI) [56, 57] is used to refine the approximate Fiedler vector. The Rayleigh quotient iteration algorithm can take advantage of a good initial approximation to obtain a refined vector, since

the Lanczos algorithm lacks its advantage. The SYMMLQ [53] implementation is used for very ill-conditioned matrices and it terminates early when a good approximation is found. The combination of RQI and SYMMLQ has proved to be efficient for refining the coarse grid approximation of an eigenvector [2].

1.2.3 Neural-Networks and Related Partitioning Methods

In this section we describe two commonly neural-network-based methods used in the graph-partitioning problem. The common energy function to be minimized can be defined as

$$\mathcal{H} = \mathcal{H}_{calc} + \mu \mathcal{H}_{comm}, \tag{23}$$

where H_{calc} represents calculation cost and H_{comm} represents communication cost.

Hopfield Neural Network The Hopefield Neural Network (HNN) [36, 37, 38] is one of the models most commonly used for optimization problems, since the graph-partitioning is an optimization problem. Hopfield networks iteratively update spin values to decrease the energy function. A Hopfield-network energy function is

$$\mathcal{H}(s) = -\frac{1}{2}\sum_i \sum_j c_{ij} s_i s_j + \sum_i T_i s_i, \tag{24}$$

where s_i, w_{ij}, and T_i represent the ith spin which is either 1 or -1, the weight of the connection from the ith spin to the jth spin, and the external input to the ith spin of the network. The spin s_i is chosen to be updated as

$$s_i^{new} = sgn\left(\sum_j c_{ij} s_j^{old} - T_i\right), \tag{25}$$

such that the energy function is decreasing. The change of energy is

$$\Delta\mathcal{H} = -net_j(s_j^{new} - s_j^{old}), \tag{26}$$

where $net_j = \left(\sum_{i \neq j} c_{ij} s_j^{old} + T_j\right)$. The change of the spin s_j can be described as follows:

$$s_j^{new} = \begin{cases} 1 & : \quad (s_j^{old} = -1 \text{ and } net_j > 0), \\ -1 & : \quad (s_j^{old} = 1 \text{ and } net_j < 0), \\ s_j^{new} = s_j^{old} & : \quad \text{otherwise.} \end{cases} \tag{27}$$

Simulated annealing Simulated Annealing (SA) [25, 28, 30, 42] is a general optimization method to solve difficult combinatorial optimization problems. SA simulates the slow cooling of a physical system to reach the stabilized state.

Initialization for initial mapping and T_0
while ($T_0 > 0$ **and** (**not** converged)) **do**
  repeat
    Perturb the solution
    if ($\Delta\mathcal{H} \leq 0$) **then**
      Accept the solution
    else
      if ($Probability$ ¡ $e^{\frac{-\Delta\mathcal{H}}{T}}$) **then**
        Accept the solution
      else
        Reject the solution
  until Equilibrium
  $T_{i+1} = f\ T_i$
end while

There is a cost function $\mathcal{H}$ (a Hamiltonian) which associates a T (temperature) and various ways to change the state of the system. SA iteratively proposes changes to evaluate the $\Delta\mathcal{H}$ in $\mathcal{H}$ such that each change is either accepted or rejected by the *Metropolis* criterion; if $\Delta\mathcal{H} < 0$ the change is accepted; otherwise the acceptance is determined by the probability $e^{\frac{-\Delta\mathcal{H}}{T}}$. SA hopefully reaches the global optimum by starting at a high temperature, exploring the state space, and gradually decreasing the temperature to zero. It can be shown that if the temperature decreases sufficiently slowly, then the probability of being in a global optimum is certain [30]. The SA process is described as follows.

Mean Field Annealing The Mean Field Annealing (MFA) algorithm can be used to solve combinatorial optimization problems. MFA is essentially a combination of the Hopfield Neural Network and Simulated Annealing. MFA is similar to HNN in the respect that the spin or neural variables evolve iteratively. The MFA algorithm is derived by making an analogy to the Ising spin model that is used to estimate the state of a system of spins in thermal equilibrium. MFA was first proposed to solve the traveling salesman problem [6] and then was applied to the graph-partitioning problem [9, 10, 58, 6]. In the Ising spin model the energy of a system with N spins can be defined as

$$\mathcal{H}(s) = \frac{1}{2}\sum_{i=1}^{N}\sum_{j\neq i}\beta_{ij}s_i s_j + \sum_{i=1}^{N} h_i s_i, \tag{28}$$

where β_{ij} represents the interaction between spins i and j, and $s_i \in 0, 1$. The spin average $\langle s_i\rangle$ can be computed using Boltzmann distribution

$$\langle s_i\rangle = \frac{1}{1 + e^{\frac{-\phi_i}{T}}}, \tag{29}$$

where $\phi_i = \langle\mathcal{H}(s)\rangle|_{s_i=0} - \langle\mathcal{H}(s)\rangle|_{s_k=1}$ indicates the *mean field* acting on the ith spin and

$$\langle\mathcal{H}(s)\rangle = \sum_{i=1}^{N}\sum_{j\neq i}\beta\langle s_i s_j\rangle + \sum_{i=1}^{N} h_i\langle s_i\rangle. \tag{30}$$

Since $\langle\mathcal{H}(s)\rangle$ is linear in $\langle s_i\rangle$, the mean field ϕ_i can be computed as

$$\phi_i = \langle\mathcal{H}(s)\rangle|_{s_i=0} - \langle\mathcal{H}(s)\rangle|_{s_i=1} = -\frac{\partial\langle\mathcal{H}(s)\rangle}{\partial\langle s_i\rangle} = -\left(\sum_{j\neq i}\beta_{ij}\langle s_j\rangle + h_i\right). \tag{31}$$

The MFA algorithm starts at a high temperature, while the system is not stabilized at the current temperature, an ith spin is randomly selected, and the mean field acting on its computations until the system is stabilized for the current temperature. The new spin value is computed by using this mean field and the current temperature. The temperature is then lowered and the above processes are repeated until the temperature is freezing. The result spin value at the freezing temperature is the solution to the system.

Bultan and Aykanat defined the energy function as

$$\mathcal{H}(s) = \frac{1}{2}\sum_{i=1}^{N}\sum_{j\neq i}\sum_{p=1}^{P}\sum_{q\neq p} e_{ij}s_{ip}s_{jq}d_{pq} + \frac{R}{2}\sum_{i=1}^{N}\sum_{j\neq i}\sum_{p=1}^{P} s_{ip}s_{jp}w_iw_j. \tag{32}$$

Detailed descriptions and experimental results can be found in Bultan and Aykanat's paper [10].

1.3 Modeling a Parallel Machine

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 μ. For our complexity analysis we assume that τ and μ are constant and independent of link congestion and distance between two processors. With new techniques, such as wormhole routing and randomized routing, the distance between communicating processors is less of a determining factor on the amount of time needed to complete 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.

Sending a Message Assume that sending a message from one processor 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, assuming that there is no contention at the receiving node [7].

Global Concatenation Assume that each processor has $\frac{N}{P}$ elements. Each processor contains a vector $V_i[0 \cdots \frac{N}{P} - 1]$. A global operation computes a concatenation of the local list in each of the processors. The resultant vector $R[0 \cdots N-1]$ is stored in all the processors.

$$R[j] = V_{j \text{ div } P}[j \bmod P]. \tag{33}$$

This operation can be completed in $O(\phi_1 N)$ time on the CM-5, where ϕ_1 is a constant [7].

Global Combine Assume that each processor contains a vector $V_i[0 \cdots N-1]$. The global combine 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:

$$R[j] = \sum_{i=0}^{P-1} V_i[j]. \tag{34}$$

This operation can be completed in $O(\phi_2 N)$ time on the CM-5, where ϕ_2 is a constant [7].

All-to-Many Personalized Communication 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. Several strategies for performing this operation have been discussed in [61, 63]. Assuming that the maximum outgoing and incoming traffic (sum of the size of messages) is T on every processor, this can be completed in $(P\tau + \mu T)$ when $T \geq O(P^2 + P\frac{\tau}{\mu})$.

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 assuming that a concatenation of all these arrays in ascending order of the processor number is also sorted. The load on each processor is balanced such that the global ordering of 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. 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 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 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 $Z[0..P-1] = \textbf{Concatenate}(X_i)$

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 5: Order-Maintaining Load Balance Algorithm.

1.4 Outline of the Dissertation

In Chapter 2 we propose a new and fast graph-partitioning method (Index-Based Partitioning technique) corresponding to the physical information of the graph. The Index-Based Partitioning method (IBP) maps a high-dimensional graph to a linear representation of indices. IBP performs sorting to rearrange the linear list in ascending order and arbitrarily divides the list into a number of sublists. Each sublist represents the partition in the graph. We study the effectiveness of this method both from the view of static partitioning and from adaptive partitioning.

In Chapter 3 we present a set of remapping methods for adaptive applications based on IBP. Generally, adaptive applications can be categorized into two sets of problems, perturbation and addition/deletion. For both cases of adaptive applications, IBP provides a sorted precedent list that can be effectively used in solving the almost-sorted list and a small number of additional elements. From the application point of view, most perturbed vertices in the graph change slightly when compared to the preceding graph. We developed an incremental sorting algorithm, which corresponds to a graph after perturbation, to quickly sort the almost sorted list, and the cost of this sorting is only a fraction of the cost of using IBP from the scratch. For the applications with additional vertices, we develop several algorithms to quickly merge the additional vertices into the precedent list. We study the impact of perturbation and addition on graphs and derive several effective algorithms for adaptive applications.

In Chapter 4 we present a remapping algorithm which partitions the graph by using the connectivity information. This method can effeciently solve the partitioning problem for adaptive applications when starting at non-index-based partitioning methods. Since the costs of good-quality partitioning techniques are high, these partitioners cannot be used well in adaptive applications. This remapping algorithm applies the linear programming method to solve the incremental problems. Further, we improved this method by using a multilevel approach.

In Chapter 5 we present a method to take the hardware designation into account to generate a better partitioning according to the architecture. In the previous chapters we proposed several methods to solve the graph-partitioning problem. We showed that this method can solve the graph-partitioning problem on the underlying parallel architecture.

In Chapter 6 we develop a graph-partitioning library (SPRINT) to effectively solve irregular and/or adaptive applications. SPRINT includes improved, commonly used coordinate-based partitioners and index-based partitioners.

Conclusions and directions of future research are given in Chapter 7. All graphs used in this thesis can be found in the Appendix.

Chapter 2

Index-Based Partitioning

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. Index-based mapping schemes for partitioning two- and three-dimensional irregular grids on parallel machines are described. We show these methods to be extremely fast and easy to parallelize. They produce good-quality mappings and are useful for a wide variety of irregular problems.

Graphs with these properties can be transformed into architecture-independent representations that encapsulate the locality in these graphs. This representation allows a fast mapping of the computational graph onto the underlying architecture at the time of execution. This is necessary for environments where available computational resources can be determined only at the time of execution or that change during execution.

2.1 Index-Based Mapping

The assumption is made that the vertices of the computational graph are embedded in a d-dimensional space. 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 [66, 69]. 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 to represent the bin to which a vertex belongs along the ith dimension is given by l_i. Each vertex is now represented by an integer tuple $(cord_1, cord_2, ...cord_d)$, which is transformed into a one-dimensional index. Consider a graph in which the set of vertices are arranged in an 8×8 grid. Row-major indexing, z-Curve indexing, and the Hilbert Space-Filling Curve are three of several ways to index pixels in a two-dimensional grid (Figure 2). These three indexing schemes are shown in Figures 1 (a), (b), and (c), respectively, and can be described as follows.

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

(a)

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

(b)

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

(c)

Figure 1: Different indexing schemes for an 8×8 image: (a) Row-Major, (b) z-Curve, and (c) Hilbert Space-Filling Curve.

1. **Row-Major Indexing** This indexing scheme is straightforward. It indexes each node in the graph in row-major fashion, which assigns the indices to the nodes starting at the lower-left corner and ending at the upper-right corner in increasing order. This method maintains the proximity relation along the only dimension. From the perspective of the irregular application, the node has interactions with its neighbor nodes in any dimension instead of interacting with neighboring nodes at the same dimension.

2. **z-Curve Indexing** This is an algorithm for mapping a graph using a bit-interleave-based index. It 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. Figure 2 shows the basic indexing curve on a two-dimensional grid, and Figure 3 shows the mapping from a two-dimensional grid to a one-dimensional z-curve index.

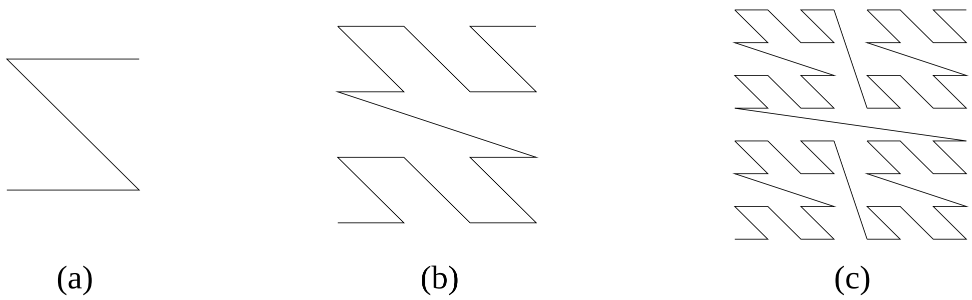


Figure 2: z-curve: (a) shows the basic curve on a 2×2 grid; (b) shows the curve on a 4×4 grid; (c) shows the curve on a 8×8 grid.

Bit-interleaving is one of the many transformations 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.

3. **Hilbert Space-Filling Curve Indexing** This is another well-known indexing method to map a graph to a linear representation. It recursively assigns indices to four subregions on a two-dimensional grid according to the information of the previous regions. Details can be found in the paper [35]. Figure 4 shows the basic indexing curve on a two-dimensional grid, and Figure 5 shows the mapping from a two-dimensional grid to a one-dimensional Hilbert space-filling curve index.

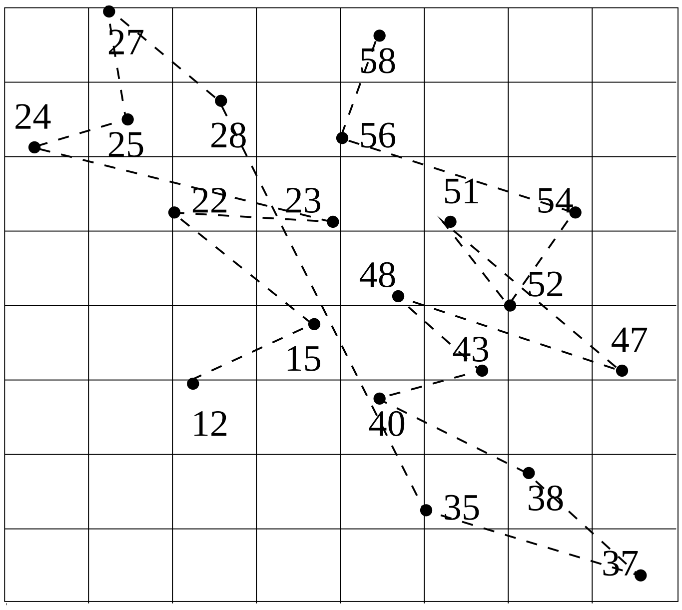


Figure 3: Mapping from a two-dimensional grid to a one-dimensional z-curve index.

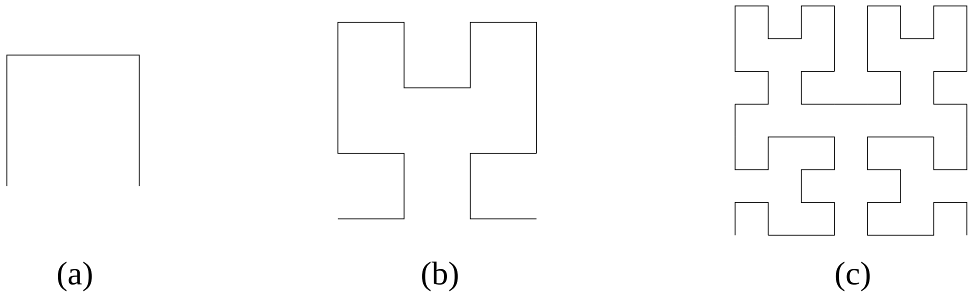


Figure 4: Hilbert space-filling curve: (a) shows the basic curve on a 2×2 grid; (b) shows the curve on a 4×4 grid; (c) shows the curve on a 8×8 grid.

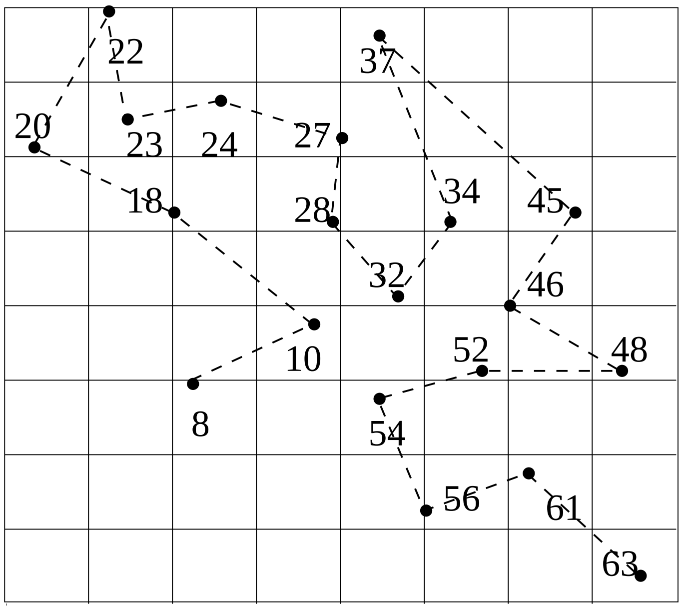


Figure 5: Mapping from a two-dimensional grid to a one-dimensional Hilbert space-filling curve index.

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.

2.2 Experimental Results

2.2.1 Sequential 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.

All previous partitioning schemes are applied to several graphs on a 40-MHz SUN4

Graph	Graph 1	Graph 2	Graph 3	Graph 4	Graph 5
$\lvert\mathcal{V}\rvert$	6019	9428	10166	15606	53961
$\lvert\mathcal{E}\rvert$	17473	59863	30471	45878	353476

Table 1: Experimental graphs

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

Table 2: 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	0.089	0.159	0.169	0.249	1.009
	RCB	0.116	0.183	0.199	0.316	1.033
	MRSB	3.369	8.149	5.369	7.049	48.353
8	IBP	0.089	0.159	0.169	0.249	1.009
	RCB	0.166	0.249	0.266	0.416	1.483
	MRSB	5.319	13.098	10.974	12.182	69.817
16	IBP	0.089	0.159	0.169	0.249	1.009
	RCB	0.199	0.316	0.349	0.516	1.899
	MRSB	8.479	17.063	20.081	19.966	93.319
32	IBP	0.089	0.159	0.169	0.249	1.009
	RCB	0.249	0.399	0.433	0.649	2.333
	MRSB	14.401	26.763	30.730	30.317	120.355
64	IBP	0.089	0.159	0.169	0.249	1.009
	RCB	0.299	0.483	0.499	0.783	2.733
	MRSB	23.487	33.233	50.677	50.091	155.619

Table 3: Execution time in seconds for IBP, RCB, and MRSB on a SUN4

with 60 MB memory. Results for five representative graphs in Table 1 on a SUN4 workstation are presented in Tables 2 and 3. The time required for index-based partitioning is independent of the number of partitions. For large graphs, as well as a 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 inferior to spectral methods.

2.2.2 Parallel Results

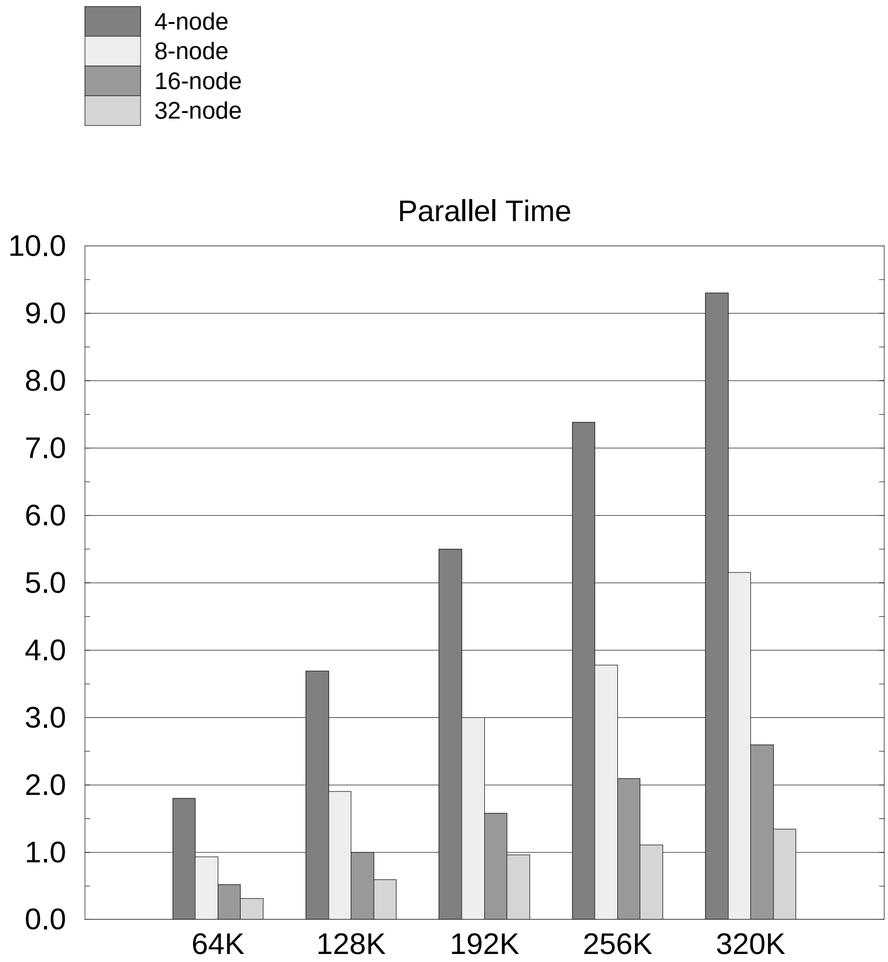


Figure 6: Timing for parallel mapping algorithm on 4, 8, 16, and 32 processors

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.

Table 4 gives the execution time for Graphs 1, 3, and 4 on a one-node CM-5. The times for a different number of processors (and corresponding number of partitions) are given in Table 5. 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 of the sorting algorithm. It corresponds to 41%, 34%, and 30% of the total time for Graphs 1, 3, and 4 for 32 partitions on a 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 to remap all the vertices based on index-based partitioning. As the data size increases, this fraction reduces significantly. Table 5 also gives the parallel execution time for the RSB-based partitioner [39] (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 using the SPARC chip only) is about two to three orders of magnitude faster when vector units are not used than RSB-implemented (which in the CMSSL library), depending on the size of the graph. Even when vector units are used, the performance is approximately two orders of magnitudes better.

[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 numbers of finite elements for Graphs 1, 3, and 4 are 11451, 20232, and 30269, respectively.

Graph 1	Graph 3	Graph 4
0.276	0.611	0.880

Table 4: Sequential execution time in seconds for IBP on 1-node CM-5

	Partitioner	4-node	8-node	16-node	32-node
Graph 1	IBP	0.151	0.094	0.068	0.051
	RSB	10.086	13.084	15.317	17.381
	RSB/VU	4.284	5.524	6.396	7.218
Graph 3	IBP	0.254	0.159	0.103	0.069
	RSB	14.009	22.206	28.393	33.951
	RSB/VU	5.563	9.076	10.961	13.419
Graph 4	IBP	0.356	0.211	0.121	0.096
	RSB	27.407	40.573	50.961	63.619
	RSB/VU	7.860	11.983	15.224	18.089

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

2.3 Conclusions

In this chapter a simple index-based mapping algorithm is proposed for mapping computational graphs. The experiments show that these methods can

1. provide good solutions with a relatively low cost,

2. be parallelized, and

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

Chapter 3

Index-Based Incremental Partitioning

In this chapter we present fast parallel algorithms for remapping adaptive problems on coarse-grained distributed memory machines when index-based partitioning is initially used. We show that the remapping of these applications [55, 54], using a 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.

The rest of the chapter is organized as follows. Section 1 describes the important parallel primitives required for the remapping algorithms in Section 2. These primitives provide a level of architectural independence for our algorithms. Section 3 describes the results for a number of data sets on the CM-5 for the different algorithms. We present our conclusions in Section 4.

3.1 Incremental Partitioning for Perturbation

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 physically proximate points. Assuming index-based mapping is used to partition these points, the corresponding index is expected to change by a small amount, though this is not always the case. (For example, the index of (3, 3) is 15, while the index for (3, 4) is 26).

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 average) by adding or subtracting a small random number.

We would like to develop an algorithm to sort the new list B efficiently by using 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 1, 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 sized $local\_bound$ 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, to a different bucket on the same processor, or to 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 keys will be of Type 1, and an even smaller fraction of the keys will be of Type 2. For this reason, a search algorithm that is biased towards the current position of the key is used.

The algorithm described in Figure 1 reflects this bias. Checks are made for keys of Type 0 before the search for Type 1, followed by the search for Type 2. We opted to choose an interpolation search over a binary search for the same reasons. Type 0 and Type 1 keys are added to local buffers representing the different buckets on the same 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 assumes 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 search is $O(\log \log n)$ on an average. In many practical cases 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. The complexity of Steps 2 and 4 can therefore be approximated by $O(\frac{n}{p})$. The complexity of Step 5 will depend on

---

/* $Bound[i]$ is the largest key of A_i. */

/* $local\_bound[j]$ is the largest key of A_i in $[\frac{(j-1)n}{pl}, \frac{jn}{pl})$, 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 1: Bucket-Based Incremental Sorting Algorithm

---

l and on 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 the 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.

3.1.1 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, 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 $[0,\ 2\ \pi)$. Using these three random numbers, the values of the three coordinates were calculated in the following fashion.

$$x_{i,j} = x_{i,j} + \triangle x_{i,j},$$

where

$$\triangle x_{i,0} = \lambda_i r \sin(\theta_i),$$
$$\triangle x_{i,1} = \lambda_i r \cos(\theta_i) \sin(\phi_i),$$

and

$$\triangle x_{i,2} = \lambda_i r \cos(\theta_i) \cos(\phi_i).$$

The experimental results in Figure 2 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 indexing can also be substantially reduced by using table look-up methods.

3.2 Incremental Partitioning for Addition or 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. The following

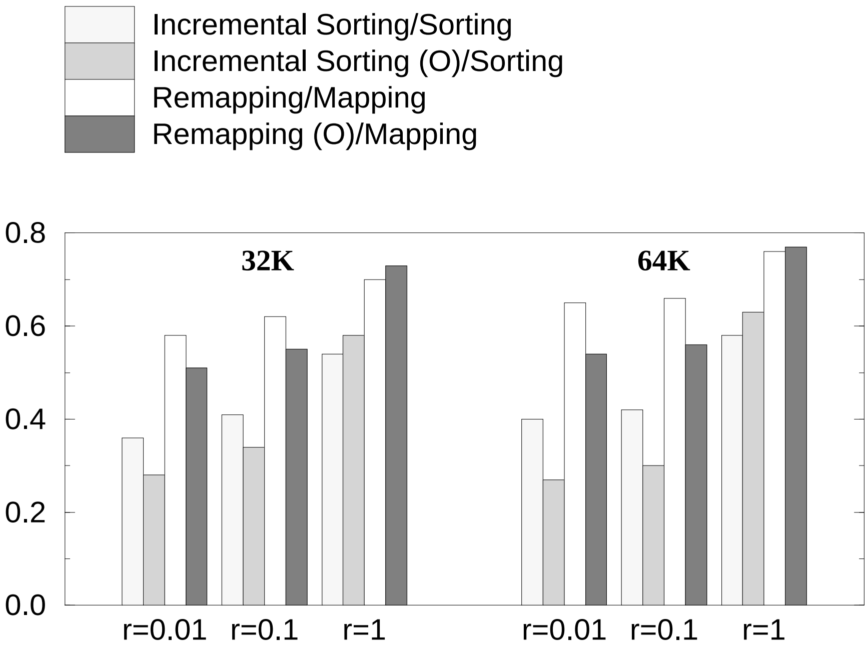


Figure 2: The performance of different remapping algorithms compared with mapping using the index-based scheme. (O) indicates that an optimization was used.

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 $m \ll n$.

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 [62]. The cost of this load-balancing scheme is nominal compared to the cost of merging for most cases.

Most of the analyses provided in the following section correspond 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 in 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.

3.2.1 Simple Merging Algorithm

A simple merging algorithm is given in Figure 3. 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. All-to-many communication algorithms have been described in Chapter 1.3. 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 Steps 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

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

/* 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**

$\quad\quad send\_list[k] := nil$

Step 2 : **For** $k \longleftarrow 1$ **to** m_i **do**

$\quad\quad proc :=$ **Binary_search**$(B_i[k], Bound)$

$\quad\quad$ 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 3: Simple Merging Algorithm

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 : $COUNT[1..pq]$:= **Global_Combine** ($count[1..pq]$)

Step 5 : **For** $j \longleftarrow 1$ **to** pq **do**

$SUM[j] := \frac{jn}{pq} + \sum_{k=1}^{j} COUNT[k]$

Step 6 : **For** $k \longleftarrow 0$ **to** $p-1$ **do**

$Coarse\_Bound[k] := SUM[j]$, where $\min_{j=1}^{pq}$ **abs**($SUM[j] - \frac{k(n+m)}{p}$)

Step 7 : Perform **Order Maintaining Data Movement** using $Coarse\_Bound$

/* Stage 2 : Merging phase */

Step 8 : Apply **Simple Merge** /* See Figure 3 */

Figure 4: Bucket-Merging Algorithm

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.

3.2.2 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 elements of the sublist of one processor, the effect will be of that processor receiving all the messages as well as sorting the elements received. This algorithm attempts to improve the worst-case scenario of the previous algorithm (Figure 3). A high-level description of the bucket-merging algorithm is given in Figure 4. 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 is again $O(m\log m + (p\tau + \mu m) + \frac{n}{p})$. 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 a potentially better performance in the second stage.

3.2.3 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 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 5. 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 ($\frac{n}{p}$ and $\frac{m}{p}$, 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 underlying 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 an **Order Maintaining Data Movement** operation on B. The cost of these operations is difficult to analyze and depends 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

/* 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 δ.

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 5: Parallel algorithm for merging two sorted lists

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.

3.2.4 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, the simple merge algorithm performs the best (Figure 6). 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

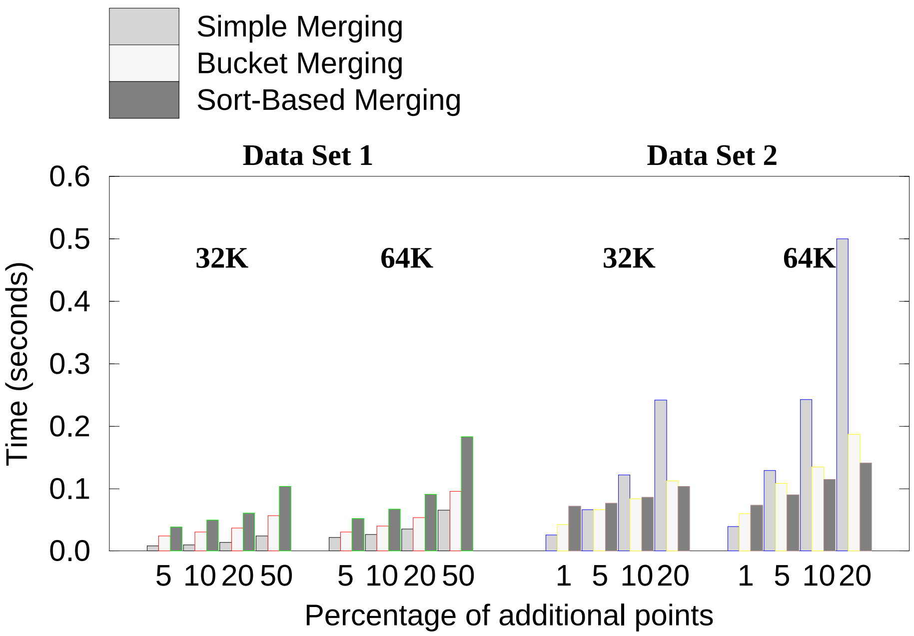


Figure 6: Performance comparison of different merging algorithms on 32-node CM-5 for different fraction of additional elements for Data Set 1 and Data Set 2.

with the corresponding cost of sorting, which is 0.195 seconds.

The situation is totally different for Data Set 2 (Figure 6). 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 is 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.

Figure 7 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 for all 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 three to four times better for Data Set 2.

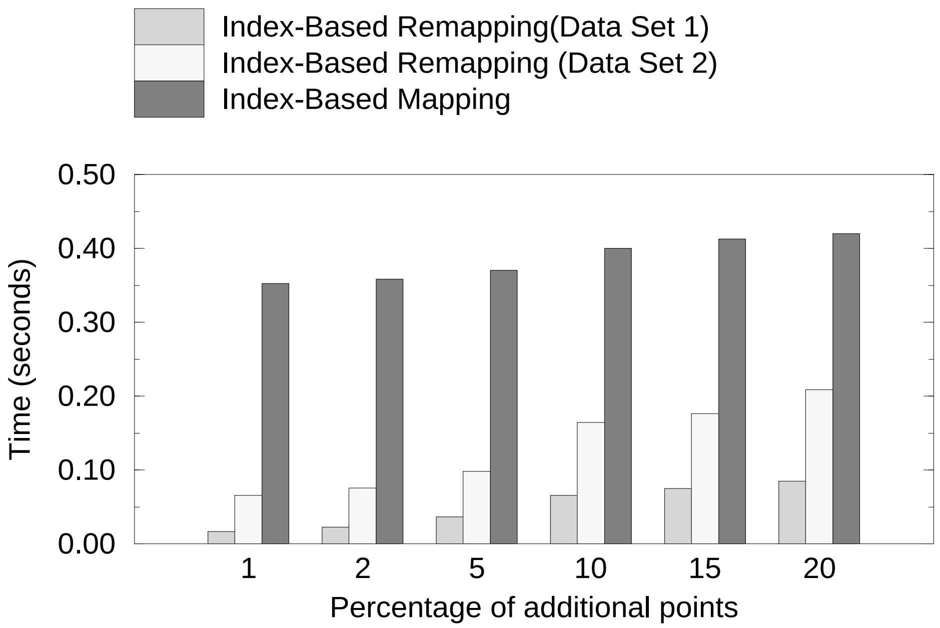


Figure 7: Comparison for Data Set 2 index-based mapping and incremental mapping algorithms on $| V |= 64,000$

3.3 Conclusions

In this chapter we present parallel remapping algorithms for a class of incremental and adaptive data parallel applications. We show that by using the methods developed in this chapter, remapping can be achieved at a fraction of the time required for mapping. Experimental results for these algorithms on the CM-5 support our conclusions.

These methods are proposed to solve the incremental applications for using index-based partitioning method from scratch but not for general-purposed partitioning methods. We believe these methods to be crucial in the parallelization of the class of incremental and adaptive data parallel applications targeted in this chapter.

Chapter 4

Edge-Based Incremental Partitioning

Generally, edge-based partitioning methods have better quality when compared to coordinate-based methods. These methods take the edges into the evaluation of partitioning and maintain the connectivity of the graph. The disadvantages of using edge-based methods are the high cost of partitioning and the difficulty in solving the incremental graph-partitioning problems.

In this chapter we propose methods that use edge information to solve incremental graph-partitioning problems. Our methods assume that a good quality partitioning is available for the initial graph by using one of the graph partitioning techniques (e.g., recursive spectral bisection, simulated annealing, meanfield annealing, etc.).

4.1 Incremental Graph Partitioning

In this section the incremental graph partitioning problem is formulated in terms of linear programming. A high-level overview of the four phases of the incremental graph-partitioning algorithm is shown in Figure 1. Some notation is in order.

Let

1. P be the number of partitions.
2. $\mathcal{B}'(i)$ represent the set of vertices in partition i.
3. μ represent the average load for each partition $\mu = \frac{\sum_i |\mathcal{B}'(i)|}{P}$.

The four steps are described in detail in the following sections.

Step 1: Assign the new vertices to one of the partitions (given by $\mathcal{M}'$).
Step 2: Label each vertex to the closest non-self partition (given by $\mathcal{L}'$).
Step 3: Formulate the linear programming problem based on the mapping of Step 1 and balance loads (i.e., modify $\mathcal{M}'$) minimizing the total number of changes in $\mathcal{M}'$.
Step 4: Refine the mapping in Step 2 to reduce the communication cost.

Figure 1: Four different steps used in incremental graph-partitioning algorithm.

4.1.1 Assigning an Initial Partition to the New Nodes

The first step of the algorithm is to assign an initial partition to the nodes of the new graph (given by $\mathcal{M}'(\mathcal{V})$). A simple method for initializing $\mathcal{M}'(\mathcal{V})$ is given as follows. Let

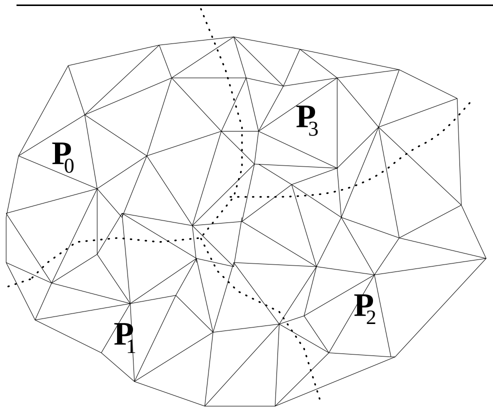
$$\mathcal{M}'(v) = \mathcal{M}(v) \;\; \forall v \in \mathcal{V} - \mathcal{V}_1. \tag{35}$$

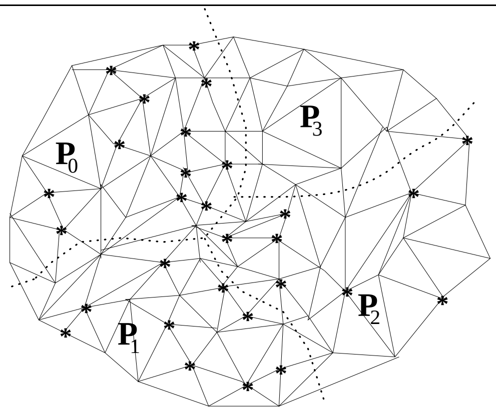
For all the vertices $v \in \mathcal{V}_1$,

$$\mathcal{M}'(v) = \mathcal{M}(x) \;\; \text{where} \;\; \min_{x \in \mathcal{V} - \mathcal{V}_2} (d(v, x)), \tag{36}$$

$d(v, x)$ is the shortest distance in the graph $\mathcal{G}'(\mathcal{V}', \mathcal{E}')$. For the examples considered in this chapter we assume that $\mathcal{G}'$ is connected. If this is not the case, several other strategies can be used.

- If $\mathcal{G}''(\mathcal{V} \cup \mathcal{V}_1, \mathcal{E} \cup \mathcal{E}_1)$ is connected, this graph can be used instead of $\mathcal{G}$ for calculation of $\mathcal{M}'(\mathcal{V})$.
- If $\mathcal{G}''(\mathcal{V} \cup \mathcal{V}_1, \mathcal{E} \cup \mathcal{E}_1)$ is not connected, then the new nodes that are not connected to any of the old nodes can be clustered together (into potentially disjoint clusters) and assigned to the partition that has the least number of vertices.





(a) (b)

Figure 2: (a) Initial Graph (b) Incremental Graph (New vertices are shown by "*").

For the rest of the chapter we will assume that $\mathcal{M}'(v)$ can be calculated using the definition in (7), although the strategies developed in this chapter are, in general, independent of this mapping. Further, for ease of presentation we will assume that the edge and the vertex weights are of unit value. All of our algorithms can be easily modified if this is not the case. Figure 2 (a) describes the mapping of each of the vertices of a graph. Figure 2 (b) describes the mapping of the additional vertices using the above strategy.

4.1.2 Layering Each Partition

The above mapping would ordinarily generate partitions of unequal size. Vertices are moved from one partition to another to achieve load balancing, while keeping the communication cost as low as possible. This is achieved by making sure that the vertices transferred between two partitions are close to the boundary of the two partitions. Each vertex of a given partition is assigned to a different partition it is close to (ties are broken arbitrarily).

$$\mathcal{L}'(v) = \mathcal{M}(x) \tag{37}$$

where x is such that

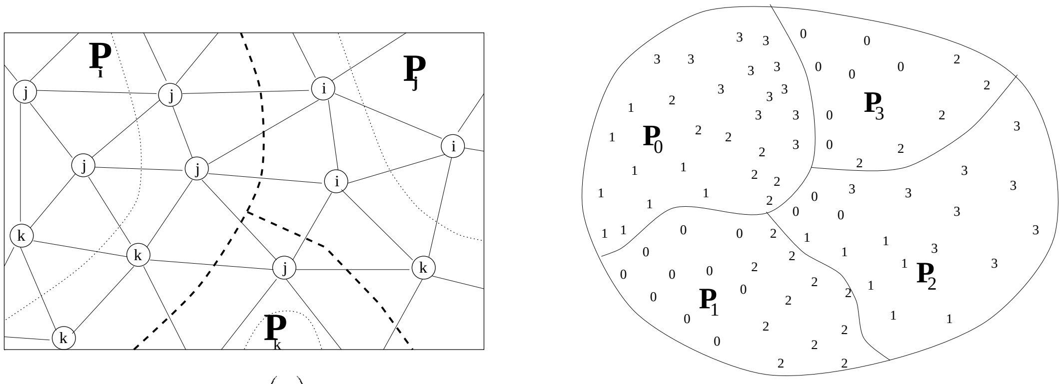
$$\min_{x \notin \mathcal{B}'(\mathcal{M}(v))} (d(v,x)) \tag{38}$$

is satisfied; $d(v,x)$ is the shortest distance in the graph between v and x.

A high-level description algorithm to perform the layering (assuming the graph is connected) is given in Figure 3. This algorithm first finds all the boundary vertices (any vertex which has an edge to a vertex outside its own partition). These vertices are assigned a label equal to the partition which has the maximum number of adjacent vertices (i.e., if a vertex has to be moved to another partition, it should be moved to a partition with a maximum number of adjacent vertices). After this assignment the next layer consists of all the neighbors of these boundary vertices, which are assigned the labels based on labels of the boundary vertices. This process is repeated until all the vertices are labeled. Let α_{ij} represent the number of vertices of partition i that can be moved to partition j for load balancing. Clearly, one would expect that in many cases the number of vertices to be moved from partition i to partition j could be less than α_{ij}. Assuming that only a subset of nodes partition i with label j have to be moved to partition j, a preference would be given to those nodes close to the boundary (this information is saved during the layering process). For the example case of Figure 3, labels of all the vertices are given in Figure 4. A label 2 of vertex in partition 1 corresponds to the fact that this vertex belongs to the set that contributed to α_{12}.

/* Let V_i be the set of vertices belonging to partition i. */
/* Let $adj(v)$ represent the set of all adjacent vertices of vertex v. */
/* Let $map(v)$ represent the partition of vertex v. */
$B := \phi$;
For $v \in V_i$ **do**
    $done(v) :=$ **false**;
    /* $done(v) =$ **false** implies that v has not been labeled so far. */
For $v \in V_i$ **do**
    If $\exists u$, $(u \in adj(v)) \wedge (map(u) \neq i)$ **then**
        $B := B \cup \{v\}$;
        $label(v) :=$ the partition which has maximum number of adjacent vertices to v;
            /* ties are broken arbitrarily */
        $done(v) :=$ **true**;
Repeat
    $D := \phi$;
    For $v \in B$ **do**
        If $\exists u$, $(u \in adj(v)) \wedge$ (**not** $done(u)$) **then**
            $D := D \cup \{u\}$;
    For $v \in D$ **do**
        $label(v) :=$ the label which repeats the most number of times among all the
                adjacent nodes of $v \in B$;
                /* ties are broken arbitrarily */
    $B := D$;
Until $B = \phi$;
$\alpha_{ij} :=$ number of vertices belonging to partition i with label j;

Figure 3: Layering Algorithm for Partitioning



(a) (b)

Figure 4: Labeling the vertices of a graph to the closest outside partition; (a) a microscopic view of the layering for a graph near the boundary of three partitions; (b) label of each vertex of the graph in Figure 2 (b); no edges are shown.

4.1.3 Load Balancing

Let l_{ij} represent the number of vertices to be moved from partition i to partition j to achieve load balance. There are several ways to achieve load balancing, but since one of our goals is to minimize communication cost we would like to minimize $\sum_i \sum_j l_{ij}$, because this would correspond to a minimization of the amount of vertex movement (or "deformity") in the original partitions. Thus the load-balancing step can be formally defined as the following linear programming problem.

$$\text{Minimize} \quad : \quad \sum_{0 \leq i \neq j \leq P} l_{ij} \tag{39}$$

$$\text{Subject to} \quad : \quad 0 \leq l_{ij} \leq \alpha_{ij} \leq |\mathcal{B}'(i)| \tag{40}$$

$$\sum_{0 \leq i < P} (l_{ij} - l_{ji}) = |\mathcal{B}'(j)| - \mu \qquad 0 \leq j < P \tag{41}$$

Constraint 41 corresponds to the load balance condition.

The above formulation is based on the assumption that changes to the original graph are small and the initial partitioning is well balanced, hence moving the boundaries by a small amount will give balanced partitioning with low communication cost.

Constraints in (40):

$$l_{01} \leq 9,\ l_{02} \leq 7,\ l_{03} \leq 12,\ l_{10} \leq 10,\ l_{12} \leq 11,$$
$$l_{20} \leq 3,\ l_{21} \leq 7,\ l_{23} \leq 9,\ l_{30} \leq 7,\ l_{32} \leq 5$$

Constraints in (41):

$$\begin{gathered} l_{01} + l_{02} + l_{03} - l_{10} - l_{20} - l_{30} = 8 \\ l_{10} + l_{12} - l_{01} - l_{21} = 1 \\ -l_{20} - l_{21} - l_{23} + l_{02} + l_{12} + l_{32} = 1 \\ -l_{30} - l_{32} + l_{03} + l_{23} = 8 \end{gathered}$$

Solution using the Simplex Method

$$l_{03} = 8,\ l_{12} = 1$$

all other values are zero.

Figure 5: Linear programming formulation and its solution, based on the mapping of the graph in Figure 2; (b) using the labeling information in Figure 4 (b).

There are several approaches to solving the above linear programming problem. The simplex method is chosen, because it has been shown to work well in practice and because it can be easily parallelized.[1] The simplex formulation of the example in Figure 2 is given in Figure 5. The corresponding solution is $l_{03} = 8$ and $l_{12} = 1$. The new partitioning is given in Figure 6.

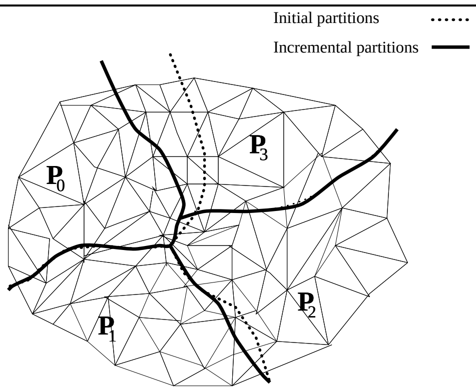


Figure 6: The new partition of the graph in Figure 2 (b) after the **Load Balancing** step.

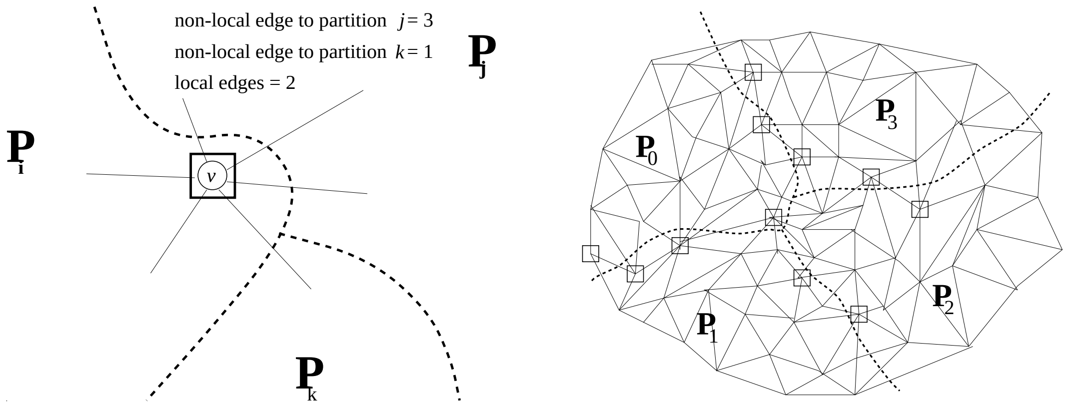
The above set of constraints may not have a feasible solution. One approach is to relax the constraint in (40) and not have $l_{ij} \leq \alpha_{ij}$ as a constraint. Clearly, this would achieve load balance but may lead to major modifications in the mapping. Another approach is to replace the constraint in (41) by

$$\sum_{0 \leq i < P} (l_{ij} - l_{ji}) = \frac{|\mathcal{B}'(j)| - \mu}{\Delta} \quad 0 \leq j < P. \tag{42}$$

Assuming $C > \Delta > 1$, this would not achieve load balancing in one step, but several such steps can be applied to do so. If a feasible solution cannot be found with a reasonable value of Δ (within an upper bound C), it would be better to start partitioning from scratch or solve the problem by adding only a fraction of the nodes at a given time, i.e., solve the problem in multiple stages. Typically, such cases arise

[1] A dense version of simplex algorithm is used in the implementation. The total time can potentially be reduced by using sparse representation.

when all the new nodes correspond to a few partitions and the amount of incremental change is greater than the size of one partition.



(a) (b)

Figure 7: Choosing vertices for refinement. (a) Microscopic view of a vertex which can be moved from partition P_i to P_j, reducing the number of cross edges; (b) the set of vertices with the above property in the partition of Figure 6.

4.1.4 Refinement of Partitions

The formulation in the previous section achieves load balance but does not try explicitly to reduce the number of cross edges. The minimization term in (39) and the constraint in (40) indirectly keep the cross edges to a minimum under the assumption that the initial partition is good. In this section we describe a linear programming-based strategy to reduce the number of cross edges, while still maintaining the load balance. This is achieved by finding all the vertices of partitions i on the boundary of partition i and j such that the cost of edges to the vertices in j are larger than the cost of edges to local vertices (Figure 7), i.e., the total cost of cross edges will decrease by moving the vertex from partition i to j, which will affect the load balance.

In the following a linear programming formulation is given that moves the vertices while keeping the load balance.

Let $M''(k) : V' \longrightarrow P$ represent the mapping of each vertex after the load-balancing step. Let $out(k, j)$ represent the number of edges of vertex k in partition $M''(k)$ connected to partition $j(j \neq M''(k))$, and let $in(k)$ represent the number of vertices a vertex k is connected to in partition $M''(k)$. Let b_{ij} represent the number of vertices in partition i which have more outgoing edges to partition j than local edges.

$$b_{ij} = |\{V \in B_i'' | out(V, j) - in(V) \geq 0\}|.$$

We would like to maximize the number of vertices moved so that moving a vertex will not increase the cost of cross edges. The inequality in the above definition can be changed to a strict inequality. We leave the equality, however, since by including such vertices the number of points that can be moved can be larger (because these vertices can be moved to satisfy load balance constraints without affecting the number of cross edges).

The refinement problem can now be posed as the following linear programming problem:

$$\text{Maximize} \quad : \quad \sum_{0 \leq i \neq j \subset P} l_{ij} \tag{43}$$

$$\text{Subject to} \quad : \quad 0 \leq l_{ij} \leq b_{ij} \qquad 0 \leq i \neq j < P \tag{44}$$

$$\sum_{0 \leq i < j} (l_{ij} - l_{ji}) = 0 \qquad 0 \leq j < P. \tag{45}$$

This refinement step can be applied iteratively until the effective gain by the movement of vertices is small. After a few steps, the inequalities ($l_{ij} \leq b_{ij}$) need to be replaced by strict inequalities ($l_{ij} < b_{ij}$); otherwise, vertices having an equal number of local and nonlocal vertices may move between boundaries without reducing the total cost. The simplex formulation of the example in Figure 6 is given in Figure 8, and the new partitioning after refinement is given in Figure 9.

Constraint (44)

$$l_{01} \leq 1,\ l_{02} \leq 1,\ l_{03} \leq 1,\ l_{10} \leq 2,\ l_{12} \leq 1,$$
$$l_{20} \leq 0,\ l_{21} \leq 1,\ l_{23} \leq 1,\ l_{30} \leq 2,\ l_{32} \leq 1$$

Load Balancing Constraint (45)

$$l_{01} + l_{02} + l_{03} - l_{10} - l_{20} - l_{30} = 0$$
$$l_{10} + l_{12} - l_{01} - l_{21} = 0$$
$$-l_{20} - l_{21} - l_{23} + l_{02} + l_{12} + l_{32} = 0$$
$$-l_{30} - l_{32} + l_{03} + l_{23} = 0$$

Solution using Simplex Method

$$l_{01} = 0,\ l_{02} = 1,\ l_{03} = 1,\ l_{10} = 1,\ l_{12} = 1$$
$$l_{20} = 0,\ l_{21} = 1,\ l_{23} = 1,\ l_{30} = 1,\ l_{32} = 1$$

Figure 8: Formulation of the refinement step using linear programming and its solution.

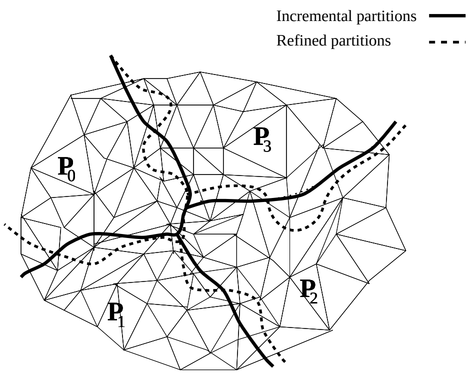


Figure 9: The new partition of the graph in Figure 6 after the **Refinement** step.

4.1.5 Time Complexity

Let the number of vertices and the number of edges in a graph be given by $|\mathcal{V}|$ and $|\mathcal{E}|$, respectively. The time for layering is $O(|\mathcal{V}| + |\mathcal{E}|)$. Let the number of partitions be P and the number of edges in the partition graph[2] be R. The number of constraints and variables generated for linear programming are $O(P + R)$ and $O(2R)$, respectively.

Thus the time required for one iteration of the linear programming is $O((P+R)R)$. Assuming R is $O(P)$, this reduces to $O(P^2)$. The number of iterations required for linear programming is problem-dependent. We will use $f(P)$ to denote the number of iterations required for solving the linear programming problem. Thus the time required for the linear programming is $O(P^2 f(P))$. This gives the total time for repartitioning as $O(|\mathcal{E}| + P^2 f(P))$.

The parallel time is considerably more difficult to analyze. We will analyze the complexity of neglecting the setup overhead of coarse-grained machines. The parallel time complexity of the layering step depends on the maximum number of edges assigned to any processor. This could be approximated by $O(|\mathcal{E}|/P)$ for each level, assuming the changes to the graph are incremental and that the graph is much larger than the number of processors. The parallelization of the linear programming requires a broadcast of length proportional to $O(P)$. Assuming that a broadcast of size

[2] Each node of this graph represents a partition. An edge in the super graph is present whenever there are any cross edges from a node of one partition to a node of another partition.

P requires $b(P)$ amount of time on a parallel machine with P processors, the time complexity can be approximated by $O(\frac{|\mathcal{E}|}{P} + f(P)(P + b(P)))$.

4.2 A multilevel approach

For small graphs a large fraction of the total time spent in the algorithm described in the previous section will be on the linear programming formulation and its solution. Since the time required for one iteration of the linear programming formulation is proportional to the square of the number of partitions, it can be substantially reduced by using a multilevel approach. Consider the partitioning of an incremental graph for 16 partitions. This can be completed in two stages: partitioning the graph into 4 super partitions and partitioning each of the 4 super partitions into 4 partitions each. Clearly, more than two stages can be used. The advantage of this algorithm is that the time required for applying linear programming to each stage will be much less than the time required for linear programming using only one stage. This is due to a substantial reduction in the number of variables as well as in the constraints, which are directly dependent on the number of of partitions. However, the mapping initialization and the layering needs to be performed from scratch for each level. Thus the decrease in cost of linear programming leads to a potential increase in the time spent in layering.

The multilevel algorithm requires combining the partitions of the original graph into super partitions. For our implementations, recursive spectral bisection was used as an *ab initio* partitioning algorithm. Due to its recursive property it creates a natural hierarchy of partitions. Figure 10 shows a two-level hierarchy of partitions. After the linear programming-based algorithm has been applied to repartition a graph that has been adapted several times, it is possible that some of the partitions corresponding to a lower level subtree have a small number of boundary edges between them. Since the multilevel approach results in repartitioning with a small number of partitions at the lower levels, the linear programming formulations may produce infeasible solutions at the lower levels. This problem can be partially addressed by reconfiguring the

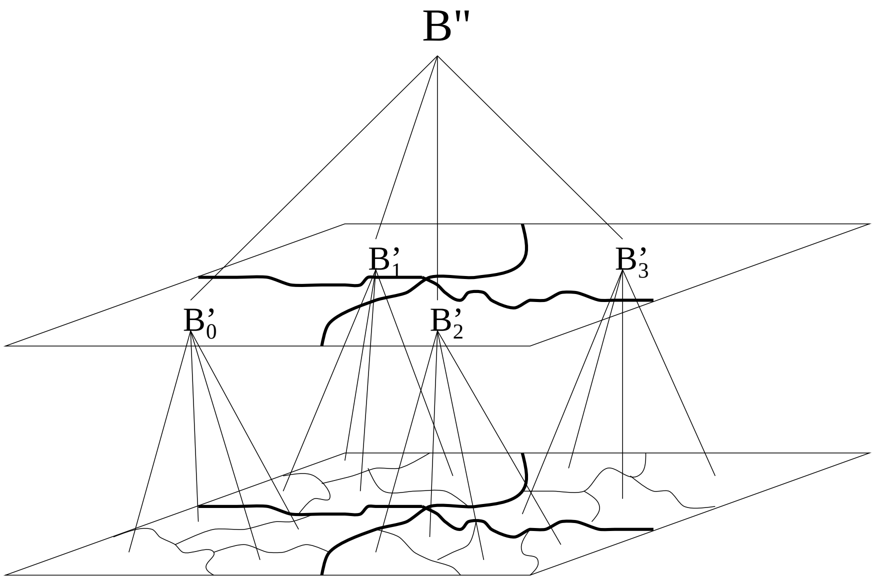


Figure 10: A two-level hierarchy of 16 partitions

partitioning hierarchy.

A simple algorithm can be used to achieve reconfiguration. It attempts to group proximate partitions to form a multilevel hierarchy. At each level it tries to combine two partitions into one larger partition. Thus the number of partitions is reduced by a factor of two at every level by using a procedure FIND_UNIQUE_NEIGHBOR(P) (Figure 11), which finds a unique neighbor for each partition such that the number of cross edges between them is as large as possible. This is achieved by applying a simple heuristic (Figure 12) that uses a list of all the partitions in a random order (each processor has a different order). If more than one processor is successful in generating a feasible solution, ties are broken based on the weight and the processor number. The result of the merging is broadcast to all the processors. In case none of the processors are successful, another heuristic (Figure 12) is applied that tries to modify the partial assignments made by heuristic 1 to find a neighbor for each partition. If none of the processors are able to find a feasible solution, each processor starts with another random solution and the above step is iterated a constant number

```
FIND_UNIQUE_NEIGHBOR(P)
/* A_0, A_1, A_2, ··· A_{P-1} represent the P partitions */
/* Edge_ij represents the number of edges from partition i to partition j. */
global_success := false;    trial := 0;
While (not global_success) and (trial < T) do
   For each processor i do
      Mark[0..P − 1] := -1;
      Random_list[0..P − 1] := list of all partitions in a random order;
      Weight := 0;
      FIND_PAIR(success, Mark, Weight, Edge);
      global_success := GLOBAL_OR(success);
      if (not global_success) then
         FIX_PAIR(success, Mark, Weight, Edge);
         global_success := GLOBAL_OR(success);
      if (global_success) then
         winner := FIND_WINNER(success, Weight);
         /* Return the processor number of maximum Weight */
         BROADCAST(winner, Mark);
         /* Processor winner broadcast Mark to all the processors */
         return(global_success);
      else
         trial := trial+1;
```

Figure 11: Reconstruction Algorithm

FIND_PAIR($success, Mark, Weight, Edge$)
$success$:= **true**;
for $j \longleftarrow 0$ **to** $P - 1$ **do**
    if ($Mark[j] < 0$) **then**
        Find a neighbor k of j where ($Mark[k] < 0$);
        if k exists **then**
            $Mark[k] := j$; $Mark[j] := k$;
            $Weight := Weight + Edge(i, j)$;
        else
            $success$:= **false**;

FIX_PAIR($success, Mark, Weight, Edge$)
$success$:= **true**; $j = 0$;
While ($j < P$) **and** ($success$) **do**
    if ($Mark[j] < 0$) **then**
        if a x exists such that ($Mark[x] < 0$) $\wedge$ (x is a neighbor of l) $\wedge$ ($Mark[l] = k$)
        $\wedge$ (k is a neighbor of j)
            $Mark[x] := l$; $Mark[l] := x$;
            $Mark[j] := k$; $Mark[k] := j$;
            $Weight := Weight + Edge(j, k) + Edge(l, x)$ - $Edge(k, l)$;
            $j := j + 1$;
        else
            $success$:= **false**;
    else
        $j := j + 1$;

Figure 12: A high-level description of the procedures used in the reconstruction algorithm.

(L) times.[3] Figure 11 shows the partition reconfiguration for a simple example.

If the reconfiguration algorithm fails, the multilevel algorithm can be applied with a lower number of levels (or only one level).

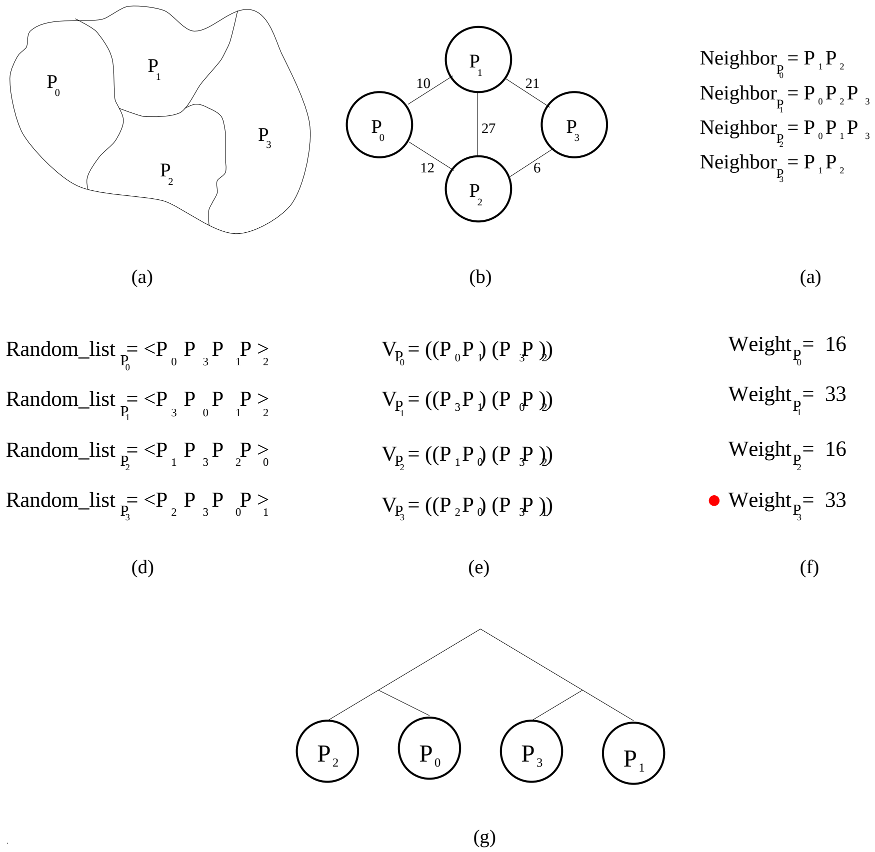


Figure 13: A working example of the reconstruction algorithm. (a) Graph with 4 partitions; (b) partition graph; (c) adjacency lists; (d) random order lists; (e) partition rearrangement; (f) processor 3 broadcasts the result $((P_1P_0)(P_3P_1))$ to the other processors; (g) hierarchy after reconfiguration.

[3]In practice, we found that the algorithm never requires more than one iteration.

4.2.1 Time complexity

In the following we provide an analysis assuming that reconfiguration is not required. The complexity of reconfiguration will be discussed later. For the multilevel approach we assume that at each level the number of partitions done is equal and given by k, thus the number of levels generated is $log_k P$. The time required for layering increases to $O(|\mathcal{E}|log_k P)$. The number of linear programming formulations can be given by $O(\frac{P}{k})$, thus the total time for linear programming can be given by $O(\frac{P}{k}k^2 f(k))$. The total time required for repartitioning is given by $O(|\mathcal{E}|log_k P + Pkf(k))$. An appropriate value of k would minimize the sum of the cost of layering and the cost of the linear programming formulation. The choice of k also depends on the quality of partitioning achieved; increasing the number of layers would usually have a deteriorating effect on the quality of partitioning. Thus values of k have to be chosen based on the above tradeoffs. However, the analysis suggests that for reasonably sized graphs the layering time would dominate the total time. Since the layering time is bounded by $O(|\mathcal{E}|logP)$, this time is considerably lower than applying spectral bisection-based methods from scratch.

Parallel time is considerably more difficult to analyze. The parallel time complexity of the layering step depends on the maximum number of edges any processor has for each level. This can be approximated by $O(\frac{|\mathcal{E}|}{P})$ for each level, assuming the changes to the graph are incremental and that the graph is much larger than the number of processors. As discussed earlier, the parallelization of linear programming requires a broadcast of length proportional to $O(k)$. For small values of k, each linear programming formulation has to be executed on only one processor, else the communication will dominate the total time. Thus the parallel time is proportional to $O((\frac{|\mathcal{E}|}{P} + k^2 f(k))log_k P)$.

The above analysis did not take reconfiguration into account. The cost of reconfiguration requires $O(kd^2)$ time in parallel for every iteration, where d is the average number of partitions to which every partition is connected. The total time is $O(kd^2 \log P)$ for the reconfiguration. This time should not dominate the total time required by the linear programming algorithm.

4.3 Experimental results

In this section we present experimental results of the linear programming-based incremental partitioning methods presented in the previous section. We will use the term "incremental graph partitioner" (IGP) to refer to the linear programming-based algorithm. Our software was written in C language with the CMMD communication library and conducted on the 32-node CM-5 available at NPAC at Syracuse University. The experimental time shown is the median of 11 executions of the same problem. The time represents the "busy time" given by the CMMD library calls.

Graphs

We used two sets of adaptive graphs for our experiments. These graphs were generated using the DIME environment [70].

Set A was generated to study the effect of multiple refinements on a given initial graph. The initial graph of Set A is given in Table 1 (a). The other incremental graphs are generated by refining the previous graph in a localized area. The number of nodes in this set are 1071, 1096, 1121, 1152, and 1192, respectively. Graph set A can be found in Appendix B. We will use $A1$ through $A5$ to represent these graphs. The partitioning of the initial graph ($A1$) was determined using the recursive spectral bisection. This was the partition used to determine the partition of $A2$ using IGP. The partitioning of $A2$, $A3$, and $A4$ using IGP were used as inputs for partitioning $A3$, $A4$, and $A5$, respectively.

Set B was generated to study the effect of the addition of different number of new vertices to the original graph. The initial graph of Set B is given in Table 2 (a). The other graphs in Set B are generated by making refinements in localized areas of the initial graph. The numbers of nodes in the five graphs are given by 10166, 10214, 10305, 10395, and 10838, respectively. Graph set B can be found in Appendix B. We will use $B1$ through $B5$ to represent these graphs. The partitions of the initial graph ($B2$) were determined using recursive spectral bisection. These were the partitions used to partition $B2$, $B3$, $B4$, and $B5$ using IGP.

Graphs	A1	A2	A3	A4	A5
$\mathcal{N}$	1071	1096	1121	1152	1192
$\mathcal{E}$	3185	3260	3335	3428	3548

Table 1: Graphs set A

Graphs	B1	B2	B3	B4	B5
$\mathcal{N}$	10166	10214	10305	10395	10838
$\mathcal{E}$	30471	30615	30888	31158	32487

Table 2: Graphs set B

Results

Direct approach The results of the one-level IGP for Set A graphs on 4-, 8-, 16-, and 32-node CM-5's are presented in Figure 14. The results show that, even after multiple refinements, the quality of partitioning achieved is comparable to that achieved by recursive spectral bisection from scratch; thus this method can be used for repartitioning multiple stages of refinements. The sequential time is around two orders of magnitude faster than Spectral Bisection. The algorithm achieves speedup between 7.01 and 7.75 for 16 nodes and between 15.01 and 17.22 for 32 nodes. Most of the time spent by our algorithm is in the solution of the linear programming formulation using the simplex method. The number of variables and constraints generated by the one-level linear programming algorithm for the load-balancing step

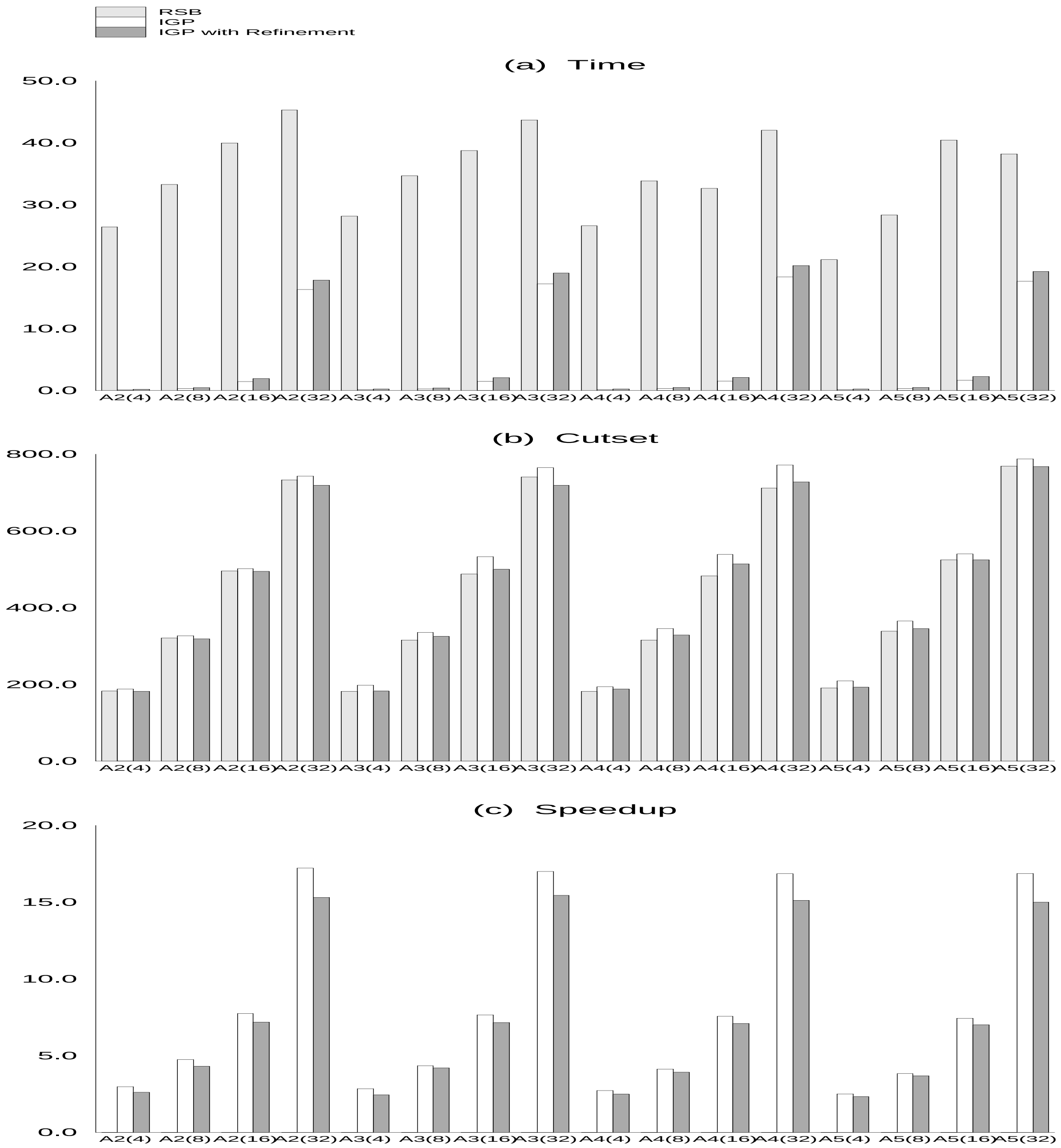


Figure 14: Experimental results of direct approach for Set A on different number of processors. A$x(P)$ refers to results of partitioning graph Ax into P partitions using P processors. (a) Sequential performance of different algorithms. (b) Comparison of Cutsets produced using different algorithms. (c) Speedup of IGP and IGP (with Refinement) for different number of processors.

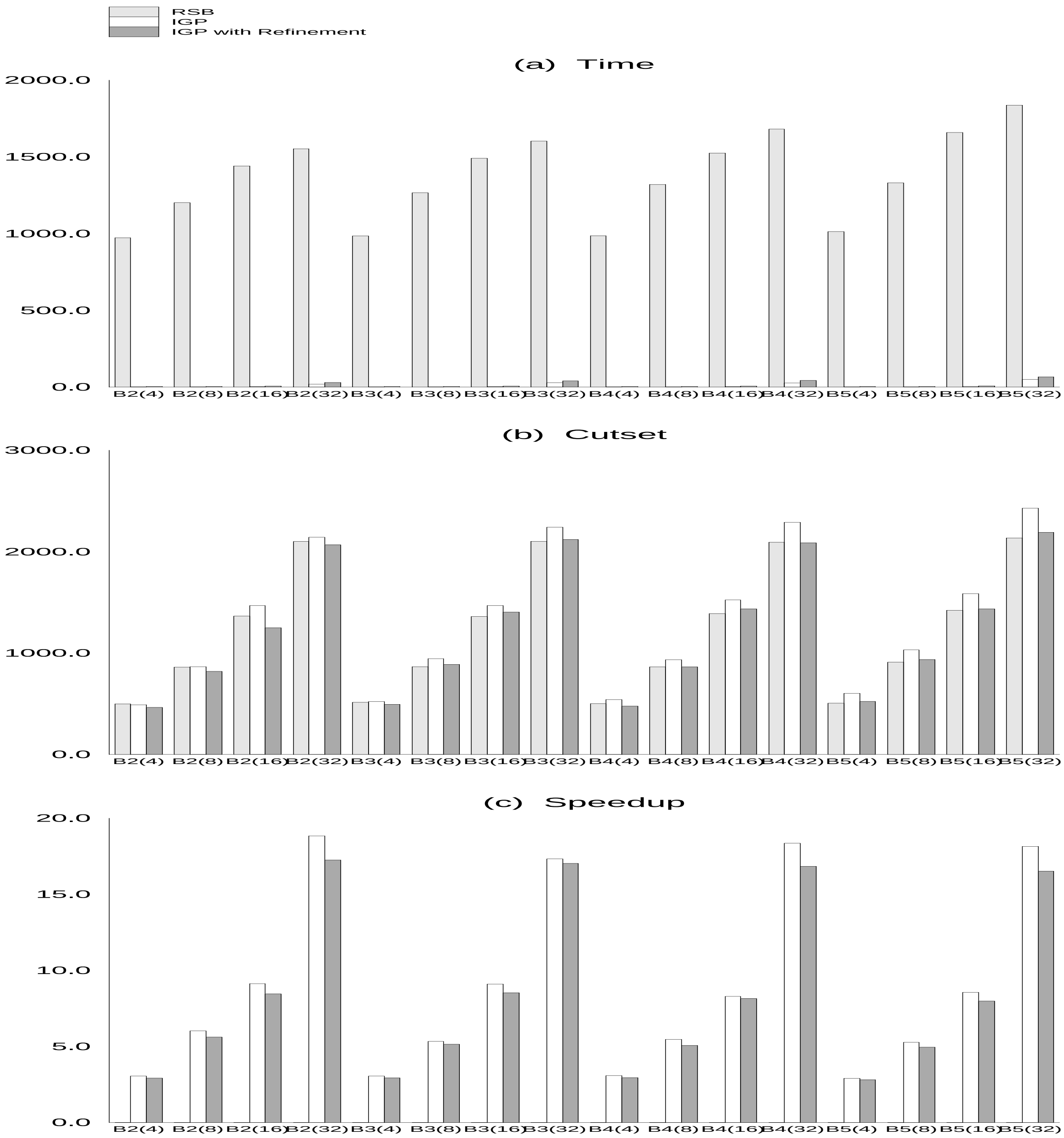


Figure 15: Experimental results of direct approach for Set B on different number of processors. B$x(P)$ refers to results of partitioning graph Bx into P partitions using P processors. (a) Sequential performance of different algorithms. (b) Comparison of Cutsets produced using different algorithms. (c) Speedup of IGP and IGP (with Refinement) for different number of processors.

for graphs in Figure 14 with $|\mathcal{V}| = 1096$ and $|\mathcal{E}| = 3260$ for 32 partitions are 188 and 126, respectively.

The results of the one-level IGP for Set B graphs on 4-, 8-, 16-, and 32-node CM-5's are presented in Figure 15. These results show that, even for substantial additional nodes, the quality of partitioning achieved is comparable to that achieved by recursive spectral bisection from scratch, thus this method can be used for repartitioning even when the amount of additional nodes is large. The sequential implementation is two to three orders of magnitude faster than Spectral Bisection. The algorithm parallelized well. The speedup obtained is between 7.98 and 9.13 for 16 nodes and between 16.53 and 18.84 for 32 nodes. The speedups achieved are generally better than the speedups of Set A. This shows that performance should improve for larger graphs.

The load imbalance ($\frac{maximum\ number\ of\ nodes\ -\ average\ number\ of\ nodes}{average\ number\ of\ nodes}$) before repartitioning for the two sets of graphs is given in Table 3. The number of stages (cf. Equation (13)) required to achieve the load balancing (in all cases the maximum difference between the number of nodes in any two partitions was less than equal to one after repartitioning) are also provided in Table 3.

	4 partitions		8 partitions		16 partitions		32 partitions	
	Imbalance	Stages	Imbalance	Stages	Imbalance	Stages	Imbalance	Stages
$A2$	6.2%	1	14.5%	1	17.6%	1	20.5%	1
$A3$	6.7%	1	14.2%	1	17.1%	1	20.0%	1
$A4$	5.9%	1	11.8%	1	16.6%	1	19.4%	1
$A5$	4.0%	1	11.4%	1	16.2%	1	21.6%	1
$B2$	1.4%	1	2.9%	1	5.8%	1	12.2%	1
$B3$	2.1%	1	5.0%	1	8.7%	1	21.7%	2
$B4$	3.1%	1	4.8%	1	12.2%	1	26.2%	2
$B5$	3.6%	1	6.1%	1	16.8%	1	39.6%	3

Table 3: Then load imbalance before partitioning and the number of stages required to achieve load balance for two sets of graphs

Multilevel approach For the multilevel approach, the linear programming formulation for each subproblem at a given level was solved by assigning a subset of processors. Figure 16 gives the time required for different algorithms and the quality of partitioning achieved for different numbers of levels. The results are presented in Figure 16. The solution qualities of multilevel algorithms show an insignificant deterioration in number of cross edges and a considerable reduction in total time (sequential as well as parallel).

The multilevel approach was also able to achieve perfect load balance in all cases. The different measures for the Set A and Set B using the multilevel approach is given in Figure 16 and Figure 17, respectively. These results show that the multilevel approach was able to keep the quality of partition obtained close to that of applying recursive spectral bisection from scratch. The multilevel approach parallelized well. For the smaller data set, the speedups for 3 levels were between 9.35 and 9.73 for a 32-node CM-5. For the larger data set the speedups for 3 levels were between 18.67 and 23.52. The number of stages required for load balancing at every level of the multilevel algorithm was equal to one.

Tables 4 and 5 show the detailed timing for different steps for the graph $B2$ of the sequential and parallel versions of the repartitioning algorithm, respectively. The time spent for linear programming in a multilevel algorithm is much less than that in a direct algorithm. For the multilevel parallel algorithm, the total time for layering dominates the time. Since the time requirement of layering is $O(levels\frac{\mathcal{E}}{P})$, these results support the complexity analysis in the previous section. The time spent on reconfiguration is negligible compared to the total time.

4.4 Conclusions

In this chapter we have presented novel linear programming-based formulations for solving incremental graph-partitioning problems. The quality of partitioning produced by our methods is close to that achieved by applying the best partitioning methods from scratch. Further, the time needed is a small fraction of the latter and

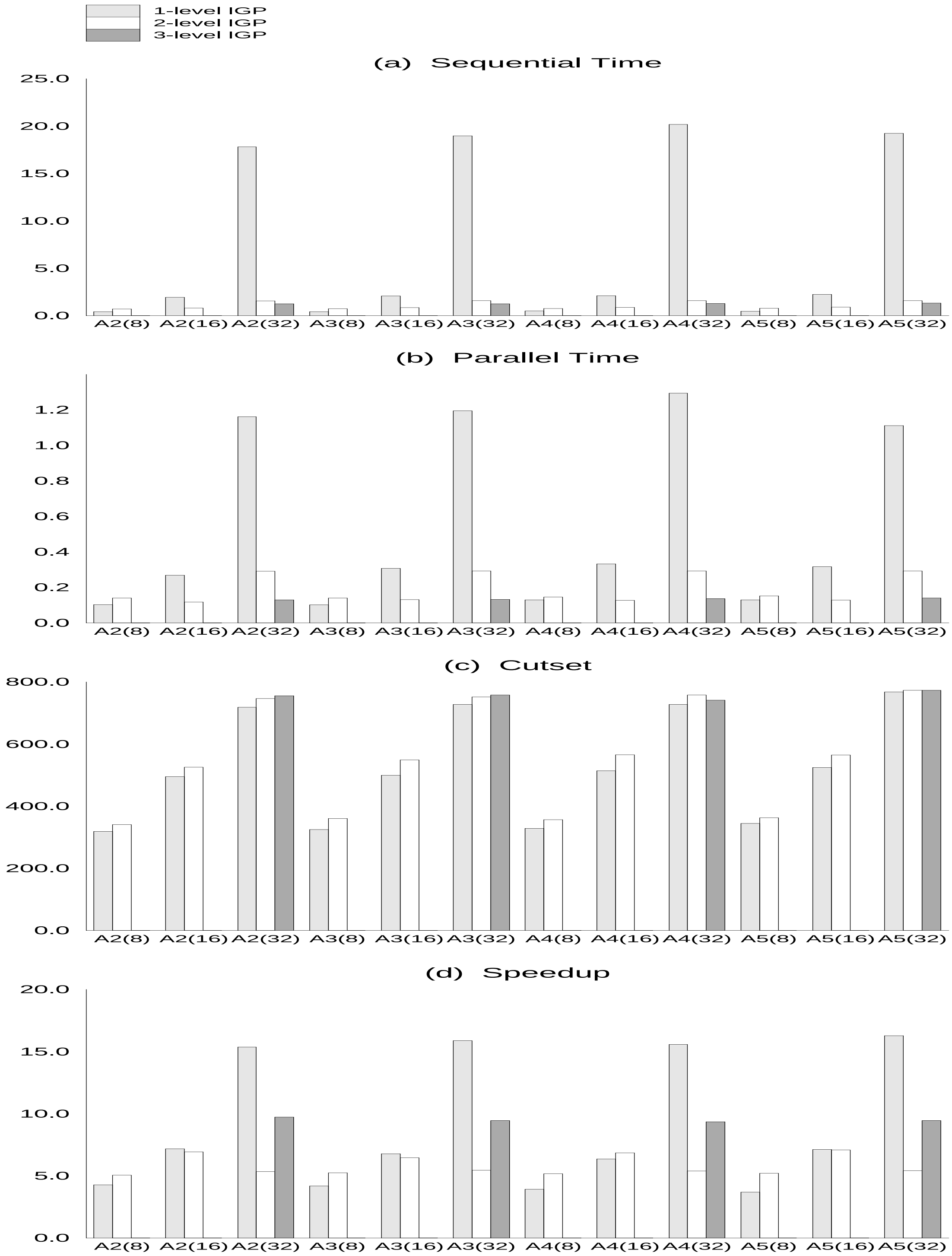


Figure 16: Incremental multilevel IGP and its comparison with direct method (1-level IGP) for Set A. (a) Sequential performance; (b) Parallel performance; (c) Comparison of Cutsets; (d) Speedup for different number of processors.

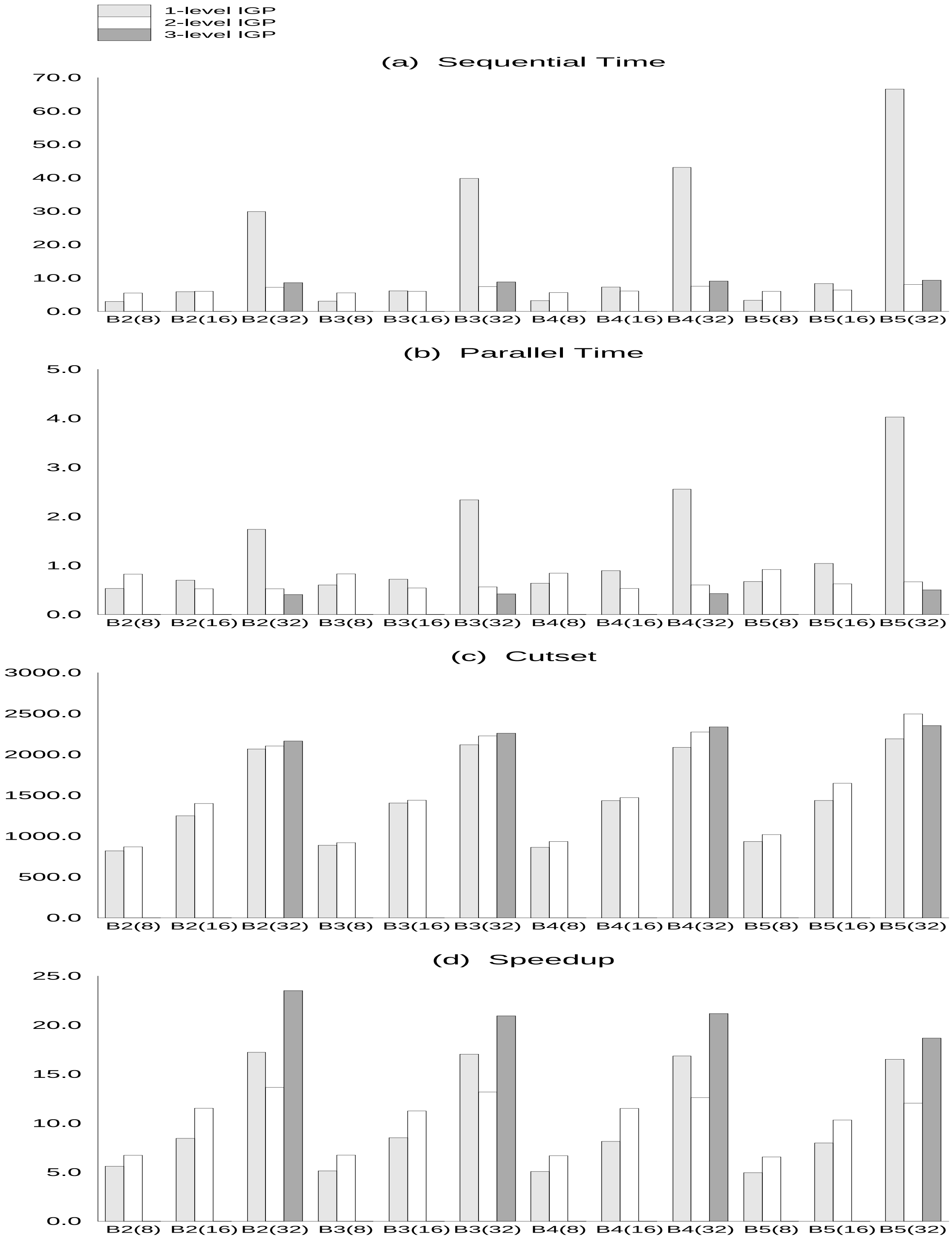


Figure 17: Incremental multilevel IGP and its comparison with direct method (1-level IGP) for Set B. (a) Sequential performance; (b) Parallel performance; (c) Comparison of Cutsets; (d) Speedup for different number of processors.

Processors		4	8		16		32		
Level		1	1	2	1	2	1	2	3
Reconfiguration		—	—	0.004	—	0.008	—	0.008	0.012
Layering	B	2.443	1.982	2.651	1.730	2.990	2.602	3.091	4.190
	R	0.591	0.704	2.802	0.808	2.831	1.620	3.033	4.211
	T	3.041	2.681	5.453	2.542	5.832	4.221	6.121	8.413
Linear programming	B	0.007	0.197	0.015	2.211	0.037	17.221	0.217	0.045
	R	0.012	0.101	0.051	1.173	0.121	8.463	0.874	0.171
	T	0.019	0.299	0.067	3.391	0.158	25.600	1.092	0.216
Total	B	2.450	2.183	2.681	3.950	3.062	19.800	3.301	4.231
	R	0.604	0.808	2.852	1.983	3.001	10.100	3.910	4.382
	T	3.064	2.986	5.559	5.949	6.086	29.907	7.217	8.626

B – Balancing.
R – Refinement.
T – Total.

Table 4: Time (in seconds) required to partition Graph $B2$ ($|\mathcal{V}| = 10214$, $|\mathcal{E}| = 30615$) for different steps in the sequential multilevel IGP.

our algorithms are inherently parallel. This linear-programming based method can be applied to solve incremental applications for any existing graph partitioning method from scratch.

Processors		4	8		16		32		
Level		1	1	2	1	2	1	2	3
Reconfiguration		—–	—–	0.014	—–	0.010	—–	0.013	0.016
Layering	B	0.761	0.279	0.331	0.151	0.217	0.087	0.120	0.162
	R	0.154	0.091	0.343	0.055	0.196	0.033	0.111	0.155
	T	0.915	0.370	0.671	0.206	0.413	0.120	0.232	0.317
Linear programming	B	0.005	0.061	0.005	0.268	0.006	0.954	0.160	0.009
	R	0.010	0.037	0.011	0.191	0.017	0.635	0.049	0.023
	T	0.015	0.099	0.016	0.460	0.024	1.583	0.211	0.033
Data movement	B	0.038	0.021	0.042	0.013	0.026	0.011	0.015	0.023
	R	0.075	0.040	0.083	0.022	0.051	0.015	0.039	0.050
	T	0.114	0.061	0.126	0.036	0.077	0.026	0.054	0.074
Total	B	0.805	0.362	0.381	0.433	0.249	1.051	0.315	0.206
	R	0.240	0.169	0.426	0.269	0.265	0.684	0.200	0.202
	T	1.048	0.531	0.825	0.703	0.528	1.734	0.528	0.409

B – Balancing.
R – Refinement.
T – Total.

Table 5: Time (in seconds) required to partition Graph $B2$ ($|V| = 10214$, $|E| = 30615$) for different steps in the parallel multilevel IGP (on a 32-node CM-5).

Chapter 5

Architecture Dependent Mapping

In the previous chapters, we studied the graph partitioning-problem but the actural mapping onto the underlying architecture. In this chapter we present a fast method to map the partitions on the processor architecture.

A parallel machine consists of a set of processors linked by an interconnection network. Each processor has its own memory that is directly accessible only by this processor. Data exchange and global operations among processors are accomplished through message passing or appropriate hardware support. In the following we discuss several methods to derive this mapping. In the rest of the paper we assume that the number of partitions produced in the previous stage is equal to the number of processors.

5.1 Arbitrary Assignment

One simple way to map partitions onto processors is to assign arbitrarily the partitions to different processors such that each processor gets one partition. This would be appropriate for the following architectures:

- Bus-Based Systems: Clusters of workstations connected by a high bandwidth bus are becoming increasingly popular for high-performance computing. For such networks the partitions can be arbitrarily assigned to any processor. Appropriate modifications need to be made to the clustering step if the computational capabilities of the workstations are unequal.

- Virtual Crossbars: Although theoretical models assume that processors are connected through structured interconnection networks (such as hypercubes and meshes [43]) with new techniques such as wormhole routing and randomized routing (as seen on the CM-5), the distance between communicating processors seems to be less of a determining factor for the amount of time needed to complete the communication. If the paths traversed by two messages have common links, the time required for transmission of messages may cause them to affect each other. The effect of link contention is limited due to the use of virtual channels and the fact that link bandwidths are much larger than node interface bandwidths.

In the above cases the communication cost can be derived based on the measures described in the previous section.

5.2 Minimizing Hop Weight

In this section we discuss methods to perform mapping when the hop weight is a better representative of the communication cost. The interconnection networks of nearly all parallel architectures are hierarchical in nature [43]. A one-dimensional representation of the architecture such that nearby processors are proximate in the representation can easily be derived.[1] In this case, partition i can be assigned to a processor with index i. Examples of such architectures and their mappings are:

- *Fat Tree:* An inorder traversal of the leaf nodes can be used to achieve this mapping.

[1]The clustering in the previous phase ensures that the partitions are ordered, keeping proximity in mind. Hence the communication generated by these computational grids is not random, but to proximate partitions.

- *Linear Array or a Ring:* An ordering can be achieved by starting from the leftmost processor and ending at the rightmost processor. In such cases h_{ij}, the distance between processor i and processor j, can be defined as $|i - j|$.

- *Meshes:* A Hilbert curve ordering can be used to index a multi-dimensional mesh into a one-dimensional array; h_{ij} is given by $|i_x - j_x| + |i_y - j_y|$ where subscripts x and y represent the x-coordinate and y-coordinate of the processor i and j, respectively.

- *Hypercube:* A gray code can be used to index a hypercube; h_{ij} is given by the number of 1's bits in $i \oplus j$ ($\oplus$ represents bitwise exclusive-or).

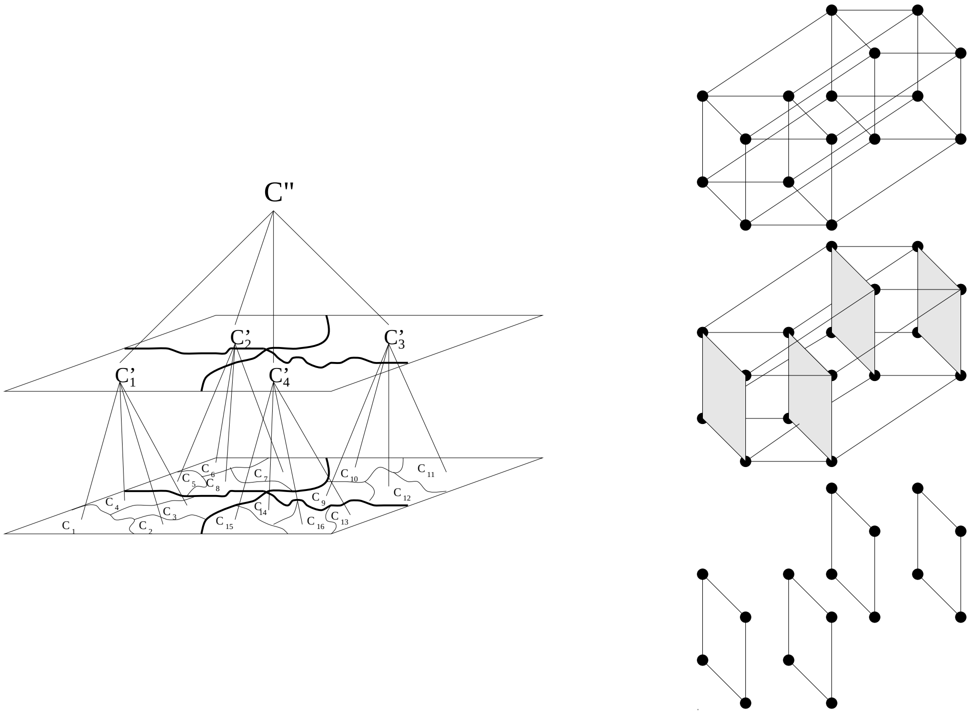


Figure 1: A two-level hierarchy of 16 clusters and associated 4-dimensional hypercube processor architecture

The above methods are simple and should provide mappings with small hop weights relatively quickly. A better, albeit computationally expensive, approach would be to find the mapping of the partitions to the processors that would minimize the cost given in equation 10 or equation 11. Since it is computationally expensive to evaluate all possible permutations, a multilevel approach can be used to significantly reduce the time requirements when the number of processors is large. The clustering in the previous phase ensures that the partitions are ordered with proximity in mind. A natural way of organizing the partitions is to use a multilevel tree with the degree of each node between 4 and 8 (Figure 1). At each level we minimize the cost measure defined in equations 10 or 11 by trying out all the permutations (or a large subset of them). The best one is chosen based on the cost measure (Architecture-Dependent Cost Function (ADCF)).

A high-level description of the algorithm is given in Figure 2. For ease of presentation, the algorithm in Figure 2 is given for a complete q-ary tree and q is a power of 2.[2] The algorithm proceeds level by level. An internal node (including the root) represents a set of q partition-clusters. The mapping of nodes at each level decides on the next $\lg q$ bits, with the most significant bits decided at the root. The mapping of the jth partition-cluster at a given level is decided based on all the previous $q(j-1)$ partition-clusters at the next level (mappings for these have already been obtained) and the remaining $q^{level-1} - j - 1$ partition-clusters at the same level. The ADCF value is minimized based on the subgraph containing these interactions. Figure 3 shows the execution trace of the algorithm for a tree with two levels. Figures 4 through 7 describe the above process for 16-partition trees with the degree of each internal node equal to four. Figure 4 gives the initial mapping of the 16 partitions. The corresponding ADCF value is 605. The first level (root) of the partition tree represents four partition-clusters (each containing four partitions). The initial mapping is given in Figure 5 (b). The corresponding best permutation is given in Figure 5 (c). For the next level, each node represents four partitions. Two out of four subgraphs at this level (corresponding to $j = 1$ and $j = 2$ and shown by dotted lines in Figure 6 (1a) and Figure 6 (2a), respectively) are given in Figure 5 (1b) and Figure 5 (2b), respectively. The corresponding ADCF values are 237 and 116, respectively. Figure 5 (1c)

[2]Our implementation assumes a variable degree and does not restrict the degree to powers of 2.

and Figure 5 (2c) gives the permutation with the best ADCF values. Figure 7 gives the resulting permutation of the whole graph with a considerably lower ADCF value of 470. The total number of subgraphs evaluated is proportional to $O(\frac{P}{q-1})$.

/* Let the number of nodes at each level of the subtree be given by q */

Step 1: $maxlevel = \log_q |P| - 1$.

Step 2: **For** $level \longleftarrow 1$ **to** $maxlevel$ **do**

Step 3: **For** $j \longleftarrow 1$ **to** $q^{level-1}$ **do**

Step 4: Evaluate the ADCF for all permutations of the jth subgraph.

Step 5: Rearrange the of nodes in the jth subgraph according to the permutation with the minimum ADCF value.

Figure 2: Architecture-dependent mapping algorithm

5.3 Experimental Results

In this section we limit our experimental results to the cases where the numbers of partitions are powers of two. The main reason for this restriction is that we wanted to compare the experimental results with the methods in [31], which incorporate the hop weight in the minimization function. The Chaco software that is available to us for achieving this mapping is limited to powers-of-two partitions. For the multilevel algorithm, we chose $q = 4$ and $level = \log_4 |p|$.

We chose the following interconnection networks for evaluation of the hop weight: one-dimensional mesh (linear array), two-dimensional mesh, hypercube, and crossbar. These interconnection networks are in the order of increasing degree and decreasing diameter. A comparison of the average and maximum hop weight using a one-dimensional representation is given in Table 1 for two meshes (Graph 1 and Graph

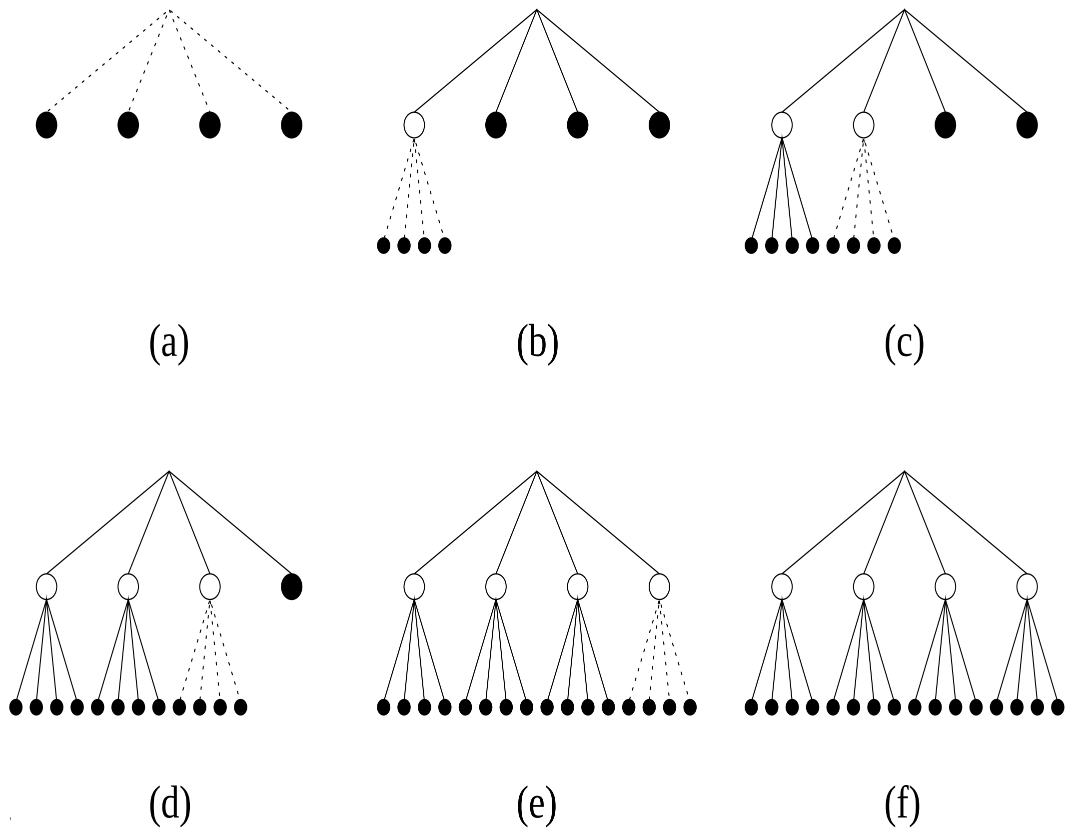


Figure 3: Execution trace of Architecture-Dependent Mapping Algorithm for 2-level tree; dotted lines represent partition-clusters whose mapping is currently being updated; solid nodes represent nodes whose mapping may be used for evaluation of ADCF value for current set of partition-clusters

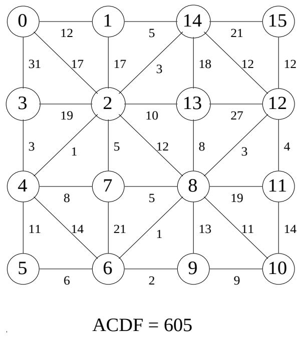


Figure 4: The input partition graph and its ADCF value

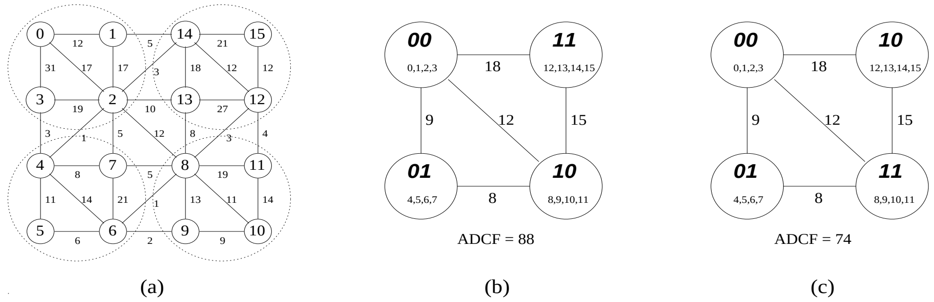


Figure 5: (a) Clusters of four partitions, each shown by dotted circle, (b) The corresponding graph, and (c) rearrangement with minimal ADCF value

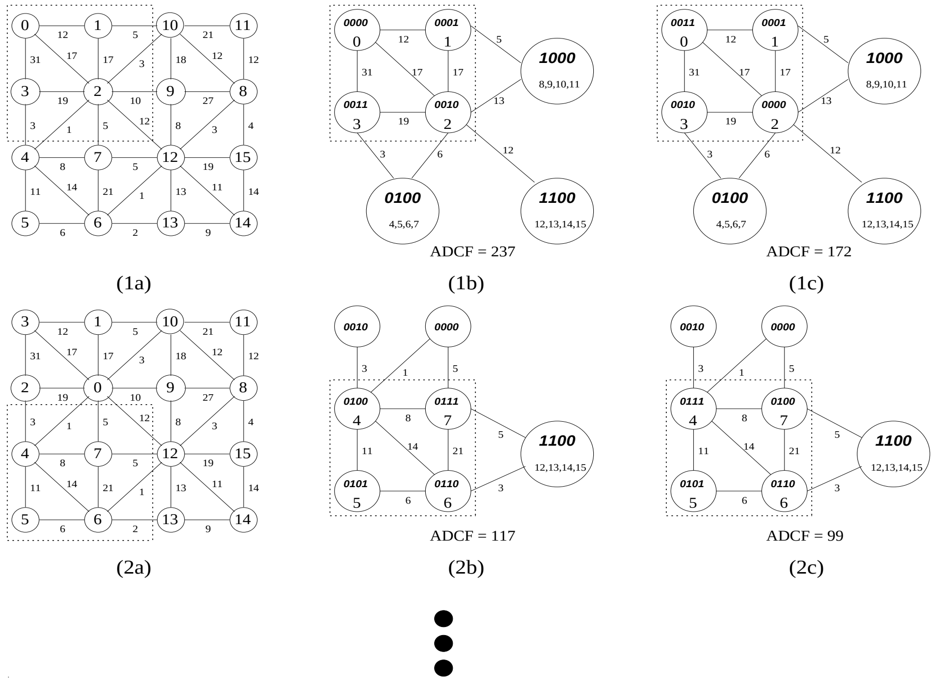


Figure 6: The second level (1a) four partitions at top left corner; (1b) subgraph corresponding to four partitions in (1a); (1c) rearrangement with minimal ACDF value; (2a) to (2c) are corresponding figures for partitions at bottom left corner

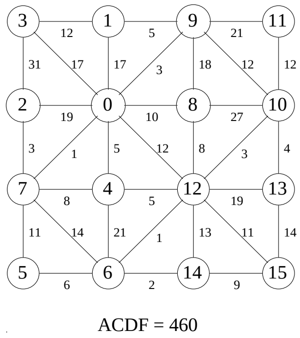


Figure 7: The final permutation and its ADCF value

Graph 1		4		16		64		256	
Architecture	Method	*Ave*	*Max*	*Ave*	*Max*	*Ave*	*Max*	*Ave*	*Max*
Linear	IBP	0.058	0.069	0.276	1.006	1.105	10.05	4.428	62.84
Array	RSB	0.020	0.025	0.100	0.219	0.486	1.800	2.193	14.00
Mesh	IBP	0.047	0.063	0.166	0.345	0.419	1.399	1.011	4.330
	RSB	0.019	0.025	0.067	0.116	0.227	0.640	0.767	2.267
Hypercube	IBP	0.047	0.063	0.115	0.208	0.216	0.382	0.423	0.723
	RSB	0.019	0.25	0.055	0.099	0.147	0.248	0.455	0.744
Crossbar	IBP	0.041	0.054	0.091	0.141	0.174	0.267	0.330	0.515
	RSB	0.016	0.019	0.041	0.064	0.107	0.171	0.261	0.418

Graph 4		4		16		64		256	
Architecture	Method	*Ave*	*Max*	*Ave*	*Max*	*Ave*	*Max*	*Ave*	*Max*
Linear	IBP	0.032	0.045	0.141	0.401	0.566	3.213	2.285	21.07
Array	RSB	0.011	0.014	0.074	0.164	0.353	2.459	1.522	16.69
Mesh	IBP	0.030	0.045	0.097	0.266	0.251	0.766	0.616	2.889
	RSB	0.009	0.010	0.045	0.077	0.159	0.323	0.468	1.144
Hypercube	IBP	0.030	0.045	0.067	0.134	0.135	0.218	0.270	0.436
	RSB	0.009	0.010	0.035	0.061	0.100	0.162	0.265	0.426
Crossbar	IBP	0.027	0.040	0.055	0.093	0.107	0.167	0.211	0.335
	RSB	0.008	0.009	0.025	0.039	0.067	0.104	0.165	0.266

Table 1: A comparison of AverageHopWeight and MaxHopWeight for different architectures using the one-dimensional representation of the architecture.

Graph 1		4		16		64		256	
Architecture	Method	*Ave*	*Max*	*Ave*	*Max*	*Ave*	*Max*	*Ave*	*Max*
Linear	IBP	0.057	0.067	0.245	0.412	0.969	2.932	3.843	25.59
Array	RSB	0.020	0.025	0.091	0.195	0.389	1.364	1.589	12.04
Mesh	IBP	0.048	0.063	0.136	0.208	0.348	0.716	0.810	2.342
	RSB	0.019	0.025	0.056	0.098	0.166	0.482	0.443	1.678
Hypercube	IBP	0.048	0.063	0.118	0.155	0.217	0.323	0.411	0.815
	RSB	0.019	0.025	0.047	0.087	0.122	0.222	0.289	0.486
Crossbar	IBP	0.041	0.054	0.091	0.141	0.174	0.267	0.330	0.515
	RSB	0.016	0.019	0.041	0.064	0.107	0.171	0.261	0.418

Graph 4		4		16		64		256	
Architecture	Method	*Ave*	*Max*	*Ave*	*Max*	*Ave*	*Max*	*Ave*	*Max*
Linear	IBP	0.032	0.045	0.128	0.356	0.491	2.104	1.965	13.93
Array	RSB	0.011	0.014	0.056	0.117	0.238	1.656	0.987	10.87
Meah	IBP	0.031	0.045	0.081	0.184	0.207	0.493	0.497	1.612
	RSB	0.009	0.010	0.034	0.050	0.101	0.235	0.269	0.920
Hypercube	IBP	0.031	0.045	0.066	0.133	0.130	0.230	0.262	0.441
	RSB	0.009	0.010	0.028	0.042	0.073	0.129	0.173	0.269
Crossbar	IBP	0.027	0.040	0.055	0.093	0.107	0.167	0.211	0.335
	RSB	0.008	0.009	0.025	0.039	0.067	0.104	0.163	0.256

Table 2: A comparison of AverageHopWeight and MaxHopWeight for different architectures using the multilevel algorithm.

4). These results show that as the degree of the interconnection network increases (from linear array to a hypercube), the average and maximum hop weight comes close to a crossbar (for which the hop weight is the same as the cut weight). Further, the measures based on IBP are only slightly worse than those based on RSB, except for linear arrays.

The same measures are provided for the mapping by using the multilevel algorithm in Table 2. These results show that the multilevel approach can produce improvements ranging from 10% to 20% for most cases. For MaxHopWeight the improvements can be substantial for IBP, especially for small-degree networks and a large number of partitions. On the other hand, if computational time is at a premium, one-dimensional representation-based mapping can provide a reasonable performance.

Using the above methods we also compared the quality of the hop weight using RSB as described in [31], which directly maps the graph (assuming an underlying hypercube architecture) and refines it using K-L refinement. The parameters of the Chaco software package were set to perform quadrisections.[3] The quality of three different mapping strategies for different sized hypercubes is described in Table 3. The results show that the hop weights for multi-level algorithms are within 5% to 20% for Graph 1, and within a few percent (and sometimes better) for Graph 4. Thus the multilevel algorithm can be used effectively in cases where hop weight is a good measure of communication cost. The time requirements for the multilevel algorithm for different meshes and numbers of partitions are shown in Table 4 and are nominal.

The above results based on RSB have one caveat. Since the results presented are for powers of two, the quality corresponds directly applying recursive bipartitioning to the graph. However, the results do indicate that minimization of the hop weight can be achieved even though the first stage of the partitioning did not explicitly take hop weight into account.

[3] In the final version of the paper we hope to present results based on octasections that can potentially reduce the hop weight reported. We were unable to perform octasections using the current software in a reasonable time.

		Spectral Quadrisection		Linear Representation		Multilevel Algorithm	
Graph	Number of Processors	*Ave*	*Max*	*Ave*	*Max*	*Ave*	*Max*
Graph 1	4	0.015	0.020	0.019	0.025	0.019	0.025
	16	0.045	0.077	0.055	0.099	0.047	0.087
	64	0.112	0.204	0.147	0.248	0.122	0.222
	256	0.252	0.400	0.455	0.744	0.289	0.486
Graph 4	4	0.010	0.011	0.009	0.010	0.009	0.010
	16	0.030	0.051	0.035	0.061	0.028	0.042
	64	0.074	0.144	0.100	0.162	0.073	0.129
	256	0.167	0.273	0.265	0.426	0.173	0.269

Table 3: Comparison of the HopWeight for Different Approaches

	Number of Processors			
Graph	4	16	64	256
Graph 1	0.109	0.179	0.309	0.969
Graph 4	0.269	0.479	0.759	1.569
Time unit in seconds (on SUN4)				

Table 4: Time required for mapping Graph 1 and Graph 4 onto 4-, 16-, 64-, and 256-processor hypercube architecture using multilevel algorithm

5.4 Conclusions

Most methods described in the literature do not explicitly take hop weight into account while performing the partitioning. Once partitioning is achieved by using any of these methods, the multilevel algorithm described in the paper can be used to minimize the hop weight. This can be followed by local refinement techniques to improve the solution.

Chapter 6

SPRINT Software

The methods for partitioning computational graphs for practical application can be broadly classified into two classes based on whether physical coordinate information is used for the partitioning. The use of coordinate-based partitioning methods provides faster but lower quality partitioning than methods that use edge information explicitly.

In this chapter we describe the SPRINT software for partitioning graphs while utilizing only coordinate information on parallel machines.

Related research

In recent years several graph-partitioning libraries have been proposed, including TOP/DOMEDEC [65] by Charbel Farhat and Host Simon, Chaco [34] by Bruce Hendrickson and Robert Leland, MeTiS [40] by George Karypis and Vipin Kumar, and PAFD [54] by Manish Parashar and James Browne.

TOP/DOMDEC is written in C++ and includes a number of partitioning algorithms (a greedy algorithm [19], RCM-based algorithms [47], principal inertia algorithms [20], recursive graph bisection algorithms, and recursive spectral bisection algorithms [59]). The user interface includes high-speed three-dimensional graphics, an interprocessor communication simulator, and an output function with parallel I/O data structures.

Chaco allows for recursive application of any of several different methods for finding small edge separators in weighted graphs. These methods include inertia-based, spectral-based, Kernighan-Lin, and multilevel methods. Chaco not only provides bisection methods, but also multidimensional methods.

MeTiS is a set of programs for partitioning graphs and for producing fill-reducing orderings for sparse matrices. The algorithms in MeTiS are based on multilevel graph partitioning schemes.

PAFD uses the adaptive indexing scheme to solve the Adaptive Finite-Difference Method. The difference between AFD and our approach is that PAFD uses the dynamice indexing method instead of the global indexing method. PAFD assigns a hierachy-structured index to each vertex without the upper limit, but it takes more computational cost on this data structure.

6.1 SPRINT: Scalable Partitioning, Refinement and INcremental partitioning Techniques

The SPRINT library provides software for parallel graph-partitioning using coordinate information such as index-based partitioning, recursive coordinate bisection, and recursive inertial bisection. It is built on the P4 parallel programming system and allows portability across a wide variety of architectures. As MPI (Message-Passing Interface) becomes more popular, we expect to modify our communication routines to MPI.

Figure 1 describes a simple function to generate a random graph. This example is included to show the basic data structures and metadata used in SPRINT to describe graphs: **lower_global_ref** and **upper_global_ref** specify the boundaries of the graph in each processor; **local_n** represents the number of vertices in a processor; **x** represents the double-precision coordinates of vertices in one-dimensional array; **perm** is the global reference for vertices. Currently, SPRINT supports double-precision

```
int     generate_random_graph(n,weight,perm)
int     n, **perm;
double  **x, **weight;
{
int     i,j,lower_global_ref,upper_global_ref,local_n;

lower_global_ref = n*SPRINT_my_proc/SPRINT_num_procs;
upper_global_ref = n*(SPRINT_my_proc+1)/SPRINT_num_procs;
local_n = upper_global_ref-lower_global_ref;

*x = (double *) malloc(local_n*DIM*sizeof(double));
*perm = (int *) malloc(local_n*sizeof(int));
if (weight != NULL) *weight = (double *) malloc(local_n*sizeof(double));

srand48(SPRINT_my_proc);
for (i=0;i<local_n;i++) {
    for (j=0;j<DIM;j++)
        x[i*dim+j] = drand48();
    perm[i] = i+lower_global_ref;
    if (weight != NULL) *weight[i] = NNODES*drand48();
}
return(local_n);
}
```

Figure 1: A simple function to generate a random graph.

coordinates and weights of vertices.

The example in Figure 2 shows a simple code that uses SPRINT software. The code generates a random graph and partitions this graph into the desired number of partitions. The function **p4_initenv** reads the argument list that contains the information needed in P4 (includes process creation). **SPRINT_init** initializes the working space used for SPRINT; **generate_rendom_graph** generates a random graph for partitioning. The partitioning function **SPRINT_coordinate_bisection** partitions the random graph into the desired number of partitions. The space can be deallocated by using **SPRINT_done** and **p4_wait_for_end**. Throughout the chapter examples that illustrate the functionality of SPRINT will use similar data structures.

6.2 Partitioning Functions

6.2.1 Index-Based Partitioners

An index-based partitioning scheme does not need to sort the local elements, quickly finding the m smallest or largest elements instead. The algorithm used in SPRINT is based on quick selection to find the desired elements and is followed by a communication phase to balance the elements.

Syntax

int SPRINT_ibp_hilbert(ndims, local_n, x, xtype, perm)
int SPRINT_ibp_zcurve(ndims, local_n, x, xtype, perm)

int SPRINT_ibp_hilbert_w(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_ibp_zcurve_w(ndims, local_n, x, xtype, wgt, wtype, perm)

int SPRINT_ibp_hilbert_move(ndims, local_n, x, xtype, perm)
int SPRINT_ibp_zcurve_move(ndims, local_n, x, xtype, perm)

```
#include <stdio.h>
#include "P4_HOME_DIR/include/p4.h"
#include "SPRINT_HOME_DIR/include/sprint_defs.h"
#include "SPRINT_HOME_DIR/include/sprint_funcs.h"

#define NNODES 20 /* number of vertices of the graph */
#define DIM 2 /* number of dimensions of the graph */

void    main(argc,argv)
int     argc;
char    **argv;
{
int local_n;
double *x;
int *perm;

p4_initenv(&argc,argv);
SPRINT_init(0,0,1);

local_n = genertae_random_graph(NNODES,&x,NULL,&perm);

SPRINT_coordinate_bisection(DIM,local_n,&x,SPRINT_DBL,&perm);

free(x);
free(perm);
SPRINT_done();
p4_wait_for_end();
}
```

Figure 2: A simple example demonstrating the use of the SPRINT library.

int SPRINT_ibp_hilbert_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_ibp_zcurve_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)

int *ndims*: input the number of dimensions of the graph.
int *local_n*: input the number of vertices in the processor.
void ***x*: input coordinates of vertices in one dimensional row-major representation. It returns the new pointer of coordinates of the local vertices.
int *xtype*: coordinate data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
void ***wgt*: input weights of vertices and return the new pointer of weights of the local vertices.
int *wtype*: weight data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
int ***perm*: input the pointer of the global references and return the new pointer of the global references of the local vertices.

The global-distributed reference table needs to be updated for applications that require the new location of the node. These four sets of functions perform the parallel index-based partitioning using table-look-up for the input graph.; **_hilbert** and **_zcurve** use the Hilbert space-filling curve and z-Curve, respectively. If the input graph is non-uniformly weighted, the suffix with "_w" will take the weights into account to partition the graph. If the attributes of vertices in the graph are coordinates and/or weights, users may move these attributes along with the partitioner by calling the functions with the suffix "_move". These functions allow users to acquire the data that has been located at the proper processors after the completion of the function. All four of these sets of functions require that users assign a globally unique integer reference to each vertex. The index will be used to create the global-distributed reference table. The range of global references is $[0 .. N)$ where N represents the number of vertices of the graph. All these functions return the new number of vertices after partitioning. An example for the correct function call sequence in SPRINT is given in Figure 3.

```
#include        <stdio.h>
#include        "P4_HOME_DIR/include/p4.h"
#include        "SPRINT_HOME_DIR/include/sprint_defs.h"
#include        "SPRINT_HOME_DIR/include/sprint_funcs.h"

#define NNODES 20       /* number of vertices of the graph */
#define DIM 2           /* number of dimensions of the graph */

void    main(argc,argv)
int     argc;
char    **argv;
{
int     local_n, new_local_n;
double  *x, *wgt;
int     *perm;

p4_initenv(&argc,argv);
SPRINT_init(0,0,1);
local_n = genertae_random_graph(NNODES,&x,&wgt,&perm);

new_local_n = SPRINT_ibp_hilbert_move_w(DIM,local_n,&x,SPRINT_DBL,
                    &wgt,SPRINT_DBL,&perm);
free(x);
free(perm);
free(wgt);
SPRINT_done();
p4_wait_for_end();
}
```

Figure 3: An example of using the index-based partitioner based on the Hilbert space-filling curve to redistribute weighted graphs.

6.2.2 Recursive Coordinate Bisection (RCB) and Recursive Inertial Bisection

Syntax

int SPRINT_coordinate_bisection(ndims, local_n, x, xtype, perm)
int SPRINT_inertial_bisection(ndims, local_n, x, xtype, perm)

int SPRINT_coordinate_bisection_w(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_inertial_bisection_w(ndims, local_n, x, xtype, wgt, wtype, perm)

int SPRINT_coordinate_bisection_move(ndims, local_n, x, xtype, perm)
int SPRINT_inertial_bisection_move(ndims, local_n, x, xtype, perm)

int SPRINT_coordinate_bisection_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_inertial_bisection_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)

int *ndims*: input the number of dimensions of the graph.
int *local_n*: input the number of local vertices in the processor.
void ***x*: input the pointer of coordinates of local vertices and return the new pointer of coordinates of local vertices after partitioning.
int *xtype*: coordinate data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
void ***wgt*: input the pointer of weights of local vertices and return the new pointer of weights of local vertices after partitioning.
void ***wgt*: input pointer of weights of vertices.
int *wtype*: weight data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
int ***perm*: input the pointer of global references of local vertices and return the new pointer of the global references of local vertices after partitioning.

These four functions perform recursive coordinate bisection. The functions with the suffix "_w" or "_move" retain the same meanings as those in the previous sections. The total cost may be reduced if one moves the partial attributes of vertices to the proper processor and redistributes the rest of the attributes by using **SPRINT_data_redistribute**, because this method needs coordinates to compute the momentum of the graph and subgraphs. Coordinates have to be moved after each recursive step of coordinate-based bisection, as do the weights.

6.2.3 Incremental Index-Based Partitioning–Perturbation

Syntax

int SPRINT_perturb_ibp_hilbert(ndims, local_n, x, xtype, perm)
int SPRINT_perturb_ibp_zcurve(ndims, local_n, x, xtype, perm)

int SPRINT_perturb_ibp_hilbert_w(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_perturb_ibp_zcurve_w(ndims, local_n, x, xtype, wgt, wtype, perm)

int SPRINT_perturb_ibp_hilbert_move(ndims, local_n, x, xtype, perm)
int SPRINT_perturb_ibp_zcurve_move(ndims, local_n, x, xtype, perm)

int SPRINT_perturb_ibp_hilbert_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)
int SPRINT_perturb_ibp_zcurve_w_move(ndims, local_n, x, xtype, wgt, wtype, perm)

int *ndims*: input the number of dimensions of the graph.
int *local_n*: input the number of vertices in the processor.
void **x: input the pointer of coordinates of the local vertices in one-dimensional row-major representations and return the new pointer of coordinates of the local vertices after partitioning.

int *xtype*: coordinate data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
void ***wgt*: input the pointer of weights of local vertices and return the new pointer of weights of local vertices after partitioning.
int *wtype*: weight data type (e.g., SPRINT_INT, SPRINT_FLT, or SPRINT_DBL).
int ***perm*: input the pointer of the global references and return the new pointer of global references of the local vertices.

Four sets of functions perform the incremental index-based partitioning for the applications that graphs slightly change after each computation iteration. The suffix "_w" and "_move" represent the same meanings in the section of Index-Based Partitioning. Uses of the incremental index-based partitioning are restricted to calling the incremental partitioning functions in SPRINT instead of developing the communication functions themselves, since the the global reference information will not be maintained for the next calling routine. If communication functions to move data are needed, static partitioners may be used instead. Since the cost of an incremental partitioner is much less than the cost of a static partitioner, users may lose this advantage.

6.3 Incremental Index-Based Partitioning – Addition

Syntax

int SPRINT_inc_ibp_hilbert(ndims, local_n, perm, add_local_n, add_x, xtype, add_perm)
int SPRINT_inc_ibp_zcurve(ndims, local_n, perm, add_local_n, add_x, xtype, add_perm)

int SPRINT_inc_ibp_hilbert_w(ndims, local_n, perm, add_local_n, add_x, xtype, add_wgt, wtype, add_perm)

int SPRINT_inc_ibp_zcurve_w(ndims, local_n, perm, add_local_n, add_x, xtype, add_wgt, wtype, add_perm)

int SPRINT_inc_ibp_hilbert_move(ndims, local_n, x, xtype, perm, add_local_n, add_x, add_perm)
int SPRINT_inc_ibp_zcurve_move(ndims, local_n, x, xtype, perm, add_local_n, add_x, add_perm)

int SPRINT_inc_ibp_hilbert_w_move(ndims, local_n, x, xtype, wgt, wtype, perm, add_local_n, add_x, add_wgt, add_perm)
int SPRINT_inc_ibp_zcurve_w_move(ndims, local_n, x, xtype, wgt, wtype, perm, add_local_n, add_x, add_wgt, add_perm)

int *ndims*: input the number of dimensions of the graph.
int *local_n*: input the number of old vertices in the processor.
void ***x*: input the pointer of coordinates of the old local vertices and return the new pointer of coordinates of the old local vertices and the additional vertices.
int *xtype*: coordinate data type.
void ***wgt*: input the pointer of weights of the old local vertices and return the new pointer of weights of the old local vertices and the additional vertices.
int *wtype*: weight data type.
int ***perm*: input the pointer of the global references of the old local vertices and return the new pointer of the global reference indices of the old vertices and new vertices.
int *add_local_n*: input the number of local additional vertices in the processor.
void **add_x*: input the coordinates of additional vertices in one dimensional row-major representations.
void ***add_wgt*: input the weighted of local additional vertices.
int *add_perm*: input the global references of additional vertices.

These four sets of functions perform incremental index-based partitioning for adding new vertices to the graph. Functions with the suffix "_w" or "_move" retain the same meanings mentioned in the previous sections. The same restrictions as in the previous section (Incremental Index-Based Partitioning – Perturbation) are applied to these four sets of functions. The advantages of these functions are that the costs are much less than the costs of the static partitioners, since most of the computations and communications occur in the additional vertices.

6.3.1 Incremental Index-Based Partitioning–Deletion

Syntax

int SPRINT_del_ibp(local_n, perm)
int SPRINT_del_ibp_w(local_n, perm)

int *local_n*: input rest number of local vertices after deletion.
int ***perm*: input the pointer of the global references of the rest vertices and return the new pointer of the global reference indices of the local vertices.

These functions need global references instead of graph data, because the attributes of the vertices remain the same. The only argument the user is asked to input is the global references of the rest of the graph's vertices. SPRINT does not provide the options as did for the previous partitioning functions, since the attributes of vertices are not involved in these functions. The data redistributing function in the "Utilities" section can be used to rearrange the vertices.

6.4 Other Utilities

6.4.1 Initialization

Syntax

void SPRINT_init(debug_flag, demo_flag, log_flag)

int *debug_flag*: turn on the debuging process to track the current SPRINT status if the flag is set to 1; otherwise set the flag to 0. This option will help users to check the status of communications and generate the detail timing inside the set of index-based partitioners.
int *demo_flag*: turn on the demonstrating process to allow users check the current information of partitions if the flag is set to 1; otherwise set the flag to 0.
int *log_flag*: turn on the script option if the flag is set to 1; otherwise set the flag to 0. This will generate a script file for each processor to record the intremedia output for debuging or demonstrating the current partitions.

This function is used to set up the following global variables used in SPRINT:

- *SPRINT_num_procs* is an integer to indicate the total number of processors involved in SPRINT.
- *SPRINT_my_proc* is an integer to indicate the processor identification number of the processor.
- *SPRINT_logic_id* is an integer to indicate the processor identification number of the current processor group, especially for recursive coordinate bisection and recursive inertial bisection.
- *SPRINT_global_max* is a double-precision array of the size of dimensions of the graph to store the global maximum coordinates in each dimension. In other words, this will provide the upper bound for the input graph to all partitioning functions in the SPRINT.

- *SPRINT_global_min* is a double precision array of the size of dimensions of the graph to store the minimum coordinates in each dimension and this corresponds to lower bound for the graph.
- *SPRINT_dim* is an integer of the number of useful dimensions used in "Indexing". Most of the time this value will equal to the dimensions of the graph. The exception occurs when one or more dimensions are too small as compared to the most significant dimension.
- *SPRINT_vtx_boundary* is an integer array to store the boundaries of the number of vertices for all processors.
- *SPRINT_idx_boundary* is an integer array to stored the largest index in each processor for the usage of incremental IBP for additional vertices and perturbation. This will help SPRINT locate the new vertices to the proper processors.
- *SPRINT_ref_boundary* is an integer array to store the boundary entries of the reference table.
- *SPRINT_ibp_old_idx* is a structural array to store the index values for each local vertices acquired from the previous partitioning. This is used in incremental IBP for additional vertices, perturbation, and deleting vertices. Each element of this array contains two attributes, which are the index generated from the index-based partitioner and the global reference.
- *SPRINT_ibp_old_idx_w* is a structural array to store the index values for each local vertices acquired from the previous partitioning. This is used in incremental IBP for additional vertices, perturbation, and deleting vertices. Each element of this array contains three attributes: the index generated from the index-based partitioner, the weight, and the global reference.
- *SPRINT_ref_table* is a structure array containing two attributes, the processor ID and the local address, to indicate the location of the physical data (Figure 4).

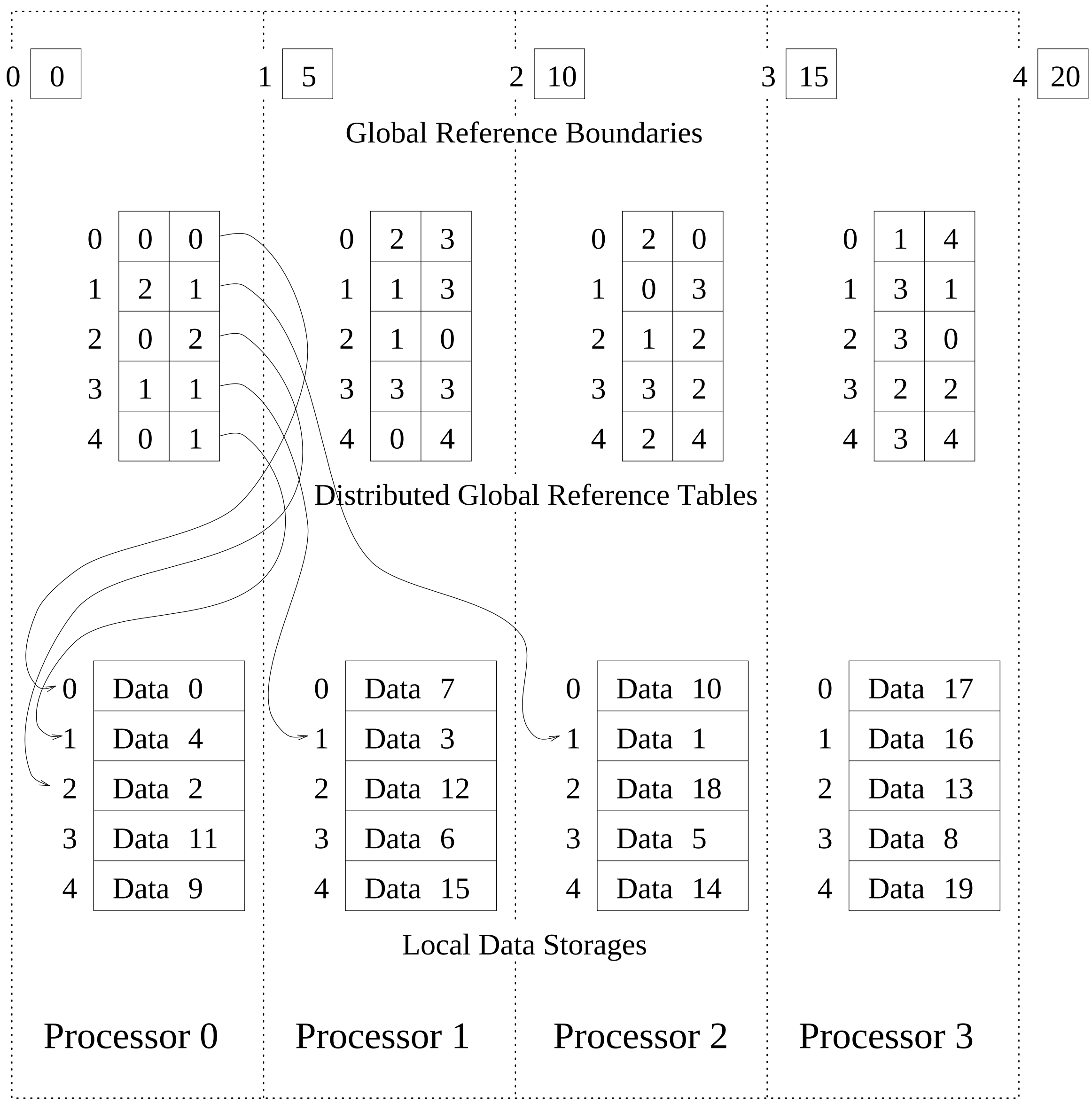


Figure 4: An example of distributed global reference tables on 4 processors: The first row represents the boundaries of the reference table for all processors. The second row shows that the distributed reference tables point to the local data storages where the first column represents the processor ID, and it represents the local address in that processor. The third row represents the local data storages on all processors.

6.4.2 Termination

Syntax

void SPRINT_done()

The following function will free the space of those global variables used in SPRINT.

6.4.3 Data Redistribution

Syntax

int SPRINT_data_redistribute(local_n, data, size)

int *local_n*: input the number of local vertices in the processor. **char** ***data*: input the pointer of local data associated with the global references and return the new pointer of local data after the partitioning. Data can be in any type of array.
int *size*: input the size of an element in the data.

int SPRINT_data_redistribute_v(local_n, data, size)

int *local_n*: input the number of local vertices in the processor. **char** ***data*: input the pointer of local data associated with the global references and return the new pointer of local data after the partitioning. Data can be in any type of array.
int ***size*: input the pointer of the size of each element in the data and returned the new pointer of the size of each element in the redistributed data.

SPRINT_data_redistribute will redistribute the graph to the processors based on the mapping returned from partitioning functions. **SPRINT_data_redistribute_v** will redistribute variable sized graphs. Both functions return the number of local vertices in the processor after partitioning. A better data structure that users may think about before implementing the applications might be a structural type in the C language because users have to use this function to redistribute each attribute of the graph if the data type is an array, or a lot of local data copies if the data type is a pointer.

6.4.4 Communication Schedule

Syntax

void SPRINT_create_schedule(local_n, ref_list, local_ref_list)

int *local_n*: input the number of local vertices in the processor after data redistribution.
int **ref_list*: input an array of global references that are needed to perform the local computations.
int **local_ref_list*: output an array of local references corresponding to global references.

This function will create the communication schedule and rearrange the collection of local and non-local references to the local references. For some applications that need to compute the attributes of neighbor vertices for each vertex in a iteration, users have to specify the reference list (adjacent list) for the input of this function and this function will return a new reference list that points to the data stored locally. This function will remove the duplicate references by using the hash table to reduce the size of messages and the size of local storage. Figure 5 describes the mapping of global references to local references.

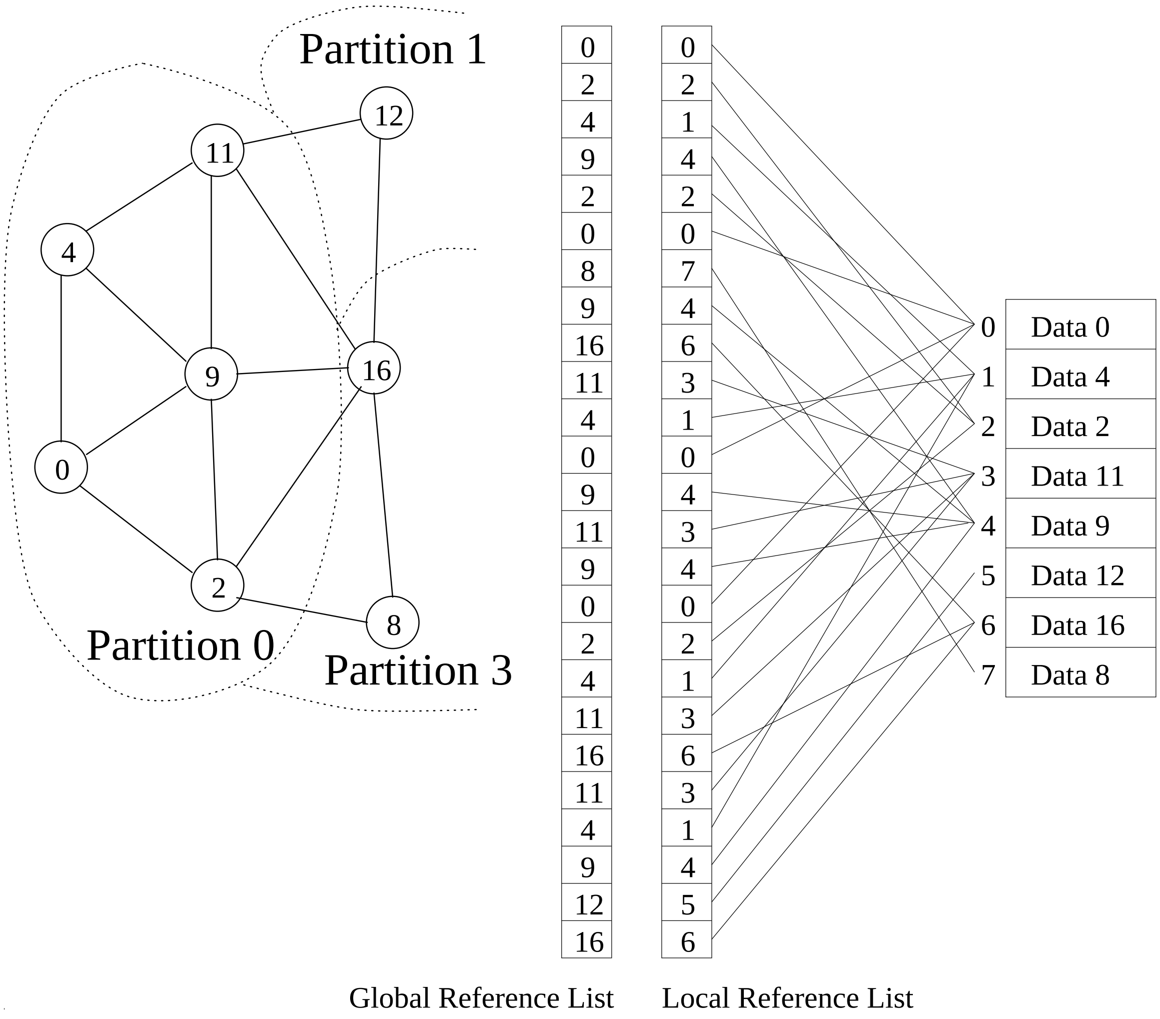


Figure 5: A mapping from the global references to local references for the example graph at the left.

6.4.5 Data Localization

Syntax

void SPRINT_gather(data, outdata, size)

char **data*: input the data of local vertices in character type.
char **outdata*: output the data of local referencing vertices in character type.
int *size*: input the size of the element in the data.

This function provides the communication to gather those data that are non-local to the processor and rearrange the localized data corresponding to the schedule created by **SPRINT_create_schedule**. Users may be asked to pack the data into a one-dimensional array and indicate the size of the element of the data array to this function. After data localization, users can access the localized data by using *local_ref_list*. Figure 6 shows the locations of all data that will be used in the next computation iterations. The data is arranged corresponding to the processor ID of the data in ascending order.

6.5 Conclusions

The performances of the four partitioning schemes included in the SPRINT package are close in quality, but index-based partitioning provides more flexibility for incremental aspects. SPRINT not only provides partitioners, but also a parallel computation environment for developing parallel applications on message-passing-based machines. Users can easily develop the parallel applications on SPRINT. We believe that SPRINT is useful for those who work on parallel computing applications.

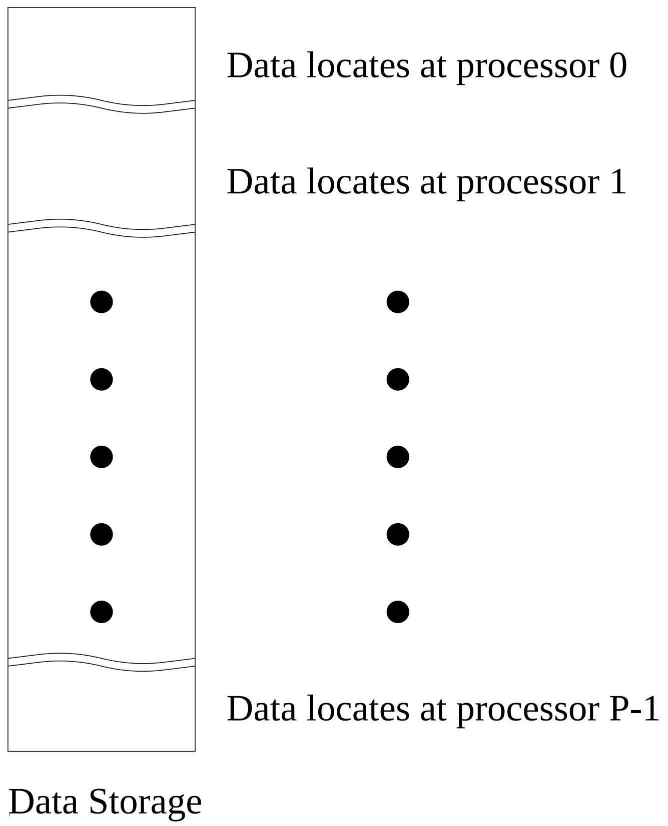


Figure 6: Storage arrangement for local and non-local data after localizing data.

Chapter 7

Conclusion

The main contribution of the dissertation is the development of novel ideas for utilizing existing well-studied methods to solve the graph-partitioning problem in irregular and adaptive applications. Our algorithms are based on the transformation from a high-dimensional computational graph to a linear representation and the usage of the existing partitioning information to perform incremental partitioning. We have shown that a transformed linear representation can be efficiently used to solve the graph-partitioning problem. In our experiments, the eigen-based partitioning methods perform the best-quality results, but their costs are much higher than coordinate-based partitioners. For adaptive irregular applications the coordinate-based methods are better fitted than those eigen-based methods since the costs of the partitioners themselves have to be taken into account. Therefore, a fast and adaptable partitioner can solve the adaptive problems better. We have also shown that our algorithms perform faster than other partitioning methods, and the quality is competitive with other coordinate-based methods but worse than eigen-based methods. Most of our algorithms have been implemented on the CM-5 and PARAGON and some of them on linear clusters (SUN-4, ALPHA, and RS6000). For the purpose of portability the SPRINT software has been implemented on the P4 parallel programming system.

The index-based remapping can be used to solve the adaptive irregular problems without starting from scratch. For the graphs of perturbation, this method uses a fast search to locate the unsorted elements that are not in an ascending order and merges the list of unsorted elements into the rest of the list. For the graphs of addition or deletion the index-based remapping quickly balances the merged list that merges the additional elements into the existed sorted list or rebalances the rest list.

The incremental graph partitioner using the linear programming approach minimizes the movements of elements to reach a balance and maximizes the movements of elements to acquire better-quality partitions. We also developed a multilevel approach to reduce the problem size for solving the linear programming problem in order to gain a better performance without distorting the quality. This method can be widely used when beginning with eigen-based partitioners instead of using eigen-based partitioners throughout the entire execution of the applications. We have shown that the performance of the linear programming approach is much faster than the eigen-based methods and the quality is comparable.

The SPRINT software not only collects several commonly used coordinated-based partitioning methods such as recursive coordinate bisection, recursive inertial bisection, and index-based partitioning, but also incremental index-based partitioning. SPRINT is implemented in the C language on the P4 parallel programming system. The SPRINT software can be used either in part of the runtime of a high performance language (High Performance FORTRAN, High C, ...,etc.) or individually.

7.1 Future Research

Several areas discussed in this dissertation can be extended in various directions.

- The quality of the index-based methods is inferior to using eigen-based methods, but it can be improved by using refinement methods (e.g., the Kernighan-Lin method, Simulated Annealing, etc.).

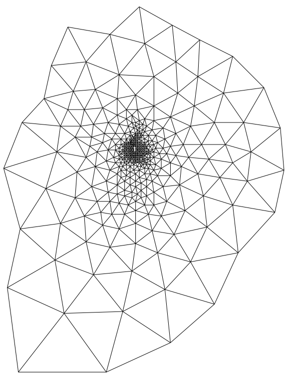
- Hybrid methods can be developed according to the index-based method, since the cost of the index-based method is very low and locality is maintained. This might improve several parallel methods (e.g., MeTiS [40, 41] and Multilevel

Recursive Spectral Bisection [2, 65]), which need to solve the maximum independent set problem on a parallel computing environment.

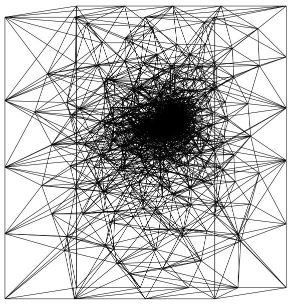
- The SPRINT software can be implemented using Message-Passing Interface (MPI) to fit well into the current parallel computing world.

Appendix A

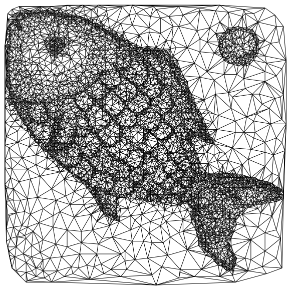
Experimental Graphs



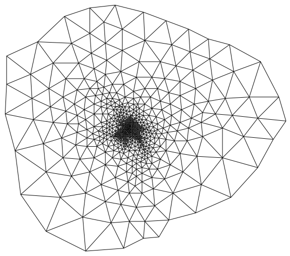
Graph 1



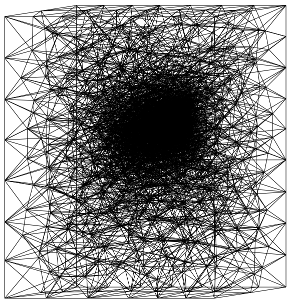
Graph 2



Graph 3



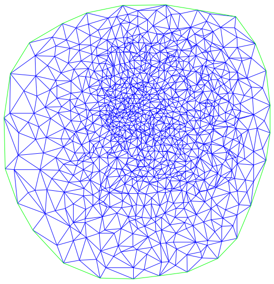
Graph 4



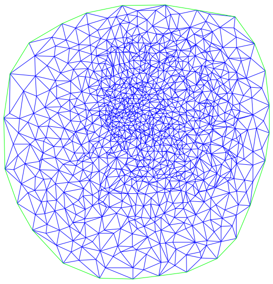
Graph 5

Appendix B

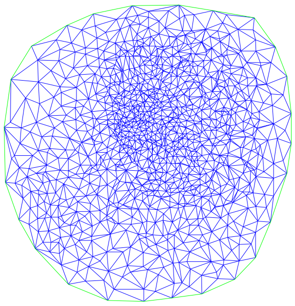
Incremental Graphs



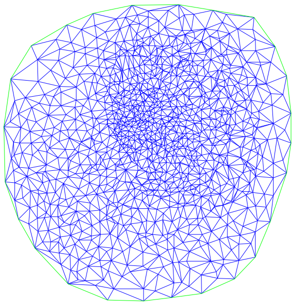
(a) Graph $A1$



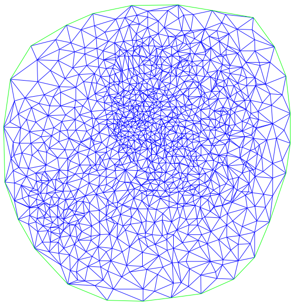
(b) Graph $A2$



(c) Graph $A3$

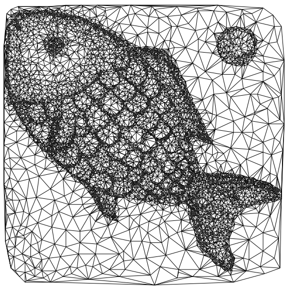


(d) Graph $A4$

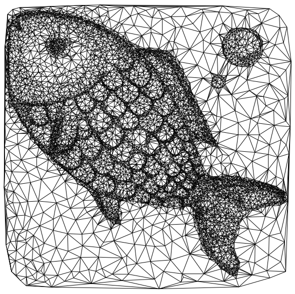


(e) Graph $A5$

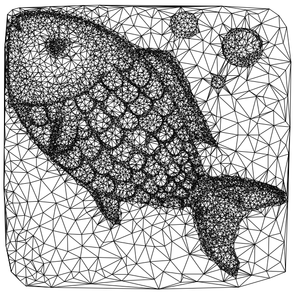
Graph set A: (a) Graph $A1$ with 1071 nodes and 3185 edges; (b) Graph $A2$ with 25 additional nodes in Graph $A1$; (c) Graph $A3$ with 25 additional nodes in Graph $A2$; (d) Graph $A4$ with 31 additional nodes in Graph $A3$; (e) Graph $A5$ with 40 additional nodes in Graph $A4$.



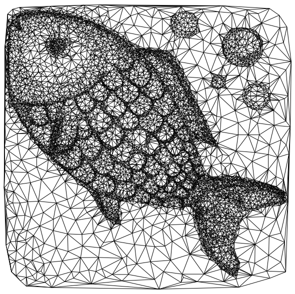
(a) Graph $B1$



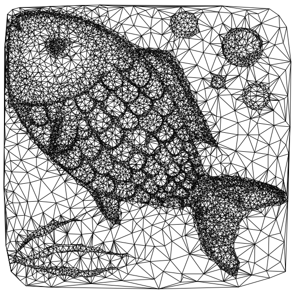
(b) Graph $B2$



(c) Graph $B3$



(d) Graph $B4$



(e) Graph $B5$

Graph set B: (a) Graph $B1$ with 10166 nodes and 30471 edges; (b) Graph $B2$ with 48 additional nodes in Graph $B1$; (c) Graph $B3$ with 139 additional nodes in Graph $B1$; (d) Graph $B4$ with 229 additional nodes in Graph $B1$; (e) Graph $B5$ with 672 additional nodes in Graph $B1$.

Bibliography

[1] I. Angus, G. Fox, J. Kim, and D. Walker. *Solving Problems on Concurrent Processors*, volume 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*, pages 711–718, 1993.

[3] D. Baxter, J. Saltz, M. Schultz, S. Eisentstat, and K. Crowley. A Experimental Study of Methods for Parallel Preconditioned Krylov Methods. *Proceedings of the 1988 Hypercube Multiprocessor Conference*, pages 1698–1711, Pasadena, CA, January 1988.

[4] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. Technical report, ICASE, May 1990.

[5] R. Boppana. Eigenvalues and Graph Bisection: An Average Case Analysis. *Proc. 28th Annual Symposium on Fundations of Computer Science*, pages 280–285, 1987.

[6] B. Van Den Bout and T. Miller. A Travling salesman Objective Function That Works. *Proc. International Conference of Neural Networks*, pages 280–285, 1988.

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

[8] 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.

[9] T. Bultan and C. Aykanat. Parallel Mean Field Algorithm for the Splution of Combinatorial Optimization Problems. *Proc. ICANN-91*, pages 591–596, 1991.

[10] T. Bultan and C. Aykanat. A New Mapping Heuristic Based on Mean Field Annealing. *Journal of Parallel and Distributed Computing*, 16:292–305, 1992.

[11] 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.

[12] N. Chrisochoides, E. Houstis, and C. Houstis. Geometry Based Mapping strategies for PDE Computations. *Proceedings of International Conference Supercomputing*, pages 115–127, 1991.

[13] N. Chrisochoides, E. Houstis, and J. Rice. Mapping Algorithms and Software Environment for Data Parallel PDE Iterative Solvers. *Journal of Parallel and Distributed Computing*, 21:75–95, April 1994.

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

[15] 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.

[16] L. Dagum. Data Parallel Sorting for Particle Simulation. *Concurrency*, 4:241–255, May 1992.

[17] D. Edelsohn. Hierarchical Tree-structures as Adaptive Meshes. Technical report. SCCS-193.

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

[19] C. Farhat. A Simple and Efficient Automatic FEM Domain Decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[20] C. Farhat and M. Lesoinne. Automatic Partitioning of Unstructured Meshes for the Parallel Solution of Problems in Computational Mechanics. *International J. Numer. Meth. Engrg.*, 36(5):745–764, 1993.

[21] M. Fiedler. Algebraic Connectivity of Graphs. *Czechoslovack Math. J.*, 23(98):298–305, 1973.

[22] M. Fiedler. A Property of Eigenvectors of Nonnegative Symmetric Matrices and Its Application to Graph Theorey. *Czechoslovack Math. J.*, 25(100):619–636, 1975.

[23] M. Fiedler. Eigenvectors of Acyclic Matrices. *Czechoslovack Math. J.*, 25(100):607–618, 1975.

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

[25] G. Fox. *A Review of Automatic Load Balancing and Decomposition Methods for The Hypercube.* Ed. M. Schultz, Springer-Verlag, Berlin, 1988.

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

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

[28] G. Fox, R. Williams, and P. Messina. *Parallel Computing Works!* Morgan Kaufmann, San Francisco, CA, 1994.

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

[30] P. Hajek. Cooling Schedules for Optimal Annealing. *Mathatics of Operational Research*, 13:311–329, 1988.

[31] B. Hendrickson and R. Leland. An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations. Technical report, Sandia National Laboratories, Albuquerque, NM 87185, 1992.

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

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

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

[35] D. Hilbert. Uber die steitige Abbildung einer linie auf ein Flachenstuck. *Math. Ann, 38*, 1891.

[36] J. Hopfield and D. Tank. 'Neural' Computation of Decisions in Optimization problems. *Biolo. Cybernet.*, 52:141–152, 1985.

[37] J. Hopfield and D. Tank. Computing with Neural Circuits: A Model. *Science*, 233:625–633, 1986.

[38] J. Hopfield and D. Tank. Collective Computation in Neuronlike Circuits. *Sci. Amer.*, 257:104–114, 1987.

[39] 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.

[40] G. Karypis and V. Kumar. MeTiS: Unstructured Graph Partitioning and Sparse Matrix Ordering System. Technical report, Department of Computer Science, University Minnesota, 1995.

[41] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphgs. Technical report, Department of Computer Science, University Minnesota, 1995.

[42] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

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

[44] Wei-Keng Liao, Chao-Wei Ou, and Sanjay Ranka. Dynamic Alignment and Distribution of Irregularly Coupled Data Aarrays for Scalable Parallelization of Particle-in-cell Problem. *IPPS '96*, 1996.

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

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

[47] J. Malone. Automated Mesh Decomposition and Concurrent Finite Element Analysis for Hypercube Multiprocessors Computers. *Eng.*, 70(1):27–58, 1988.

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

[49] D. Mavriplis. Three Dimensional Unstructured Multigrid for the Euler equations. *AIAA 10th Computational Fluid Dynamics Conference*, June 1991. paper 91-1549cp.

[50] B. Mohar and S. Poljak. Eigenvalues in Combinatorial Optimization. *Combinatorial and Graph-Theoretical Problems in Linear Algebra*, 50:107–151, 1993.

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

[52] 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*, pages 279–283, Taipei, Taiwan, December 1993. A revised version "Fast and Parallel Mapping Algorithms for Irregular Problems" to appear in Jouranl of Supercomputing.

[53] C. C. Paige and M. A. Saunders. Solution of Sparse Indefinite Systems of Linear Equation. *SIAM J. Num, Anal*, 12:617–629, 1974.

[54] M. Parashar and J. Browne. An Infrastructure for Parallel Adaptive Mesh-Refinement Techniques. Technical report, University of Texas at Austin, 1995.

[55] M. Parashar and J. Browne. Distributed Dynamic Data-Structures for Parallel Adaptive Mesh-Refinement. *Proceedings of the International Conference for High Performance Computing*, pages 22–27, Dec 1995.

[56] B. Parlett. The Rayleigh Quotient Iteration and Some Generalizations for Nonnormal Matrices. *Math. Comp.*, 28(127):679–693, 1974.

[57] B. Parlett. *The Symmetric Eigenvalue Problem.* Prentice Hall, Englewood Cliffs, NJ, 1980.

[58] C. Peterson and J. Anderson. Neural Networks and NP-Complete Optimization Problems. *Complex System*, 2:59–89, 1988.

[59] 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.

[60] D. Powers. Graph Partitioning by Eigenvectors. *Linear Algebra Applications*, 101:121–133, 1988.

[61] S. Ranka, J. Wang, and M. Kumar. All-to-Many Personalized Communication on Distributed Memory Machines. *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

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

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

[64] 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.

[65] H. Simon and C. Farhat. TOP/DOMEC; A Software Tool for Mesh Partitioning and Parallel Processing. Technical report, NASA Ames Research Center, July 1993.

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

[67] T. Tokuyama and J. Nakano. Geometric Algorithm for A Minimum Cost Assignment Problem. *Proc. 7th Annual Symposium on Computational Geometry*, pages 262–271, 1991.

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

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

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

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

Biographical Data

Name:	Chao-wei Ou
Date and Place of Birth:	May 11, 1965 Chung-Li, Tao-Yuan, Taiwan, R.O.C.
Degrees Awarded:	Master of Science, 1992 Syracuse University, Syracuse, New York, U.S.A.
	Bachelor of Science, 1987 National Chiao-Tung University, Hsin-Chu, Taiwan, R.O.C.
Professional Experience:	Graduate Research Assistant NPAC at Syracuse University, New York, U.S.A., 1993 - 1995
	System Engineer Stock and Bank Department, FIC Corp., Taiwan, R.O.C., 1989 - 1990
	System Programmer Taiwan Sugar Corporation, Taiwan, R.O.C., 1985 - 1986