COMBINATORIAL ALGORITHMS IN SCIENTIFIC COMPUTING

BY

ALI PINAR

B.S., Bilkent University, 1994
M.S., Bilkent University, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

To my family.

Acknowledgments

Throughout my PhD studies I had the privilege of working with very distinguished mentors. I could not have asked for a better supervisor than Bruce Hendrickson. I worked really hard with him, but still enjoyed it. He will be a role model for me as a researcher and as a person. Working with Mike Heath gave me an opportunity to observe a great teacher. I want to thank him for providing a very good research environment for me at Illinois. I had all the latitude I asked for, but not at the expense of strong support and encouragement. Although I could work with Dave Liu only for a very short time, I learned a lot, and his words of wisdom will inspire me throughout my life. I am grateful to Cevdet Aykanat for inducting me to the culture of research during my studies at Bilkent University, Turkey. I am very proud of working with these great people, and I will try to live up to its responsibility.

I would also like to thank the members of my committee: Jeff Erickson, Eric de Sturler and Laxmikant Kale for their helpful comments on this thesis. I feel fortunate to have learned from exceptional teachers like Herbert Edelsbrunner, Jeff Erickson, Pierre Flener, Ed Reingold, Shung-Hua Teng at the University of Illinois and Bilkent University. My discussions with my colleagues, including Tanya Berger-Wolf, Ümit Çatalyürek, Ali Daşdan, İsmail Haritaoğlu, Mitch Harris, Rebecca Hartman-Baker, Jim Jiao, Tahsin Kurç, Xiang-Yang Li, Vanessa Lopez, Boyana Norris, Prashant Saxena, Alper Üngör, have been extremely educative and contributed to this thesis.

My summer internships at Sandia National Laboratories had a great impact on my thesis. I benefited a lot from the research atmosphere of building 980. I had the opportunity to work with exceptional researchers, foremost among them Erik Boman, Steve Plimpton, David Day, Karen Devine, Bill Hart, Mike Heroux, Rob Leland, and Cindy Phillips.

I wanted to use this thesis as an opportunity to thank people who have contributed to my academic growth. However, getting to this point took much more than just academic work. For everyone, I hope I had used my chances to express what you meant to me and how much I appreciated your help. I cannot do it by making a list of names here, and words cannot express the gratitude I feel for those who have made this moment possible.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

1D One Dimensional

2D Two Dimensional

BCRS Blocked Compressed Row Storage

CCP Chains on chains partitioning

CCS Compressed Column Storage

CG Conjugate Gradient

CP Chains Partitioning

CRS Compressed Row Storage

DFS Depth First Search

DP Dynamic Programming

DVR Direct Volume Rendering

KL Kernighan-Lin Heuristic

MCF Multi-commodity Flow

MPI Message Passing Interface

SCC Strongly Connected Components

SpMxV Sparse Matrix Vector Multiplication

TSP Traveling Salesperson

Chapter 1

Introduction

Scientific computing is primarily concerned with (*i*) developing numerical algorithms for solving equations that result from mathematical models of physical phenomena, (*ii*) implementing such algorithms in computer software, and (*iii*) using the software to simulate the physical phenomena [43]. Numerical algorithms are therefore of central importance in a wide variety of physical applications. Combinatorial algorithms, on the other hand, often play a supporting rather than leading role, but they are nevertheless crucial to the efficient solution of many problems in computational science. For example, the implementation of a numerical algorithm may require complex data structures whose effectiveness directly affects the performance of the algorithm. Another common practice is ordering tasks for improved performance, such as fill-reducing orderings for sparse factorizations and locality-improving orderings for better cache performance. Parallelization, which is inevitable when very large-scale problems are considered, is an important combinatorial issue regardless of the underlying algorithm. This thesis is concerned with a number of interesting and important combinatorial techniques that are relevant to various problems in scientific computing. Six types of problems are addressed, each arising in a different application and requiring different combinatorial techniques for their solution. Before presenting the specifics of these problems, we want to discuss how and where these combinatorial problems arise in scientific computing.

Several of the problems addressed in this thesis arise from adaptive computations. The term "adaptive computation" refers to computation that is influenced by the phenomenon being modeled or the computing environment, i.e., the computation adapts itself to cope with dynamic changes in the problem and its solution, or in the computational sources provided for the computation. One example is adaptive mesh refinement, where the discrete mesh being used is adapted to maintain a given level of accuracy in the computed solution. Another example is transient dynamics, a typical example of which is simulation of an automobile crash. During a crash simulation, contacts between different components must be detected, and the computation must be adapted accordingly. For many applications, performance motivates adaptation. Adaptive mesh refinement can cause load imbalance among processors, which can be resolved by redistribution. Another example is multi-physics simulations, such as a rocket simulation, where the internal chamber grows as solid propellant is consumed, thereby changing the domains of both solid and gas components. Consequently, the load distribution among processors varies during computation and needs to be readjusted. In particle simulations, where spatial decomposition is commonly used for its good performance, particle densities, and thus processor loads, will vary.

Another reason for moving data during computation is that different software components may favor different data distributions. Plimpton et al. [85] use two different partitions for two phases of their crash simulation code PRONTO, for better performance. The first phase predicts how the mesh points would move without any contacts, and the second phase detects contacts and corrects the computation. The first phase favors topological partitioning, whereas the second favors a geometric partition. A more widely-applicable example would be solvers not sticking with the distribution provided by the solver and determining their own data distribution. Most current solvers use iterative methods, which usually require preconditioners for rapid convergence. The preconditioner often dominates the computation, and thus balance must be based primarily on the preconditioner being used. A more important motivation for using different data distributions for iterative solvers is that distribution might be the determining factor for breakdown or convergence for the solver.

These are only a few examples of the many applications that require periodic redistribution of data among processors. In this thesis, we study two types of problems motivated by redistribution of work: finding a new distribution and providing communication support to implement this distribution for subsequent computation. Three chapters are dedicated to distribution of work among processors. Chapter 3 presents efficient algorithms for one-dimensional partitioning of workload arrays. Chapter 6 studies graph partitioning problems—mostly motivated by different preconditioners—where the objective is complex, and it is not possible to achieve balance by merely assigning weights to vertices. In Chapter 7, we introduce flexibly assignable tasks and show how to exploit them for better load balance.

Redistribution of data among processors requires communication support. Two problems concerning communication support for data redistribution are studied in Chapter 5. First, we describe how processors can determine a new communication pattern after a computation is first defined. Second, we discuss how to exchange data among processors subject to memory constraints.

The need for adaptation is not always performance, but may instead be for correctness of the simulation or the underlying algorithm. In a crash simulation, for instance, contact between different components of the system being modeled must be detected, and the computation must be adapted accordingly. Another example arises in the discrete ordinates method for radiation transport [86], which motivates our work in Chapter 4. The radiation equations are approximated by angular discretization, and each angle imposes precedence constraints on the elements. This requires a topological ordering of the elements, but such an ordering does not exist when a group of elements involves a circular dependence. These groups must be identified and handled separately for correctness of the computation. It is important to note that if the mesh moves during the course of computation, groups of elements with circular dependency must be reidentified as the mesh moves, requiring a parallel algorithm. Chapter 4 presents an efficient algorithm easily amenable to parallelization.

Finally, it is important to achieve maximum performance from each processor in any application (even though it may degrade scalability curves). This includes better cache utilization and reducing the number of extra operations such as ordering for minimum fill in direct methods, or reducing number of load operations to exploit sparsity, which is the subject of Chapter 2.

1.1 Outline

The following six chapters of this thesis study six different problems. Each chapter is self-contained. In this section, we provide descriptions of the problems studied in this thesis. These problems are at the intersection of two disciplines: computer science on one hand and computational science and engineering on the other. All of the problems are motivated by engineering applications, and identifying and formulating the problem requires some understanding of the application. The solution techniques, on the other hand, use fundamental methods of computer science. We consider both the theoretical and practical aspects of these problems. The work includes complexity analysis, and design and analysis of algorithms, as well as their implementations and empirical evaluations.

- **Chapter 2: Improving Performance of Matrix-Vector Multiplication**

  Sparse matrix-vector multiplication, one of the most important computational kernels in scientific computing, often suffers from poor memory performance due to memory indirections necessary to exploit sparsity. We propose alternative data structures along with reordering algorithms to improve their effectiveness to reduce the number of memory indirections. We pack nonzeros in consecutive positions into a block so that only one memory indirection is required per block. We also consider the problem of permuting nonzeros of a matrix into contiguous positions and show that it can be stated as a traveling salesperson problem. Experiments with our method show runtime improvements of up to 33% and improvements of 25% on average. We also discuss alternative methods to improve memory performance.

- **Chapter 3: One-Dimensional Partitioning of Workload Arrays**

  Although asymptotically efficient algorithms have been designed for one-dimensional partitioning, practitioners still use heuristics, in the hope of achieving good partitions, apparently under the misconception that exact algorithms are difficult to implement and are not run-time efficient. We propose run-time efficient algorithms along with implementation tips. Experiments show that, on average, our algorithms are 100 times faster than a matrix-vector product when partitioning for 64 processors.

  Partitioning for heterogeneous systems leads to two different problems, depending on whether the order of the processors is predetermined. When the order of the processors is predetermined, the problem can be solved optimally in polynomial time, and we show how our techniques can be modified to solve this version of the problem. However, if the order of processors is not specified, then we prove that the problem becomes NP-Complete.

- **Chapter 4: Identifying Strongly Connected Components in Parallel**

  The standard serial algorithm for finding strongly connected components is based on depth-first search, which is difficult to parallelize. We propose a divide-and-conquer algorithm using reachability information that is amenable to parallelization. The expected running time for our algorithm is $O(|E| \log |V|)$, where $|E|$ and $|V|$ denote the numbers of edges and vertices respectively, in the graph. Our work is motivated by the discrete ordinates method for modeling radiation transport. In this method, the object to be studied is modeled by finite elements. Each element is a vertex in a graph and an edge connects elements that share a face. The radiation equations are approximated by an angular discretization, and edges in the graph are directed to align with the angle. Computations for an element can be performed only after those of all its predecessors. Numerical calculations must be modified if the graph has a cycle. Our algorithm detects these cycles to enable appropriate modification.

- **Chapter 5: Communication Support for Adaptive Computation** Many parallel applications involve periodic redistribution of workloads and associated data, which requires special communication routines. The first problem below is concerned with determining the communication pattern after a calculation is first defined, and the second problem studies how to migrate data subject to memory constraints.

  – After subdomains are assigned, processors can determine what data they need to receive, but will not know which processors contain those data. An efficient solution enables a more convenient interface between different software components and reduces the burden on a repartitioning tool to update the communication information. A simple solution is to have the processors exchange messages in a ring fashion, but this is slow and is not scalable. We propose using a distributed *directory*—a rendezvous algorithm—where each processor is responsible for maintaining directory information for a subset of the components.

  – While redistributing data among processors, each processor allocates space for its incoming data, posts asynchronous receives, and then sends its outgoing data. Memory for outgoing data can be deallocated only after the send is complete, which requires each processor to have space simultaneously for both its outgoing and incoming data. To overcome this problem, we can send data in phases. After each phase, processors can free the memory of the data they sent, to make it available for the next phase.

We prove that scheduling messages for the minimum number of phases is NP-Complete, and show how the problem can be phrased as an instance of multi-commodity flow. We also consider a continuous approximation that yields solutions with at most two more phases than that of a discrete solution. Finally, we devise a practical 1.5-approximation algorithm.

- **Chapter 6: Graph Partitioning for Complex Objectives**

  Unstructured computations can often be described by undirected graphs in which vertices correspond to computations and edges reflect data dependencies. A graph partitioner can be employed to partition the vertices of the graph into disjoint sets of equal weight (for load balancing) while keeping the number of edges crossing between sets small (low communication volume). For some computations however, the load is a complicated function of the partition, and the work per subdomain cannot be assessed until the partition is computed. Examples of this phenomenon include balancing computation plus communication; domain decomposition techniques where incomplete or complete factorization is used at each subdomain; and overlapped subdomain partitions where boundary vertices are duplicated in neighboring domains. We propose a framework for addressing such complex partitioning problems and apply these ideas to achieve equal-sized, overlapped subdomains and balanced computation plus communication load.

- **Chapter 7: Exploiting Flexibly-Assignable Work to Improve Load Balance**

  In many parallel applications, distribution of data implies distribution of work among processors, but there are exceptions where some tasks can be assigned to one of multiple processors without altering the total volume of communication. Examples include the force computation between two particles in different processors, or node computations in finite-element simulations when element partitions are used. We examine how to exploit this flexibility in assignment of tasks to improve load balance. We first model the problem as a flow problem and use combinatorial techniques to solve it. Our parametric search algorithms use maximum flow algorithms for probing a candidate optimal solution value. We also define augmenting paths and cuts for this problem, and show that any algorithm based on augmenting paths can be used to solve the task assignment problem optimally. Then we show that a continuous version of this problem reduces to the well-studied linearly constrained least squares problem, which is easily amenable to parallelization. We also show that all these techniques generalize to balance loads for heterogeneous processors.

Chapter 2

Improving Performance of Sparse Matrix-Vector Multiplication

One of the most important computational kernels in scientific computing is multiplying a sparse matrix by a vector. Many algorithms use sparse matrix-vector multiplication (SpMxV) in their inner loop, such as iterative methods to solve systems of linear equations. Repeated execution of this operation potentially amortizes the cost of a preprocessing phase, and leads to computational savings in subsequent executions. Despite its simplicity, SpMxV has been the subject of numerous researches, since its importance makes even small improvements very valuable.

Data structures for sparse matrices usually have two components: (*i*) an array that stores all floating-point entries of the matrix, (*ii*) arrays that store sparsity structure of the matrix, i.e., pointers to the locations of the floating-point entries in the matrix. To exploit sparsity of the matrix, use of pointers is inevitable but often limits memory system performance. One reason for this is that pointers usually lead to poor cache utilization, since they lack spatial locality. The number of cache misses for the right- and/or left-hand-side vectors can increase dramatically if the sparse matrix has an irregular sparsity structure. Another important problem, which is the subject of this chapter, is that memory indirections (pointers) require extra load operations. In sparse matrix operations, the number of floating-point operations per load operation is lower than that of dense matrix operations, limiting overall performance.

In this chapter, we propose alternative data structures, as well as reordering algorithms to increase the effectiveness of those data structures, to reduce the number of memory indirections in SpMxV. Toledo [94] proposed identifying 1×2 blocks of a matrix and writing the matrix as the sum of two matrices, the first of which contains all the 1×2 blocks and the second contains the remaining entries. Each 1×2 block requires only one memory indirection. In an alternative scheme, we pack all nonzeros in contiguous locations into a block to reduce further the number of memory indirections, because only one memory indirection is required for all nonzeros in a block. However, we then need to know how many nonzeros are in each block, which requires an extra array in storage and an extra loop in SpMxV.

We also propose reordering matrices to increase the effectiveness of our data structures. The objective of reordering is to permute the nonzeros of the matrix into contiguous locations, as much as possible, to enlarge dense blocks. We show that this problem is NP-Complete, and thus heuristics must be used for a practical solution. We propose a graph model to reduce the problem to the well-studied traveling salesperson problem, enabling adoption of heuristics designed for this problem.

We verify experimentally the effectiveness of the proposed data structures and algorithms. Our new data structures and reordering techniques produce improvements of up to 33% and improvements of 25% on average.

These results motivate exploring other ways of exploiting dense blocks within a sparse matrix to reduce load operations. First, most sparse matrices either originally are or can be permuted to a banded form, where nonzeros are gathered around the diagonal. Such a matrix potentially has a lot of nonzeros in consecutive diagonal positions, which can be exploited to decrease the number of load operations. Second, most large matrices come from discretization of partial differential equations. These matrices (meshes) have certain properties that yield special data structures and/or algorithms. We propose a data structure for 2-dimensional meshes, which requires only 2 bits of information for each nonzero to store the sparsity structure, as opposed to 32 bits (long integer) of the conventional storage scheme.

The remainder of this chapter is organized as follows. § 2.1 discusses the shortcomings of current sparse matrix data structures and proposes alternatives. § 2.2 states a new matrix reordering algorithm to permute the nonzeros of the matrix into contiguous locations. Experimental results are presented in § 2.3. Directions for future research are presented in § 2.4. Finally we conclude with § 2.5.

for $i \leftarrow 1$ **to** m **do**
  $y[i] \leftarrow 0$;
  for $j \leftarrow ROW[i]$ **to** $ROW[i+1]-1$ **do**
    $y[i] \leftarrow y[i]{+}DATA[j] * x[COL[j]]$;

Figure 2.1: SpMxV algorithm in compressed row storage

Parts of this work has been published in [79].

2.1 Sparse Matrix Data Structures

The most widely used sparse matrix storage scheme is *Compressed Row Storage* (CRS). As the name implies, this scheme stores a sparse matrix as a sequence of compressed rows. Three arrays are employed: $DATA$, COL and ROW. The nonzeros of the sparse matrix A are compressed into array $DATA$ in a rowwise manner. The column index of each nonzero entry is stored in the array COL, i.e., $COL[i]$ is the column index of the nonzero entry stored in $DATA[i]$. Finally, ROW stores the index of the first nonzero of each row with a sentinel at the end. Figure 2.1 presents a matrix-vector multiplication algorithm using compressed row storage.

Sparse matrix-vector multiplication (SpMxV) algorithms tend to suffer from poor memory performance. One reason for this is ineffective cache utilization. Locality in the computation is very important for good cache performance. We use the term *temporal* locality to refer to locality in time, i.e., the same data items being reaccessed in the near future. *Spatial* locality refers to locality in space, i.e., accessing nearby memory locations. As can be seen in the SpMxV pseudocode in Figure 2.1, temporal locality is limited to the y-vector. There is good spatial locality for arrays $DATA$ and COL, but no temporal locality. Spatial locality in the left-hand-side vector y is good, but the access pattern for the right-hand-side vector x can be quite irregular, causing excessive cache misses. Reordering the matrix to reduce cache misses was proposed by Das et al. [22], who suggested reducing the bandwidth of the matrix. Temam and Jalby [93] analyzed the number of cache misses as a function of the bandwidth for various cache parameters. Burgess and Giles [13] experimented with various ordering algorithms and found that reordering the matrix improves performance compared with random ordering, but they did not detect a notable sensitivity to

$$
\begin{pmatrix} x & x & x & & \\ & x & & x & \\ x & x & & & \\ & & & x & x \\ & & x & x & x \end{pmatrix} = \begin{pmatrix} x & x & & & \\ & & & & \\ x & x & & & \\ & & & x & x \\ & & x & x & \end{pmatrix} + \begin{pmatrix} & & x & & \\ & x & & x & \\ & & & & \\ & & & & \\ & & & & x \end{pmatrix}
$$

$$
A \qquad = \qquad A_{12} \qquad + \qquad A_{11}
$$

Figure 2.2: Fixed size blocking with 1×2 blocks

the particular ordering method used. Toledo [94] studied reordering, along with other techniques, and reported that Reverse Cuthill-McKee ordering yielded slightly better performance, but the differences were not significant.

Another problem with SpMxV is that the ratio of load operations to floating point operations is higher compared with dense matrix operations. One extra load operation is required to find the column index of each nonzero entry for each multiply-add operation. The innermost statement in the algorithm in Figure 2.1 requires three load operations for two floating-point operations. This not only increases the total number of load instructions, but also can cause the load units to be a bottleneck, especially in recent architectures as discussed in [94].

The following two sections present data structures to decrease the number of memory indirections during SpMxV operation.

2.1.1 Fixed-Size Blocking

In this approach, the matrix is written as the sum of several matrices, some of which contain the dense blocks of the matrix with a prespecified size. For instance, given a matrix A, we can decompose it into two matrices A_{12} and A_{11}, such that $A = A_{12} + A_{11}$, where A_{12} contains the 1×2 dense blocks of the matrix and A_{11} contains the remainder. An example is illustrated in Figure 2.2.

A simple greedy algorithm is sufficient to extract the maximum number of $1 \times l$ blocks in a matrix, where $1 < l \leq n$. However, the problem is more difficult for $k \times l$ blocks for $k > 1$.

Exploiting dense blocks can reduce the number of load operations, as well as the total memory requirement. Only one index per block is required, since the column index of the first nonzero defines indices for all the others. Moreover, the entry of the right-hand-side vector x can be used multiple times—as opposed to once in the conventional scheme—after a load operation, when nonzeros from different columns are packed.

A similar problem has been studied in the context of vector processors [1], but those efforts concentrate on finding fewer blocks of larger size such as very long diagonals, whereas we are interested in much smaller blocks.

2.1.2 Blocked Compressed Row Storage

In this section we propose a new sparse matrix storage scheme designed to reduce the number of load operations. The idea of this scheme is to exploit the nonzeros in contiguous locations by packing them. Unlike fixed-size blocking, the blocks will have variable lengths. This enables longer nonzero strings to be packed into a block. As in fixed-size blocking, if we know the column index of the first nonzero in a block, then we will also know the column indices of all its other nonzeros. In other words, only one memory indirection (extra load operation) is required for each block.

This storage scheme requires an array *NZPTR* (of length the number of blocks) in addition to the other three arrays used in CRS: a floating-point array *DATA* (of length the number of nonzeros) to store the nonzero values, an array *COL* (of length the number of blocks) to store the column number of the first nonzero for each block, and an array *ROW* (of length the number of rows) to point to the position where the blocks of each row start. *NZPTR* stores the location of the first nonzero of each block in array *DATA*. We refer to this storage scheme as *blocked compressed row storage* (BCRS). Figure 2.3 presents an example of BCRS, and the SpMxV operation using this storage is presented in Figure 2.4.

$$A = \begin{pmatrix} 5. & 1. & 7. & 0 & 0 \\ 0 & 1. & 0 & 2. & 3. \\ 0 & 2. & 4. & 0 & 0 \\ 0 & 0 & 1. & 3. & 0 \\ 0 & 6. & 0 & 0 & 3. \end{pmatrix} \qquad \begin{array}{rcl} \mathit{DATA} & = & (5., 1., 7., 1., 2., 3., 2., 4., 1., 3., 6., 3.) \\ \mathit{COL} & = & (1, 2, 4, 2, 3, 2, 5) \\ \mathit{ROW} & = & (1, 2, 4, 5, 6, 8) \\ \mathit{NZPTR} & = & (1, 4, 5, 7, 9, 11, 12, 13) \end{array}$$

Figure 2.3: Example of blocked compressed row storage

This storage scheme reduces extra load operations but requires an extra loop during SpMxV operation and thus suffers from additional loop overhead. If sizes of the blocks are small, overhead due to the extra loop will dominate the gain due to decreased load operations. Thus, effectiveness of this storage scheme depends directly on the sizes of the blocks in the matrix.

```
for i ← 1 to m do
  y[i] ← 0;
  for j ← ROW[i] to ROW[i + 1] − 1 do
    startcol ← COL[j];
    t ← 0;
    for k ← NZPTR[j] to NZPTR[j + 1] − 1 do
      y[i] ← y[i]+ DATA[k] ∗ x[startcol + t];
      t ← t + 1;
```

Figure 2.4: SpMxV with blocked compressed row storage

The total volume of storage has a similar tradeoff. Two numbers (one in *COL* and one in *NZPTR*) are stored for each block in the matrix, as opposed to one number for each nonzero. Thus, if the blocks are large enough, the total storage size can be reduced significantly.

2.2 Reordering to Enlarge Dense Blocks

The previous section described our data structures to exploit dense blocks of a matrix. The effectiveness of these data structures depend directly on the availability of dense blocks. In this section we describe reordering algorithms to enlarge dense blocks of a matrix. First, we will formally define the problem and prove its NP-Completeness. Then we propose a graph model to reduce the problem to the traveling salesperson problem (TSP), and discuss briefly heuristics we used for solving TSP.

2.2.1 Problem Formulation

Our objective in reordering the matrix is to increase the sizes of dense blocks in a row, i.e., to align the nonzeros in a row in consecutive locations as much as possible. This requires reordering of columns and is not affected by ordering of rows. Thus, rows of the matrix can be reordered without disturbing the aligned nonzeros within a row. The same reordering for instance can be applied to rows to preserve symmetry or to improve temporal locality of the x-vector. These techniques can also be used to align nonzeros within a column by interchanging the roles of columns and rows. A formal description of the problem follows.

Given an $m \times n$ matrix $A = (a_{ij})$, find an ordering of columns to maximize the number of (i, j) pairs satisfying $a_{ij} \neq 0$ and $a_{ij+1} \neq 0$, i.e.,

$$Maximize \quad \#((i,j) : 1 \leq i \leq m, 1 \leq j < n : a_{ij} \neq 0 \ \ and \ \ a_{ij+1} \neq 0).$$

Below, we prove NP-Completeness of the problem, using reduction from the Hamiltonian Path problem.

Theorem 2.2.1 *The following problem is NP-Complete.*

Given an $m \times n$ matrix $A = (a_{ij})$, decide if there exists an ordering of columns, where the number of (i, j) pairs satisfying $a_{ij} \neq 0$ and $a_{i,j+1} \neq 0$ is greater than or equal to a given bound B.

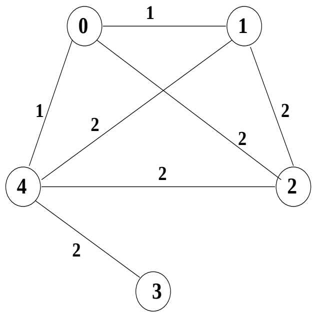
Proof: Given a simple graph $G = (V, E)$ (no loops, no parallel edges), construct an $|E| \times |V|$ matrix A. In this matrix each row represents an edge and each column represents a vertex in the graph. Let the ith column represent the ith vertex. The nonzero structure matrix A is defined such that there are nonzeros at a_{ji} and a_{jk} for each edge $e_j = (v_i, v_k)$.

To increase the size of dense blocks, we have to bring the columns of adjacent vertices to consecutive locations in the reordered matrix. Notice that two adjacent columns can share nonzeros in at most one row, because there are no parallel edges. There can be at most $|V| - 1$ blocks of size 1×2 after reordering, achieved when the vertices of consequent columns share an edge in the graph, which defines a Hamiltonian path in the graph.

It is also easy to see the problem is in NP, since a given solution can be verified in polynomial time. Thus, we can conclude that the problem is NP-Complete. $\square$

This proof is for general matrices. Many matrices however, have very special structures (e.g., matrices arising from meshes), and it might be possible to exploit these structures to design polynomial-time algorithms to find optimal solutions.

$$
\begin{array}{c}
\begin{pmatrix} 0 & 1 & 2 & 3 & 4 \end{pmatrix} \\
\begin{pmatrix}
x & x & x & & x \\
 & x & x & & x \\
x & & x & & \\
 & & & x & x \\
 & & & x & x
\end{pmatrix}
\end{array}
$$



$$
\begin{array}{c}
\begin{pmatrix} 0 & 2 & 1 & 4 & 3 \end{pmatrix} \\
\begin{pmatrix}
x & x & x & x & \\
 & x & x & x & \\
x & x & & & \\
 & & & x & x \\
 & & & x & x
\end{pmatrix}
\end{array}
$$

Figure 2.5: Edge weight computation. Graph shows edge weights of matrix on left-hand side. Matrix after reordering is presented on right.

2.2.2 Reduction to TSP

Theorem 2.2.1 shows that we must resort to heuristics for a practical solution. Such an ordering problem is naturally close to TSP. However, we are looking for a path, not a tour (cycle) of vertices. Although this is a slightly different version of the problem, heuristics designed to find a tour can still be used directly or adapted to find a path of vertices. Thus, our strategy will be first to define a graph model that reduces our matrix reordering problem to the well-studied TSP and use heuristics already designed for TSP to reorder our matrices. For the sake of presentation, we will refer to the maximization version of TSP, i.e.,

Given a graph $G = (V, E)$ and a weighting function w on its edges, find a path $\langle v_0, v_1, \ldots v_{|V|-1}\rangle$ to maximize

$$\sum_{0 \le i < |V|} w(v_i, v_{i+1}).$$

Notice that the weight of the edge $(v_{|V|}, v_0)$ is not included, since we need a path, not a tour.

Since we are trying to reorder columns of the matrix, the graph that reduces the matrix reordering problem to TSP has vertices representing the columns of the matrix. The weight of an edge between two vertices is defined as the number of rows where both respective columns have a nonzero, i.e.,

$$w(u, v) = \#(i : 1 \le i \le m : a_{iu} \ne 0 \;\; and \;\; a_{iv} \ne 0)$$

An alternative definition, which also gives the basics of an algorithm, is as follows. Let $S_A = (s_{ij})$

be a matrix of zeros and ones with the same nonzero structure as A, i.e.,

$$s_{ij} = 1 \quad \Leftrightarrow \quad a_{ij} \neq 0.$$

Let $W = S_A S_A^T$. The weight function can be defined by the matrix $W = (w_{ij})$ as $w(u, v) = w_{uv}$. Figure 2.5 illustrates an example of edge weight computation. In this example the weight of the $(0, 1)$ edge is equal to 1 because columns 0 and 1 share a nonzero only in the first row, whereas the weight of the $(1, 4)$ edge is equal to 2 because columns 1 and 4 share nonzeros in the first and second rows. Notice that although a weight (possibly zero) is assigned to any pair of vertices in the graph (any ordering is a feasible solution for the matrix reordering problem), W is expected to be sparse, which must be exploited for the sake of efficiency (or even existence) of a practical solution.

If the vertices u and v are in consecutive locations in the TSP solution, then the respective columns will be in consecutive locations in the ordered matrix, and $w(u, v)$ nonzeros will be in contiguous locations. The larger the weights of the edges in the tour, the greater the number of nonzeros in contiguous locations will be. In general, finding a tour with maximum weight in this graph corresponds to finding an ordering with maximum number of contiguous nonzeros. To be more precise, the number of blocks (as in BCRS scheme) in the reordered matrix is equal to the number of nonzeros in the matrix minus the total weight of the TSP tour, since each unit of weight corresponds to locating a nonzero adjacent to another one, thus decreasing the number of blocks by one. Consequently, a tour with maximum weight describes an ordering with minimum number of blocks.

In the example of Figure 2.5, the matrix has 13 nonzeros. The TSP solution used to reorder the matrix is $0 \to 2 \to 1 \to 4 \to 3$, the total weight of which is $2 + 2 + 2 + 2 = 8$. The number of blocks in the reordered matrix is $13 - 8 = 5$.

When fixed-size blocking will be used, pieces of the contiguous blocks will potentially be wasted. This does not change the basic combinatorial structure of the problem, and can be handled by minor changes to the heuristics.

2.2.3 Heuristics for TSP

As in many other combinatorial optimization problems, heuristics for TSP can be classified into two groups: constructive and improvement. Constructive heuristics directly construct a solution for the problem, whereas improvement heuristics start with a given solution and try to improve that solution by searching the neighborhood. A comprehensive discussion of TSP solution methods can be found in [75, 88]. In this work, we adopted heuristics from the literature instead of designing our own. Here we discuss briefly the heuristics used in our experiments. Since matrix reordering is proposed as a preprocessing step, we focused on computationally efficient heuristics.

The simplest constructive heuristic is to use the initial ordering of the matrix. Matrices often have some natural ordering that can be used as an initial solution. We also used a vertex insertion procedure. The process starts with an initial path from a random vertex, and vertices are inserted one by one to the path until all are included. At each step, a vertex that is not in the current path is randomly chosen and inserted into the path, so that the sum of weights of edges to its successor and predecessor is maximum. The final constructive heuristic we used is to start with a random vertex as the initial path and proceed by inserting vertices to the end of the path. At each step, the vertex that is connected to the last vertex of the current path with the heaviest edge weight is inserted at the end of the path.

Improvement heuristics have the same flavor as constructive heuristics. One of the heuristics we used depends on vertex insertion. But this time, a vertex is first removed from the current path, unlike the case in constructive heuristics, and then reinserted to maximize the total weight of the tour. We also used edge or path reinsertion procedures, which are similar to vertex insertion. In these heuristics, edges (paths of two vertices) or longer paths are reinserted instead of vertices (named Or-opt procedure in [75]).

2.3 Experimental Results

We implemented our proposed data structures and reordering algorithms in the `C` language and compared their performances with that of conventional compressed row storage (CRS) scheme. We experimented with a collection of matrices selected from the Harwell-Boeing sparse-matrix

Table 2.1: Statistics for test matrices

Name	N	NNZ
psmigr_1	3140	543162
bcsstk28	4410	111717
lp_qap12	8856	38304
bcsstk29	13992	316740
cavity26	4562	138187
bcsstk30	28924	1036208
bcsstk32	44609	1029655
dw8192	8192	41746
memplus	17758	126150
bcsstk35	30237	740200

collection, statistics for which are presented in Table 2.1. All experiments were performed on a Sun Enterprise 3000.

In the first set of experiments, we investigated the effectiveness of our ordering techniques. Figure 2.6 presents our experimental results with 1×2 blocked matrices, where the ordering is determined by the initial ordering, reverse Cuthill-McKee (RCM) ordering, and TSP ordering (described in Section 2.2). TSP solutions are generated by starting with the initial ordering and then using a vertex insertion procedure to improve the initial solution.

In Figure 2.6, SpMxV times are normalized with respect to SpMxV times with the conventional CRS scheme. 1×2 blocking after TSP ordering always improves performance and is always superior to the other two ordering methods. The difference becomes very significant for `dw8192` and `bcsstk28`. On average, TSP ordering reduces the runtime of a SpMxV operation to 79% of that of the conventional scheme, whereas RCM ordering and initial ordering are limited to only 95% and 92%, respectively.

Figure 2.7 compares fixed-size and variable-sized blocking (BCRS). Again, SpMxV times are normalized with respect to SpMxV times with the conventional CRS scheme. Fixed-size blocking is superior to BCRS for most matrices and on average. BCRS outperforms fixed-size blocking for four matrices: `bcsstk28, cavity26, bcsstk30`, and `bcsstk32`.

Figure 2.8 presents results of experiments for different block sizes. Increasing the block size decreases the number of blocks, but each block saves more load operations. As the results show, using 1×3 blocks instead of 1×2 increases the performance by 3% on average, but further increasing

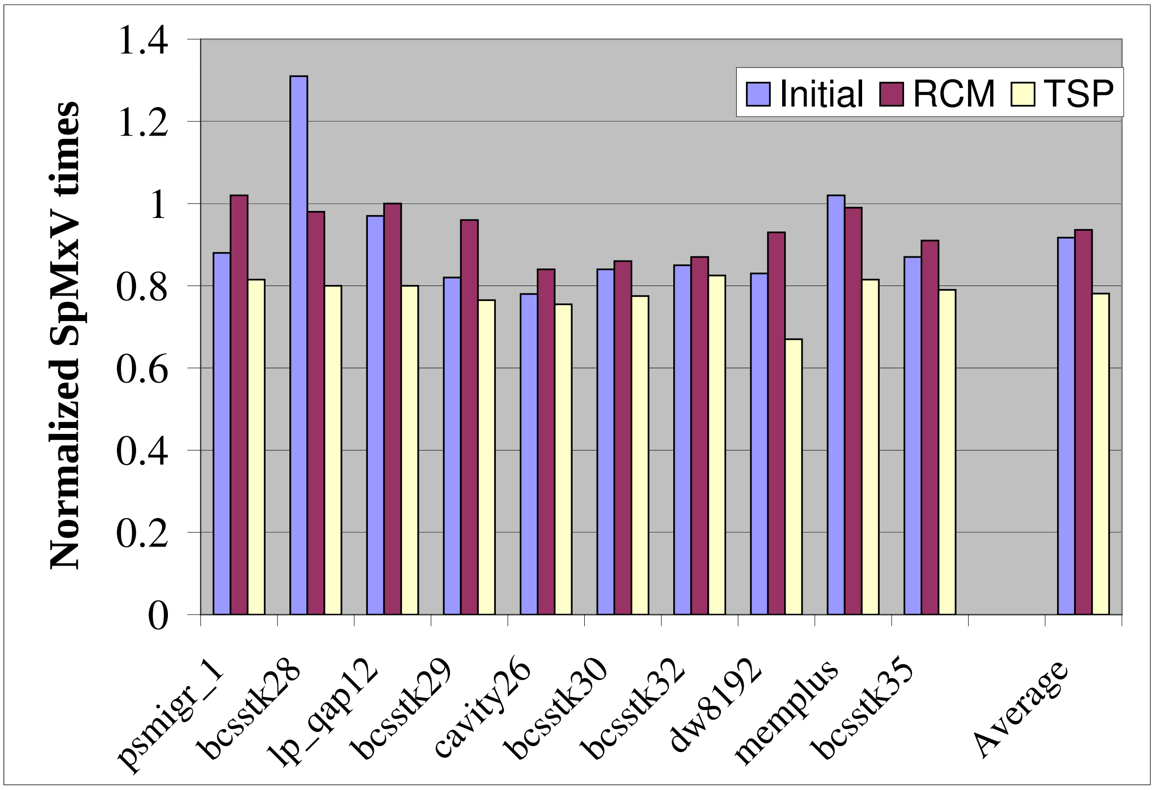


Figure 2.6: Effectiveness of ordering. Performances of 1×2 blocking on initial, reverse Cuthill-McKee (RCM), and TSP orderings are compared.

the block size does not help.

We have also worked on minor modifications to improve the performance of BCRS. Figure 2.9 shows the results for these experiments. In this figure, BCRS_1 corresponds to using blocked compressed row storage only for blocks of size greater than 1 and using the conventional scheme for the remainder, and its performance is 3% better than that of BCRS. This shows the significance of the third loop required by BCRS on performance. To avoid this loop, but still to exploit long stripes, we tried decomposing the matrix into three matrices, one with only 1×4 blocks, one with 1×2 blocks, and the remainder. B421 in Figure 2.9 correspond to this structure, and performs 7% better than BCRS on average.

Our techniques are proposed for preprocessing, thus runtimes are important. Our employed

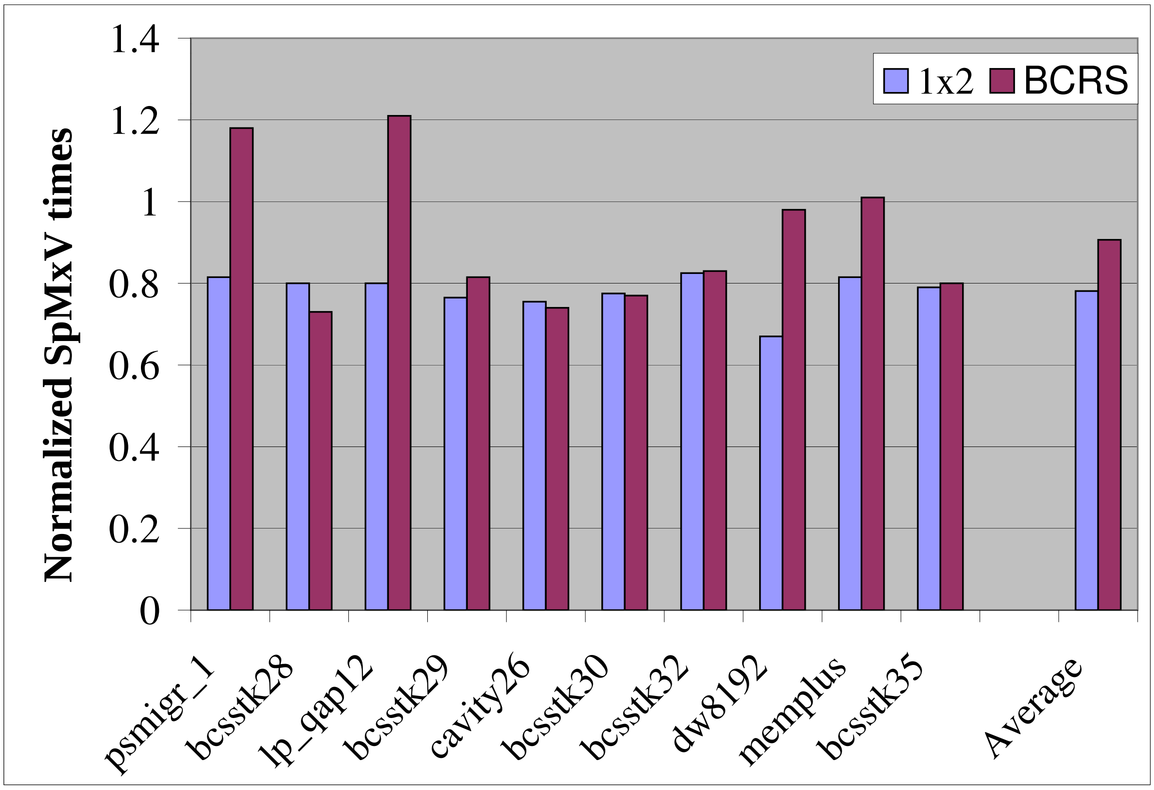


Figure 2.7: Comparison of fixed-size 1×2 and blocked compressed row storage (BCRS).

heuristics are simple and runtime efficient. The whole preprocessing took 3 to 4 times the time of a single SpMxV operation. This means that with 25% improvement in subsequent SpMxV operations, preprocessing times can be amortized in approximately 15 iterations. All these results show that significant savings can be achieved by exploiting the dense blocks in a matrix.

2.4 Future Work

The main idea of this work is to exploit regular patterns in a sparse matrix such as the dense blocks to reduce the number of memory indirections. So far we have exploited the density within rows. In this section, we investigate alternative ways to reduce the number of memory indirections.

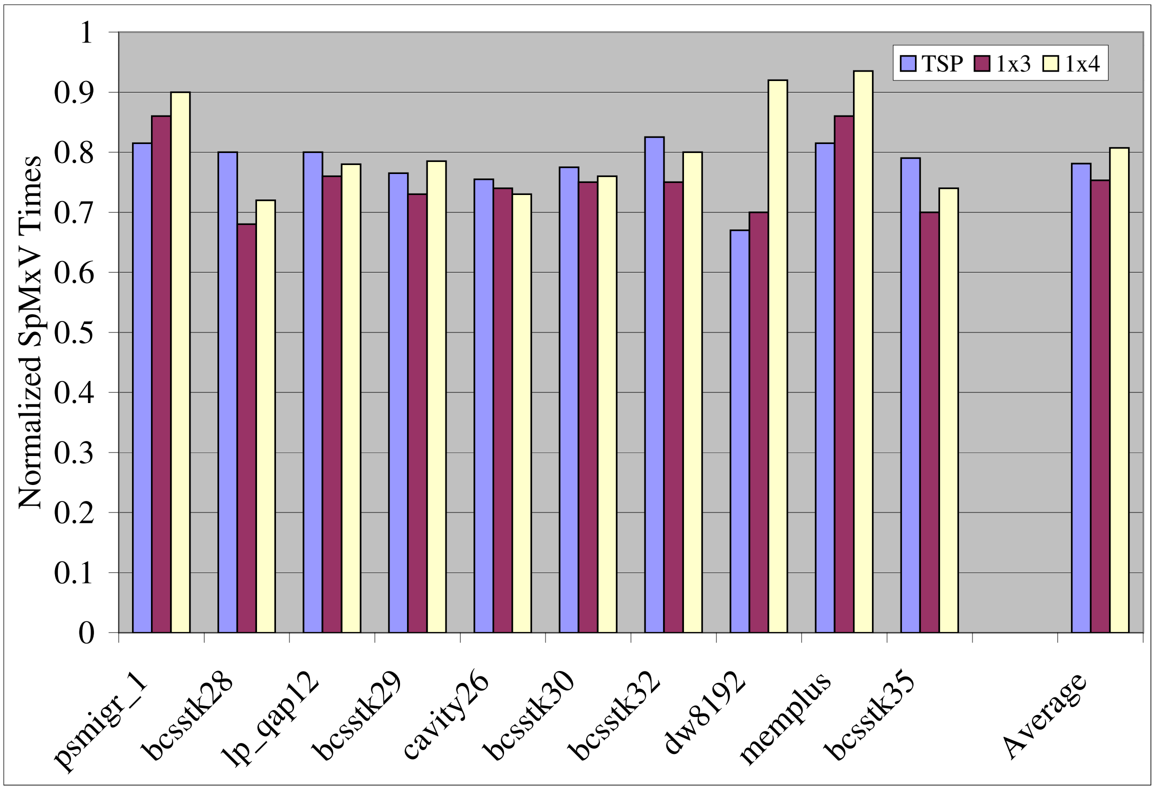


Figure 2.8: Comparison of different block sizes for fixed-size blocking

2.4.1 Finding Long Diagonals

Most sparse matrices either already are or can be permuted to be in banded form, where most nonzeros are close to the main diagonal. This observation can be exploited in two ways. First, if the diagonal region is quite dense, then it can be stored as a dense matrix, replacing structural zeros with numerical zeros if needed. Conventional techniques can be used for the remaining nonzeros. Second, banded structure potentially involves diagonal stripes, which can be exploited just like the row stripes discussed so far.

Fixed-size Dense Band

In this method, we will decompose the matrix into a dense matrix that stores all nonzeros within a specified distance of the main diagonal, and a sparse matrix for the remaining nonzeros. Storing the

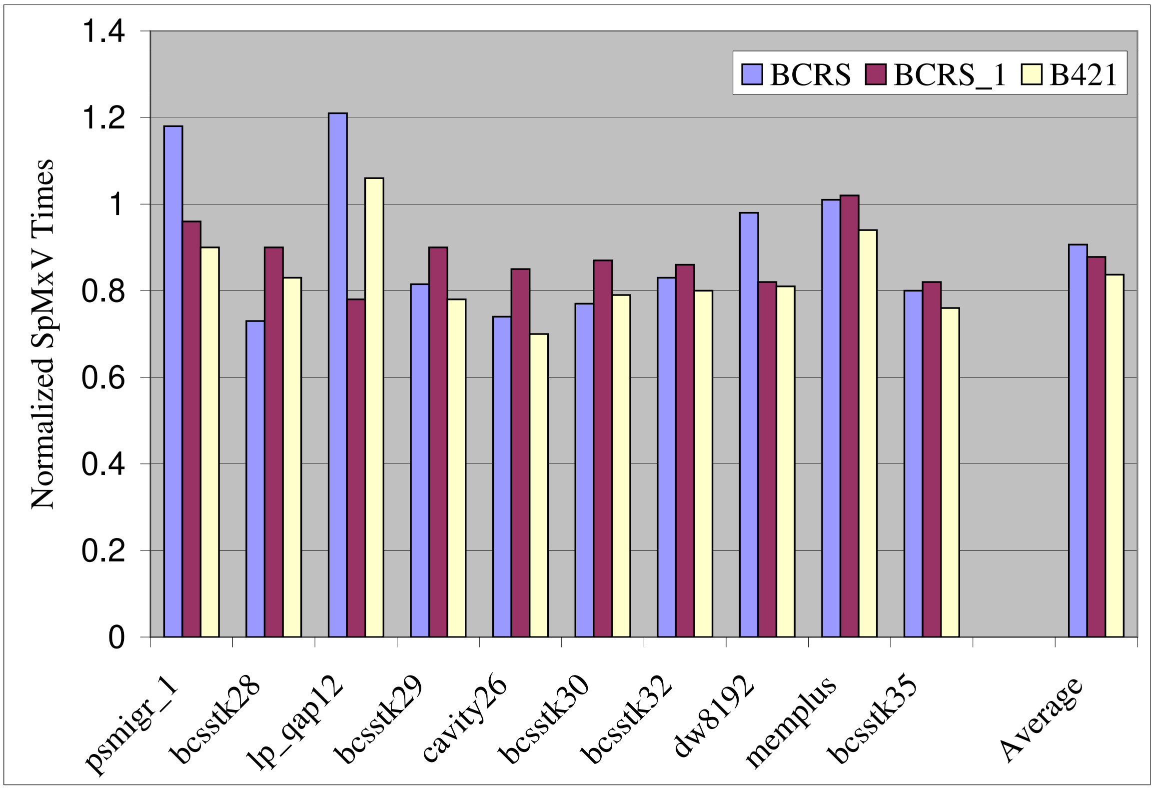


Figure 2.9: Enhancements on BCRS. BCRS_1 handles 1×1 blocks separately, and B421 decomposes matrix into 3 matrices with 1×4,1×2 and 1×1 blocks.

band as a dense matrix requires replacing structural zeros in the band with numerical zeros. This technique trades additional floating-point operations due to structural-to-numerical zero conversion for reduced number of memory indirections, since dense matrix operations do not require the extra load operation. Such a dense matrix can be used as a preconditioner as well, since factorization will be relatively inexpensive.

The effectiveness of this technique depends directly on how dense the band is. This raises the problem of finding a permutation of a matrix with maximum number of nonzeros in the band. For a symmetric matrix, the corresponding graph theoretical problem is as follows.

Given a graph $G = (V, E)$, and bandwidth B, find an ordering π that maximizes

$$\#((u,v) \in E : |\pi(u) - \pi(v)| <= B)$$

By setting $B = 1$, the problem reduces to the Hamiltonian Path problem, and thus is NP-Complete. The combinatorial structure of the problem is close to that of the traveling salesperson problem (TSP). The difference is that in TSP only the edges between successive vertices in the ordering contribute to the objective function, whereas in this problem, all edges between vertices within distance B contribute. But still, the problem is close enough to TSP so that some solution techniques can be adapted.

Diagonal Stripes

This method can be considered as *compressed diagonal storage*. Nonzeros of a matrix can be stored as compressed diagonals instead of rows or columns. Techniques of § 2.1 can be employed to pack nonzero stripes on the diagonals. The disadvantage of this method is that there is no temporal locality on accesses to the left-hand vector y, and each diagonal potentially requires loading all the entries of the x and y vectors. Natural banded structure of the matrices, on the other hand, yields longer stripes.

Ordering rows/columns of the matrix can increase the sizes of dense blocks on the diagonal. For a fixed row/column order permuting columns/rows of the matrix yields a problem similar to that described in § 2.2. We can define a graph in which each column/row is represented by a vertex, and the weight of an edge (u, v) is defined as the number of nonzero pairs that will be in consecutive positions on a diagonal if respective columns/rows are ordered consecutively. With this model, the problem is reduced to finding a maximum weighted path in a graph. On the other hand, permuting rows and columns concurrently is more complicated, and we do not have a reduction to a well-studied problem.

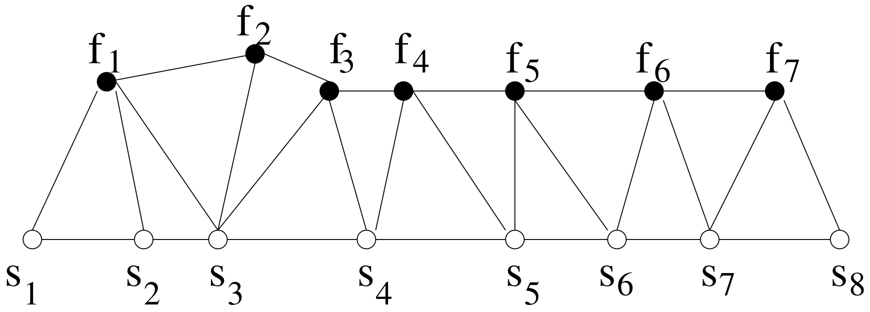


Figure 2.10: Two lines and the edges/triangles between. Closed (open) circles form the first (second) line.

2.4.2 A Special Data Structure for 2D-Meshes

In this section, we will describe an efficient data structure for matrices coming from two-dimensional (2D) finite element meshes. This data structure depends on partitioning a mesh with paths and defining edges between two paths efficiently. Here, the term *line* will be used interchangeably with path. We will use meshes of triangular elements for the purpose of presentation, although our observations are valid for quadrilateral elements as well.

The critical point in the decomposition is the way paths are defined. Take edges on the north boundary of the mesh. The edges themselves and vertices incident to these edges form the first line. By numbering the vertices in the order they appear on the line, we can avoid memory indirections for edges within a line. After the first line is removed, the north boundary of the reduced mesh will give a second line, and all remaining edges of the vertices of the first line will be adjacent to vertices on the second line. The process is illustrated in Figure 2.10. Closed circles correspond to vertices of the first line and open circles correspond to vertices of the second line. Notice that between the two lines is a sequence of triangles, and each triangle has exactly one edge on one of the lines, and two edges between the lines. Consider the leftmost edge between the two lines, and let the two end points be f_1 and s_1. The first triangle either uses one edge from the first line where the triangle is defined by points f_1, f_2 and s_1, or uses an edge from the second triangle where triangle is defined by points f_1, s_1 and s_2. The remaining triangles can be identified in the same way. At any point if the leftmost edge has vertices f_i and s_j, then the next edge will be either (f_i, s_{j+1}) or (f_{i+1}, s_j). So with only one bit of information, we can decode the information for a triangle, thus an edge, whereas a long integer (4 bytes) is required by the conventional scheme.

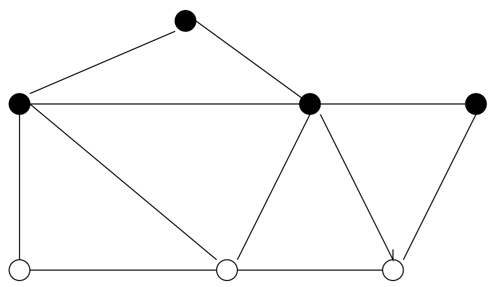


Figure 2.11: An example for triangle in a line

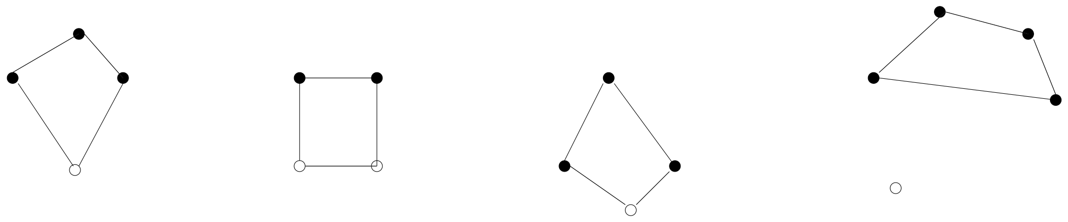


Figure 2.12: Four types of quadrilateral elements. Closed (open) circles form the first (second) line.

Notice that lines might contain triangles, i.e., all three vertices of a triangle might be on the same line, as displayed in Figure 2.11. These triangles can be handled by minor modifications of the basic idea.

The solution can be extended for quadrilateral elements. We start with generating parallel lines as explained in the previous paragraph. Out of the 4 vertices of a quadrilateral element, 1, 2, or 3 might be on the first line, as illustrated in the first three drawings of Figure 2.12, which can be decoded with only two bits. Notice that lines might contain quadrilateral elements as illustrated by the last drawing, but this basic idea still works.

This scheme has advantages of reduced storage and good locality, which typically translates into better memory performance. It is worth noting that we are designing our data structure solely to support matrix-vector multiplication, which requires only a sweep of edges of the graph, and hence less functional data structures are acceptable as long as they provide efficiency for SpMxV. A second point worth mentioning is that even without the proposed storage technique, the vertex numbering scheme can be used to increase sizes of dense blocks to support blocking techniques discussed in § 2.1.

Another application of this decomposition is on stationary iterative methods. These methods are based on relaxation, i.e., eliminating error in one equation/node at a time. At each step, the value corresponding to a node is updated to make the residual for that equation zero. Alternatively, blocks of nodes, instead of a single node, can be relaxed, which typically requires solving a system of linear equations. Clearly, a larger block means a bigger step towards convergence, but an easy-to-solve system is desirable for efficiency as well. It is a common practice to take paths as blocks, for which the corresponding system is trilinear, and thus easy to solve using a direct method.

From a graph theoretical point of view, factoring a matrix can be considered as replacing nodes by a clique of its neighbors, one at a time in a given order. This usually means adding edges to the graph, called *fill* edges. The number of fill edges depends on the order in which vertices are eliminated, thus finding a good ordering is crucial. A graph is *chordal* if there is an order that yields a factor with no fill edges. For instance, a path of vertices is a chordal graph, since eliminating a vertex at one end does not add any edges, and the reduced graph is still a path.

Using the lines described in the previous paragraphs, a graph/matrix can be decomposed into a set of chordal graphs. More specifically, the subgraph defined by two lines and triangle or quadrilateral elements connecting these elements is a chordal graph. For triangles for instance, one node of the leftmost triangle must be adjacent to nothing but nodes of that triangle. Eliminating this node does not add any edges, and the resulting graph has the same characteristics as the first one. The same observations work for quadrilateral elements.

This enables decomposing a graph into a sequence of overlapping chordal graphs. A non-overlapping sequence can be achieved by picking every other such structure (pick the one defined by the first two lines, skip the next defined by second and third lines, pick the next, ...) The corresponding system covers roughly half of the nonzeros of the original matrix and can be solved in linear time in the number of edges using a direct method. In a subsequent phase, chordal graphs skipped in the first phase can be used to cover the remaining nonzeros of the matrix. This gives a promising (or at least potentially interesting) way to solve systems using stationary iterative methods, but its numerical properties need to be investigated.

2.5 Conclusions

In this chapter we investigated methods to improve memory performance of sparse matrix-vector multiplication. The main idea is to exploit regular patterns in the matrix, such as nonzeros in contiguous positions, to reduce the number of load operations. We presented two data structures that block nonzeros in contiguous positions into fixed- or variable-sized blocks, which reduces the number of memory indirections to one for each block instead one for each nonzero. We also presented a matrix reordering method to permute nonzeros of the matrix into contiguous locations to increase the effectiveness of the proposed data structures. Experiments verify the effectiveness of the proposed techniques, with observed performance improvement of up to 33% and an average improvement of 25%. These results motivate further research on improving memory performance of sparse matrix-vector multiplication. For future work, we have discussed alternative methods for better memory performance. The first method exploits natural banded structure of many matrices, and the second method is a special technique designed for two-dimensional meshes.

Chapter 3

One-Dimensional Partitioning of Workload Arrays

This chapter investigates block partitioning of possibly multi-dimensional nonuniform domains over one-dimensional (1D) workload arrays. Communication and synchronization overhead is assumed to be handled implicitly by proper selection of ordering and parallel computation schemes at the beginning, so that load balance is the only metric explicitly considered for decomposition. The load balancing problem in the partitioning can be modeled as the *chains-on-chains partitioning* (CCP) problem with nonnegative task weights and unweighted edges between successive tasks. The objective of the CCP problem is to find a sequence of $P-1$ separators to divide a chain of N tasks with associated computational weights into P consecutive parts so that the *bottleneck value*—load of the maximally loaded part—is minimized.

The first polynomial time algorithm for solving the CCP problem was proposed by Bokhari [12]. Bokhari's $O(N^3P)$-time algorithm is based on finding a minimum path on a layered graph. Nicol and O'Hallaron [71] reduced the complexity to $O(N^2P)$ by decreasing the number of edges in the layered graph. Algorithm paradigms used in following studies can be classified as *dynamic programming* (DP), *iterative refinement*, and *parametric search*. Anily and Federgruen [5] initiated the DP approach with an $O(N^2P)$-time algorithm. Hansen and Lih [42] independently proposed an $O(N^2P)$-time algorithm. Choi and Narahari [16], and Olstad and Manne [73] introduced asymptotically faster $O(NP)$-time, and $O((N-P)P)$-time DP-based algorithms, respectively. The iterative refinement approach starts with a partition and iteratively tries to improve the solution. The $O((N-P)P\log P)$-time algorithm proposed by Manne and Sørevik [65] falls into this class.

The parametric-search approach relies on repeatedly probing for a partition with a *bottleneck* value no greater than a given value. Such a probing takes $\theta(N)$ time since every task has to be examined. Since probing needs to be performed repeatedly, an individual probe can be performed efficiently in $O(P \log N)$-time through binary search, after performing an initial prefix-sum operation in $\theta(N)$-time for task chains with zero communication costs [55]. Later, $O(P \log N)$-time probe algorithms were proposed to handle task chains with nonzero communication costs [52, 53, 54, 72]. Finally, complexity of a probe call was reduced to $O(P \log(N/P))$ by Han, Narahari, and Choi [41].

The parametric-search approach goes back to Iqbal's work [50, 52] describing an ϵ-approximate algorithm, which performs $O(\log(W_{tot}/\epsilon))$ probe calls. Here, W_{tot} denotes the total task weight and ϵ denotes the desired accuracy. Iqbal's algorithm exploits the observation that the bottleneck value is in the range $[W_{tot}/P, W_{tot}]$ and performs binary search in this range by making $O(\log(W_{tot}/\epsilon))$ probes. This work was followed by several exact algorithms involving efficient schemes for searching over bottleneck values by considering only subchain weights. Nicol and O'Hallaron [71, 72] proposed a search scheme that requires at most $4N$ probes. Iqbal and Bokhari [54] relaxed the restriction of this algorithm [71, 72] on bounded task weight and communication cost by proposing a condensation algorithm. Iqbal [51] and Nicol [70, 72] concurrently proposed an efficient search scheme that finds an optimal partition after only $O(P \log N)$ probes.

Asymptotically more efficient algorithms were proposed by Frederickson [32, 33] and Han, Narahari, and Choi [41]. Frederickson proposed an $O(N)$-time optimal algorithm. Han et al. proposed a recursive algorithm with complexity $O(N + P^{1+\varepsilon})$ for any small $\varepsilon > 0$. These two studies have focused on asymptotic complexity, disregarding the practical usefulness.

Despite these efforts, heuristics are still commonly used in the parallel computing community, and design of efficient heuristics is still an active area of research [67]. This attitude depends on ease of implementation, efficiency, and expectation of "good" quality decompositions of heuristics, and misconception that exact CCP algorithms are not affordable as a preprocessing step for efficient parallelization. By contrast, our work proposes efficient exact CCP algorithms. Implementation details and pseudocodes for proposed algorithms are presented for clarity and reproducibility. In order to justify use of the proposed algorithms, we also demonstrate that qualities of the decompositions obtained through heuristics differ substantially from those of optimal ones through experiments on a wide range of real-world problems.

We run an effective heuristic as a preprocessing step to find a "good" upper bound on the optimal bottleneck value. Then we exploit lower and upper bounds on the optimal bottleneck value to restrict the search space for separator-index values. This separator-index bounding scheme is exploited in a static manner in the DP algorithm, drastically reducing the number of table entries computed and referenced. A dynamic separator-index bounding scheme is proposed for parametric search algorithms. This scheme narrows separator-index ranges after each probe. The upper bound on the optimal bottleneck value is also exploited to find a much better initial partition for the iterative-refinement algorithm proposed by Manne and Sørevik [65]. We also propose a different iterative-refinement technique, that is very fast for small-to-medium number of processors. Observations behind this algorithm are further used to incorporate the subchain-weight concept into Iqbal's [50, 52] approximate bisection algorithm to make it an exact algorithm.

Two applications are investigated in our experiments. These are 1D decomposition of irregularly sparse matrices for parallel matrix-vector multiplication (SpMxV), and decomposition for image-space parallel volume rendering. SpMxV is one of the most important kernel operations in scientific computing, and volume rendering is widely used for scientific visualization. Integer and real valued 1D workload arrays arising in these two applications are distinct features. Furthermore, SpMxV, a fine-grain application, demonstrates the feasibility of using optimal load balancing algorithms even in sparse-matrix decomposition. Experiments with proposed CCP algorithms on a wide range of sparse test matrices show that 64-way decompositions can be achieved in 100 times less than a single SpMxV computation time, while producing 4 times better load balance values than the most effective heuristic, on average. Experimental results on volume rendering dataset show that exact algorithms can produce 3.8 times better 64-way decompositions than the most effective heuristic, while being only 11 percent slower, on average.

Most load–balancing models for parallel iterative solvers consider only SpMxV operations. In this work, we also propose a new model that considers both SpMxV and vector operations. The proposed model enables use of CCP algorithms without any additional overhead.

The load balancing problem for heterogeneous systems is also considered. We briefly address modifications required to use the proposed algorithms for heterogeneous systems. Finally, we prove NP-Completeness of the chains partitioning (CP) problem for heterogeneous systems, where processor permutation is allowed in subchain-to-processor assignment.

The organization of this chapter is as follows. § 3.1 defines the CCP problem. A survey of existing CCP algorithms is presented in § 3.2. Proposed CCP algorithms are discussed in § 3.3. Load-balancing applications used in our experiments are described in § 3.4 and performance results are discussed in § 3.5. Discussion of CCP and CP problems for heterogeneous systems is presented in § 3.6.

An earlier version of this work has been published in [77], and a complete version has been submitted as [78].

3.1 Preliminaries

In the CCP problem, a computational problem decomposed into a chain $\mathcal{T}=\langle t_1, t_2, \ldots, t_N\rangle$ of N task/modules with associated *positive* computational weights $\mathcal{W}=\langle w_1, w_2, \ldots, w_N\rangle$ is to be mapped onto a chain $\mathcal{P}=\langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P\rangle$ of P *homogeneous* processors. It is worth noting that there are no precedence constraints among the tasks in the chain. A *subchain* of $\mathcal{T} = \langle t_i, t_{i+1}, \ldots, t_j\rangle$ is defined as a subset of contiguous tasks and is denoted as $\mathcal{T}_{i,j}$. The computational load $W_{i,j}$ of subchain $\mathcal{T}_{i,j}$ is $W_{i,j}=\sum_{h=i}^{j} w_h$. From the contiguity constraint, a partition Π should map contiguous subchains to contiguous processors. Hence, a P-way chain-partition Π_N^P of a task chain $\mathcal{T}$ with N tasks onto a processor chain $\mathcal{P}$ with P processors is described by a sequence $\Pi_N^P=\langle s_0, s_1, s_2, \ldots, s_P\rangle$ of $P+1$ separator indices, where $s_0=0\leq s_1\leq\cdots\leq s_P=N$. Here, s_p denotes the index of the last task of the pth part, so that $\mathcal{P}_p$ gets the subchain $\mathcal{T}_{s_{p-1}+1,s_p}$ with load $L_p=W_{s_{p-1}+1,s_p}$. The cost $C(\Pi)$ of a partition Π is determined by the maximum processor execution time among all processors, i.e., $C(\Pi)=B=\max_{1\leq p\leq P}\{L_p\}$. This B value of a partition is called its *bottleneck* value, and the processor/part defining it is called the *bottleneck processor/part*. The CCP problem can be defined as finding a mapping Π_{opt} that minimizes the bottleneck value $B_{opt}=C(\Pi_{opt})$.

3.2 Previous Work on the CCP Problem

Each CCP algorithm discussed in this section and § 3.3 involves an initial prefix-sum operation on the task-weight array $\mathcal{W}$ to enhance the efficiency of subsequent subchain-weight computations. The prefix-sum operation replaces the ith entry $\mathcal{W}[i]$ with the sum of the first i entries ($\sum_{h=1}^{i} w_h$), so that computational load of a subchain W_{ij} can be efficiently determined as $\mathcal{W}[j]-\mathcal{W}[i]$ in $O(1)$-time. In our discussions, $\mathcal{W}$ is used to refer to the prefix-summed $\mathcal{W}$-array, and the cost of this initial prefix-sum operation is considered in the complexity analysis. The presentations focus only on finding the bottleneck value B_{opt}, since a corresponding optimal mapping can be constructed easily by making a *PROBE*(B_{opt}) call as discussed in § 3.2.4.

3.2.1 Heuristics

The most commonly used partitioning heuristic is based on *recursive bisection* (RB). RB achieves P-way partitioning through $\log P$ bisection levels, where P is a power of 2. At each bisection step in a level, the current chain is divided evenly into two subchains. Although optimal division can be easily achieved at every bisection step, the sequence of optimal bisections may lead to poor load balancing. RB can be efficiently implemented in $O(N+P\log N)$ time by first performing a prefix-sum operation on the workload array $\mathcal{W}$, with complexity $O(N)$, and then making $P-1$ binary searches in the prefix-summed $\mathcal{W}$-array, each with complexity $O(\log N)$.

Miguet and Pierson [67] recently proposed two other heuristics. The first heuristic (H1) computes the separator values such that s_p is the largest index such that $W_{s_p} \le pB^*$, where $B^*=W_{tot}/P$ is the ideal bottleneck value, and $W_{tot}=\sum_{i=1}^{N} w_i$ denotes sum of all task weights. The second heuristic (H2) further refines the separator indices by incrementing each s_p value found in H1 if $(W_{s_{p+1}}-pB^*)<(pB^*-W_{s_p})$. These two heuristics can also be implemented in $O(N+P\log N)$ time by performing $P-1$ binary searches in the prefix-summed $\mathcal{W}$-array.

Miguet and Pierson [67] have already proved the upper bounds on the bottleneck values of the partitions found by H1 and H2 as $B_{H1}, B_{H2} < B^*+w_{max}$, where $w_{max}=\max_{1\le p\le N}\{w_i\}$ denotes the maximum task weight. The following lemma establishes a similar bound for the RB heuristic.

Lemma 3.2.1 *Let $\Pi_{RB}=\langle s_0, s_1, \ldots, s_P\rangle$ be a partition constructed by the RB heuristic for a given CCP problem instance $(\mathcal{W}, N, P)$. Then $B_{RB}=C(\Pi_{RB})$ satisfies $B_{RB}\le B^*+w_{max}(P-1)/P$.*

Proof: Consider the first bisection step. There exists a pivot index $1\le i_1\le N$ such that both sides weigh less than $W_{tot}/2$ without the i_1th task, and more than $W_{tot}/2$ with it.

$$W_{1,i_1-1}, W_{i_1+1,N} \le W_{tot}/2 \le W_{1,i_1}, W_{i_1,N}$$

The worst case for RB occurs when $w_{i_1} = w_{max}$ and $W_{1,i_1-1} = W_{i_1+1,N} = (W_{tot} - w_{max})/2$. Without loss of generality, assume that t_{i_1} is assigned to the left part so that $s_{P/2} = i_1$ and $W_{1,s_{P/2}} = W_{tot}/2 + w_{max}/2$. In a similar worst-case bisection of $\mathcal{T}_{1,s_{P/2}}$, there exists an index i_2 such that $w_{i_2} = w_{max}$ and $W_{1,i_2-1} = W_{i_2+1,s_{P/2}} = (W_{tot} - w_{max})/4$, and t_{i_2} is assigned to the left part so that $s_{P/4} = i_2$ and $W_{1,s_{P/4}} = (W_{tot} - w_{max})/4 + w_{max} = W_{tot}/4 + (3/4)w_{max}$. For a sequence of $\log P$ such worst case bisection steps on the left parts, processor $\mathcal{P}_1$ will be the bottleneck processor with load $B_{RB} = W_{1,s_1} = W_{tot}/P + w_{max}(P-1)/P$. $\square$

3.2.2 Dynamic Programming

The *overlapping* subproblem space can be defined as $\mathcal{T}_i^p$, for $p = 1, 2, \ldots, P$ and $i = p, p+1, \ldots, N-P+p$, where $\mathcal{T}_i^p$ denotes a p-way CCP of prefix task-subchain $\mathcal{T}_{1,i} = \langle t_1, t_2, \ldots, t_i \rangle$ onto prefix processor-subchain $\mathcal{P}_{1,p} = \langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_p \rangle$. Notice that index i is restricted to $p \leq i \leq N - P + p$ range because there is no merit in leaving a processor empty. From this subproblem space definition, the *optimal substructure* property of the CCP problem can be shown by considering an optimal mapping $\Pi_i^p = \langle s_0, s_1, \ldots, s_p = i \rangle$ with a bottleneck value B_i^p for the CCP subproblem $\mathcal{T}_i^p$. If the last processor is not the bottleneck processor in Π_i^p, then $\Pi_{s_{p-1}}^{p-1} = \langle s_0, s_1, \ldots, s_{p-1} \rangle$ should be an optimal mapping for the subproblem $\mathcal{T}_{s_{p-1}}^{p-1}$. Hence, recursive definition for the bottleneck value of an optimal mapping is

$$B_i^p = \min_{p-1 \leq j < i} \left\{ \max\{B_j^{p-1}, W_{j+1,i}\} \right\}. \tag{3.1}$$

In (3.1), searching for index j corresponds to searching for separator s_{p-1} so that the remaining subchain $\mathcal{T}_{j+1,i}$ is assigned to the last processor $\mathcal{P}_p$ in an optimal mapping Π_i^p of $\mathcal{T}_i^p$. The bottleneck value B_N^P of an optimal mapping can be computed using (3.1) in a bottom-up fashion starting from $B_i^1 = W_{1,i}$ for $i = 1, 2, \ldots, N$. An initial prefix-sum on the workload array $\mathcal{W}$ enables constant-time computation of subchain weight of the form $W_{j+1,i}$ through $W_{j+1,i} = \mathcal{W}[i] - \mathcal{W}[j]$. Computing B_i^p using (3.1) takes $O(N-p)$ time for each i and p, and thus the algorithm takes $O((N-P)^2 P)$ time since the number of distinct subproblems is equal to $(N-P+1)P$.

Choi and Narahari [16], and Olstad and Manne [73] reduced the complexity of this scheme to $O(NP)$ and $O((N-P)P)$, respectively, by exploiting the following observations that hold for positive task weights. For a fixed p in (3.1), the minimum index value j_i^p defining B_i^p cannot occur

```
DP (𝒲, N, P)
  B[1, i] ← 𝒲[i]  for  i = 1, 2, ..., N;
  for p ← 2 to P do
    j ← p − 1;
    for i ← p to N − P + p do
      if 𝒲[i] − 𝒲[j] > B[p−1, j] then
        repeat j ← j + 1 until 𝒲[i] − 𝒲[j] ≤ B[p−1, j];
        if 𝒲[i] − 𝒲[j−1] < B[p−1, j] then
          j ← j − 1;
          B[p, i] ← 𝒲[i] − 𝒲[j];
        else
          B[p, i] ← B[p−1, j];
      else
        B[p, i] ← B[p−1, j];
  return B_opt ← B[P, N];
```

Figure 3.1: $O((N-P)P)$-time dynamic-programming algorithm proposed by Choi and Narahari [6], and Olstad and Manne [34].

at a value less than the minimum index value j_{i-1}^p defining B_{i-1}^p, i.e., $j_{i-1}^p \leq j_i^p \leq (i-1)$. Hence, the search for the optimal j_i^p can start from j_{i-1}^p. In (3.1), B_j^{p-1} for a fixed p is a nondecreasing function of j, and $W_{j+1,i}$ for a fixed i is a decreasing function of j, reducing to 0 at $j=i$. Thus, two cases occur in a semi-closed interval $[j_{i-1}^p, i)$ for j. If $W_{j+1,i} > B_j^{p-1}$ initially, then these two functions intersect in $[j_{i-1}^p, i)$. In this case, the search for j_i^p continues until $W_{j^*+1,i} \leq B_j^{p-1}$ and then only j^* and j^*-1 are considered for setting j_i^p with $j_i^p = j^*$ if $B_{j^*}^{p-1} \leq W_{j,i}$ and $j_i^p = j^*-1$ otherwise. Note that this scheme automatically detects $j_i^p = i-1$ if $W_{j+1,i}$ and B_j^{p-1} intersect in the open interval $(i-1, i)$. However if, $W_{j+1,i} \leq B_j^{p-1}$ initially, then B_j^{p-1} lies above $W_{j+1,i}$ in the closed interval $[j_{i-1}^p, i]$. In this case, the minimum value occurs at the first value of j, i.e., $j_i^p = j_{i-1}^p$. These improvements lead to an $O((N-P)P)$-time algorithm since computation of all B_i^p values for a fixed p makes $O(N-P)$ references to already computed B_j^{p-1} values. Figure 3.1 displays a run-time efficient implementation of this $O((N-P)P)$-time DP algorithm, that avoids the explicit min-max operation required in (3.1). In Figure 3.1, B_i^p values are stored in a table whose entries are computed in row–major order.

3.2.3 Iterative Refinement

The algorithm proposed by Manne and Sorevik [65], referred to here as the MS algorithm, is based on finding a sequence of non-optimal partitions such that there is only one way each partition can be improved. For this purpose, they introduce the *leftist partition (LP)*. Consider a partition Π such that $\mathcal{P}_p$ is the leftmost processor containing at least two tasks. Π is defined as an LP if

```
MS (W, N, P)
  s_p ← p  for  p ← 0, 1, ..., P−1;   s_P ← N;
  L_p ← w_p  for  p ← 1, ..., P−1;   L_P ← Σ_{p=P}^{N} w_p;
  repeat
    b ← { p | L_p is maximum over 1 ≤ p ≤ P};
    B ← L_b;
    if s_{b−1} + 1 = s_b then
      exit the repeat-loop;
    p ← b;
    while L_p ≥ B and p > 1 do
      s_{p−1} ← s_{p−1} + 1;
      L_p ← L_p − w_{s_{p−1}};
      L_{p−1} ← L_{p−1} + w_{s_{p−1}};
      if L_p ≤ B then
        p ← p − 1;
  until L_1 ≥ B;
  return B_opt ← B;
```

Figure 3.2: Iterative refinement algorithm proposed by Manne and Sorevik [26].

increasing the load of any processor $\mathcal{P}_\ell$ that lies to the right of $\mathcal{P}_p$ by augmenting the last task of $\mathcal{P}_{\ell-1}$ to $\mathcal{P}_\ell$ makes $\mathcal{P}_\ell$ a bottleneck processor with a load $\geq C(\Pi)$.

Let Π be an LP with bottleneck processor $\mathcal{P}_b$ and bottleneck value $C(\Pi) = B = L_b$. If $\mathcal{P}_b$ contains only one task, then Π is optimal. On the other hand, assume that $\mathcal{P}_b$ contains at least two tasks. The refinement step, which is shown by the inner while-loop in Figure 3.2, tries to find a new LP of lower cost by successively removing the first task of $\mathcal{P}_p$ and augmenting it to $\mathcal{P}_{p-1}$ for $p = b, b-1, \ldots,$ until $L_p < B$. Unsuccessful refinement occurs when the while-loop proceeds until $p = 1$ with $L_p \geq B$. Manne and Sorevik [65] proved that a successful refinement of an LP gives a new LP and that LP is optimal if the refinement is unsuccessful. Thus, an initial LP is needed to start the algorithm. As shown in Figure 3.2, Manne and Sorevik [65] proposed using an initial LP where the $P-1$ leftmost processors each have only one task and last processor contains the rest of the tasks.

The MS algorithm moves each separator index at most $N-P$ times, so that the total number of separator-index moves is $O(P(N-P))$. A max-heap is maintained for the processor loads to speed up the operation of finding a bottleneck processor at the beginning of each repeat-loop iteration. The cost of each separator-index move is $O(\log P)$ since it necessitates one decrease-key and one increase-key operations. Thus the complexity of the MS algorithm is $O(P(N-P)\log P)$.

```
PROBE (B)
  s_0 ← 0;   p ← 1;
  Bsum ← B;
  while p ≤ P and Bsum < W_tot do
    s_p ← BINSRCH (W, s_{p-1}+1, N, Bsum);
    Bsum ← W[s_p] + B;
    p ← p + 1;
  if Bsum ≥ W_tot then
    return TRUE;
  else
    return FALSE;
```

(a)

```
PROBE (B)
  s_0 ← 0;   p ← 1;   step ← N/K;
  Bsum ← B;
  while p ≤ P and Bsum < W_tot do
    while W[step] < Bsum do
      step ← step + N/P;
    s_p ← BINSRCH (W, step−N/K, step, Bsum);
    Bsum ← W[s_p] + B;
    p ← p + 1;
  if Bsum ≥ W_tot then
    return TRUE;
  else
    return FALSE;
```

(b)

Figure 3.3: (a) Standard probe algorithm with $O(P \log N)$ complexity, (b) $O(P \log(N/P))$-time probe algorithm proposed by Han, Narahari, and Choi [12].

3.2.4 Parametric Search

The parametric-search approach relies on repeated probing for a partition Π with a bottleneck value no greater than a given B value, i.e., $C(\Pi) \leq B$. Probe algorithms exploit the *greedy-choice* property for existence and construction of Π. The greedy choice here is to minimize remaining work after loading processor $\mathcal{P}_p$ subject to $L_p \leq B$ for $p = 1, \ldots, P-1$ in order. *PROBE*(B) functions given in Figure 3.3 exploit this greedy property as follows. *PROBE* finds the largest index s_1 so that $W_{1,s_1} \leq B$, and assigns subchain $\mathcal{T}_{1,s_1}$ to processor $\mathcal{P}_1$ with load $L_1 = W_{1,s_1}$. Hence, the first task in the second processor is t_{s_1+1}. *PROBE* then similarly finds the largest index s_2 so that $W_{s_1+1,s_2} \leq B$, and assigns the subchain $\mathcal{T}_{s_1+1,s_2}$ to processor $\mathcal{P}_2$ with load $L_1 = W_{s_1+1,s_2}$. This process continues until either all tasks are assigned or all processors are exhausted. The former and latter cases indicate feasibility and infeasibility of B, respectively.

Figure 3.3(a) illustrates the standard probe algorithm. As seen in Figure 3.3, the indices $s_1, s_2, \ldots s_{P-1}$ are efficiently found through binary search (*BINSRCH*) on the prefix-summed $\mathcal{W}$-array. In this figure, *BINSRCH*$(\mathcal{W}, i, N, Bsum)$ searches $\mathcal{W}$ in the index range $[i, N]$ to compute the index $i \leq j \leq N$ such that $\mathcal{W}[j] \leq Bsum$ and $\mathcal{W}[j+1] > Bsum$. The complexity of the standard probe algorithm is $\sum_{p=0}^{P-1} \theta(\log(N - s_p)) = O(P \log N)$. Han, Narahari and Choi [41] proposed an $O(P \log N/P)$-time probe algorithm (see Figure 3.3(b)) exploiting P repeated binary searches on the same $\mathcal{W}$-array with increasing search values. Their algorithm divides the chain into P subchains of equal length. At each probe, a linear search is performed on the weights of the last tasks of

```
ε-BISECT (𝒲, N, P, ε)
   LB ← B*;   UB ← W_tot;
   repeat
      B ← (UB + LB) / 2;
      if PROBE (B) then
         UB ← B;
      else
         LB ← B;
   until UB ≤ LB+ ε;
   return B_opt ← UB;
```

Figure 3.4: Bisection as an ϵ-approximation algorithm.

these P subchains to find out in which subchain the search value could be, and then binary search is performed on the respective subchain of length N/P. Note that since the probe search values always increase, linear search can be performed incrementally, that is, search continues from the last subchain that was searched to the right. Thus, the total time for P searches on P numbers is only $O(P)$. This means a total cost of $O(P\log(N/P))$ for P binary searches, thus for the probe function.

Bisection as an Approximation Algorithm

Let $f(B)$ be the binary-valued function where $f(B) = 1$ if *PROBE(B)* is true and $f(B) = 0$ if *PROBE(B)* is false. Clearly, $f(B)$ is nondecreasing in B, and B_{opt} lies between $LB = B^* = W_{tot}/P$ and $UB = W_{tot}$. These observations are exploited in the *bisection* algorithm leading to an efficient ϵ-approximate algorithm, where ϵ is the desired precision. The interval $[W_{tot}/P, W_{tot}]$ is conceptually discretized into $(W_{tot} - W_{tot}/P)/\epsilon$ bottleneck values, and binary search is used in this range to find the minimum feasible bottleneck value B_{opt}. The bisection algorithm, as illustrated in Figure 3.4, performs $O(\log(W_{tot}/\epsilon))$ *PROBE* calls, and each *PROBE* call costs $O(P\log(N/P))$. Hence, the bisection algorithm runs in $O(N + P\log(N/P)\log(W_{tot}/\epsilon))$ time, where $O(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$. The performance of this algorithm deteriorates when $\log(W_{tot}/\epsilon)$ is comparable with N.

Nicol's Algorithm

Nicol's algorithm [70] exploits the fact that any candidate B value is equal to weight $W_{i,j}$ of a subchain. A naive solution is to generate all subchain weights of the form $W_{i,j}$, sort them, and then use binary search to find the minimum $W_{a,b}$ value for which $PROBE(W_{a,b}) = TRUE$. Nicol's algorithm efficiently searches for the earliest range $W_{a,b}$ for which $B_{opt} = W_{a,b}$ by considering each processor in order as a candidate bottleneck processor in an optimal mapping. Let Π_{opt} be the optimal mapping constructed by greedy *PROBE*(B_{opt}), and let processor $\mathcal{P}_b$ be the first bottleneck processor with load $L_b = W_{s_{b-1}+1,s_b} = B_{opt}$ in $\Pi_{opt} = \langle s_0, s_1, \ldots, s_b, \ldots, s_P \rangle$. Under these assumptions, this greedy construction of Π_{opt} ensures that each processor $\mathcal{P}_p$ preceding $\mathcal{P}_b$ is loaded as much as possible with $L_p < B_{opt}$, for $p = 1, 2, \ldots, b-1$ in Π_{opt}. Here, *PROBE*(L_p) = *FALSE* since $L_p < B_{opt}$, and *PROBE*($L_p + w_{s_p+1}$) = *TRUE* since adding one more task to processor $\mathcal{P}_p$ increases its load to $L_p + w_{s_p+1} > B_{opt}$. Hence, if $b = 1$ then s_1 is equal to the smallest index i_1 such that *PROBE*(W_{1,i_1}) = *TRUE*, and $B_{opt} = B_1 = W_{1,s_1}$. However, if $b > 1$, then because of the greedy choice property $\mathcal{P}_1$ should be loaded as much as possible without exceeding $B_{opt} = B_b < B_1$, which implies that $s_1 = i_1 - 1$, and hence $L_1 = W_{1,i_1-1}$. If $b = 2$, then s_2 is equal to the smallest index i_2 such that *PROBE*(W_{i_1,i_2}) = *TRUE*, and $B_{opt} = B_2 = W_{i_1,i_2}$. If $b > 2$, then $s_2 = i_2 - 1$. We iterate for $b = 1, 2, \ldots, P-1$, computing i_b as the smallest index for which *PROBE*(W_{i_{b-1},i_b}) = *TRUE* and save $B_b = W_{i_{b-1},i_b}$ with $i_P = N$. Finally, the optimal bottleneck value is selected as $B_{opt} = \min_{1 \le b \le P} B_b$.

Figure 3.5 illustrates Nicol's algorithm. As seen in this figure, given i_{b-1}, i_b is found by performing a binary search over all subchain weights of the form $W_{i_{b-1},j}$, for $i_{b-1} \le j \le N$, in the bth iteration of the *for-loop*. Hence, Nicol's algorithm performs $O(\log N)$ *PROBE* calls to find i_b at iteration b, and each *PROBE* call costs $O(P \log(N/P))$. Thus, the cost of computing an individual B_b value is $O(P \log N \log(N/P))$. Since P–1 such B_b values are computed, the overall complexity of Nicol's algorithm is $O(N + P^2 \log N \log(N/P))$, where $O(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$.

Two possible implementations of Nicol's algorithm are presented in Figure 3.5. Figure 3.5(a) illustrates a straightforward implementation, whereas Figure 3.5(b) illustrates a careful implementation, which maintains and uses the information from previous probes to answer without calling the *PROBE* function. As seen in Figure 3.5(b), this information is efficiently maintained as an undetermined bottleneck-value range (LB, UB), which is dynamically refined after each probe. Any bottleneck value encountered outside the current range is immediately accepted or rejected. Although this simple scheme does not improve the asymptotic complexity of the algorithm, it drastically reduces the number of probes, as will be discussed in § 3.5.

```
NICOL- (W, N, P)
  i_0 ← 1;
  for b ← 1 to P − 1 do
    ilow ← i_{b−1};   ihigh ← N;
    while ilow < ihigh do
      imid ← (ilow + ihigh) / 2;
      B ← W[imid] − W[i_{b−1}−1];
      if PROBE (B) then
        ihigh ← imid;
      else
        ilow ← imid + 1;
    i_b ← ihigh;
    B_b ← W[i_b] − W[i_{b−1}−1];
  B_P ← W[N] − W[i_{P−1}−1];
  return B_opt ← min_{1≤b≤P}{B_p};
```

(a)

```
NICOL (W, N, P)
  i_0 ← 1;   LB ← B*;   UB ← W_tot;
  for b ← 1 to P − 1 do
    ilow ← i_{b−1};   ihigh ← N;
    while ilow < ihigh do
      imid ← (ilow + ihigh) / 2;
      B ← W[imid] − W[i_{b−1}−1];
      if LB ≤ B < UB then
        if PROBE (B) then
          ihigh ← imid;
          UB ← B;
        else
          ilow ← imid + 1;
          LB ← B;
      elseif B ≥ UB
        ihigh ← imid;
      else
        ilow ← imid + 1;
    i_b ← ihigh;
    B_b ← W[i_b] − W[i_{b−1}−1];
  B_P ← W[N] − W[i_{P−1}−1];
  return B_opt ← min_{1≤b≤P}{B_p};
```

(b)

Figure 3.5: Nicol's [33] algorithm: (a) straightforward implementation, (b) careful implementation with dynamic bottleneck-value bounding.

3.3 Proposed CCP Algorithms

In this section we present our proposed algorithms for the CCP problem.

3.3.1 Restricting the Search Space

Our proposed CCP algorithms exploit lower and upper bounds on the optimal bottleneck value to restrict the search space for separator values as a preprocessing step. Natural lower and upper bounds for the optimal bottleneck value B_{opt} of a given CCP problem instance $(\mathcal{W}, N, P)$ are $LB = \max\{B^* = W_{tot}/P, w_{max}\}$ and $UB = B^* + w_{max}$, respectively. However, $w_{max} < B^*$ in coarse grain parallelization ($P \ll N$) of most real-world applications. Our presentation will be for $w_{max} < B^* = LB$ even though all findings will be valid when B^* is replaced with $\max\{B^*, w_{max}\}$. The following lemma describes how to use these natural bounds on B_{opt} to restrict the search space for the separator values.

Lemma 3.3.1 *For a given CCP problem instance $(\mathcal{W}, N, P)$, if B_f is a feasible bottleneck value in the range $[B^*, B^* + w_{max}]$, then there exists a partition $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ of cost $C(\Pi) \le B_f$ with $SL_p \le s_p \le SH_p$, for $1 \le p < P$, where SL_p and SH_p are, respectively, the smallest and largest* *indices such that*

$$W_{1,SL_p} \;\geq\; p\,(B^* - w_{max}(P-p)/P) \quad \textit{and} \quad W_{1,SH_p} \;\leq\; p\,(B^* + w_{max}(P-p)/P).$$

Proof: Let $B_f = B^*{+}w$, where $0 \leq w < w_{max}$. Partition Π can be constructed by *PROBE(B)*, which loads the first p processors as much as possible subject to $L_q \leq B_f$, for $q = 1, 2, \ldots, p$. In the worst case, $w_{s_p+1} = w_{max}$ for each of the first p processors. Thus, we have $W_{1,s_p} \geq f(w) = p(B^*{+}w{-}w_{max})$ for $p = 1, 2, \ldots, P-1$. However, it should be possible to divide the remaining subchain $\mathcal{T}_{s_p+1,N}$ into $P-p$ parts without exceeding B_f, i.e., $W_{s_{p+1},N} \leq (P-p)\,(B^*+w)$. Thus, we also have $W_{1,s_p} \geq g(w) = W_{tot} - (P{-}p)\,(B^*{+}w)$. Note that $f(w)$ is an increasing function of w, whereas $g(w)$ is a decreasing function of w. The minimum of $\max\{f(w), g(w)\}$ is at the intersection of $f(w)$ and $g(w)$, so that $W_{1,s_p} \geq p\,(B^* - w_{max}(P{-}p)/P)$.

To prove the upper bounds, we can start with $W_{1,s_p} \leq f(w) = p(B^* + w)$, which holds when $L_q = B^* + w$ for $q = 1, \ldots, p$. The condition $W_{s_p+1,N} \geq (P-p)(B^*{+}w - w_{max})$ however, ensures feasibility of $B_f = B^*{+}w$, since *PROBE(B)* can always load each of the remaining $(P-p)$ processors with $B^*{+}w - w_{max}$. Thus, we also have $W_{1,s_p} \leq g(w) = W_{tot} - (P-p)(B^*{+}w - w_{max})$. Here, $f(w)$ is an increasing function of w, whereas $g(w)$ is a decreasing function of w, which yields $W_{1,s_p} \leq p\,(B^*{+}w_{max}(P-p)/P)$. □

Corollary 3.3.2 *The separator range weights are* $\Delta W_p = \sum_{i=SL_p}^{SH_p} w_i = W_{1,SH_p} – W_{1,SL_p} = 2\,w_{max}\,p\,(P – p)/P$ *with a maximum value* $Pw_{max}/2$ *at* $p = P/2$.

Applying this corollary requires finding w_{max}, which entails an overhead equivalent to that of the prefix-sum operation, and hence should be avoided. In this work, we adopt a practical scheme to construct the bounds on separator indices. We run the RB heuristic to find a hopefully good bottleneck value B_{RB}, and use B_{RB} as an upper bound for bottleneck values, i.e., $UB = B_{RB}$. Then we run *LR-PROBE*(B_{RB}) and *RL-PROBE*(B_{RB}) to construct two mappings $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_P^1 \rangle$ and $\Pi^2 = \langle \ell_0^2, \ell_1^2, \ldots, \ell_P^2 \rangle$ with $C(\Pi^1), C(\Pi^2) \leq B_{RB}$. Here, *LR-PROBE* denotes the left-to-right probe given in Figure 3.3, whereas *RL-PROBE* denotes a right-to-left probe function, which can be considered as the dual of the *LR-PROBE*. *RL-PROBE* exploits the greedy-choice property from right to left. That is, *RL-PROBE* assigns subchains from the right end towards the left end of the task chain to processors in the order $\mathcal{P}_P, \mathcal{P}_{P-1}, \ldots, \mathcal{P}_1$. From these two mappings, lower and upper bound values for s_p separator indices are constructed as $SL_p = \ell_p^2$ and $SH_p = h_p^1$, respectively. These bounds are further refined by running *LR-PROBE*(B^*) and *RL-PROBE*(B^*) to construct two mappings $\Pi^3 = \langle \ell_0^3, \ell_1^3, \ldots, \ell_P^3 \rangle$ and $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_P^4 \rangle$, and then defining $SL_p = \max\{SL_p, \ell_p^3\}$ and $SH_p = \min\{SH_p, h_p^4\}$ for $1 \leq p < P$. Lemmas 3.3.3 and 3.3.4 prove correctness of these bounds.

Lemma 3.3.3 *For a given CCP problem instance $(\mathcal{W}, N, P)$ and a feasible bottleneck value B_f, let $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_P^1 \rangle$ and $\Pi^2 = \langle \ell_0^2, \ell_1^2, \ldots, \ell_P^2 \rangle$ be the partitions constructed by LR-PROBE(B_f) and RL-PROBE(B_f), respectively. Then any partition $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ of cost $C(\Pi) = B \leq B_f$ satisfies $\ell_p^2 \leq s_p \leq h_p^1$.*

Proof: By the property of *LR-PROBE*(B_f), h_p^1 is the largest index such that $\mathcal{T}_{1,h_p^1}$ can be partitioned into p parts without exceeding B_f. If $s_p > h_p^1$, then the bottleneck value will exceed B_f and thus B. By the property of *RL-PROBE*(B_f), ℓ_p^2 is the smallest index where $\mathcal{T}_{\ell_p^2,N}$ can be partitioned into $P-p$ parts without exceeding B_f. If $s_p < \ell_p^2$, then the bottleneck value will exceed B_f and thus B. $\square$

Lemma 3.3.4 *For a given CCP problem instance $(\mathcal{W}, N, P)$, let $\Pi^3 = \langle \ell_0^3, \ell_1^3, \ldots, \ell_P^3 \rangle$ and $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_P^4 \rangle$ be the partitions constructed by LR-PROBE(B^*) and RL-PROBE(B^*), respectively. Then for any feasible bottleneck value B_f, there exists a partition $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ of cost $C(\Pi) \leq B_f$ that satisfies $\ell_p^3 \leq s_p \leq h_p^4$.*

Proof: Consider the partition $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ constructed by *LR-PROBE*(B_f). It is clear that this partition already satisfies the lower bounds, i.e., $s_p \geq \ell_p^3$. Assume $s_p > h_p^4$, then partition Π' obtained by moving s_p back to h_p^4 also yields a partition with cost $C(\Pi') \leq B_f$, since $\mathcal{T}_{h_p^4+1,N}$ can be partitioned into $P-p$ parts without exceeding B^*. $\square$

The difference between Lemmas 3.3.3 and 3.3.4 is that the former ensures the existence of all partitions with cost $\leq B_f$ within the given separator-index ranges, whereas the latter ensures only the existence of at least one such partition within the given ranges. The following corollary combines the results of these two lemmas.

Corollary 3.3.5 *For a given CCP problem instance $(\mathcal{W}, N, P)$ and a feasible bottleneck value B_f, let $\Pi^1 = \langle h_0^1, h_1^1, \ldots, h_P^1 \rangle$, $\Pi^2 = \langle \ell_0^2, \ell_1^2, \ldots, \ell_P^2 \rangle$, $\Pi^3 = \langle \ell_0^3, \ell_1^3, \ldots, \ell_P^3 \rangle$, and $\Pi^4 = \langle h_0^4, h_1^4, \ldots, h_P^4 \rangle$ be the* *partitions constructed by LR-PROBE(B_f), RL-PROBE(B_f), LR-PROBE(B^*), and RL-PROBE(B^*), respectively. Then for any feasible bottleneck value B in the range $[B^*, B_f]$, there exists a partition $\Pi = \langle s_0, s_1, \ldots, s_P \rangle$ of cost $C(\Pi) \leq B$ with $SL_p \leq s_p \leq SH_p$, for $1 \leq p < P$, where $SL_p = \max\{\ell_p^2, \ell_p^3\}$ and $SH_p = \min\{h_p^1, h_p^4\}$.*

Corollary 3.3.6 *The separator range weights become $\Delta W_p = 2\min\{p, (P-p)\}w_{max}$ in the worst case, with a maximum value Pw_{max} at $p = P/2$.*

Lemma 3.3.1 and Corollary 3.3.5 directly infer the following theorem since $B^* \leq B_{opt} \leq B^* + w_{max}$.

Theorem 3.3.7 *For a given CCP problem instance $(\mathcal{W}, N, P)$, and SL_p and SH_p index bounds constructed according to Lemma 3.3.1 or Corollary 3.3.5, there exists an optimal partition $\Pi_{opt} = \langle s_0, s_1, \ldots, s_P \rangle$ with $SL_p \leq s_p \leq SH_p$, for $1 \leq p < P$.*

Comparison of separator range weights in Lemma 3.3.1 and Corollary 3.3.5 shows that separator range weights produced by the practical scheme described in Corollary 3.3.5 may be worse than those of Lemma 3.3.1 by a factor of two. This is only the worst-case behavior however, and the practical scheme normally finds much better bounds, since order in the chain usually prevents the worst-case behavior and $B_{RB} < B^* + w_{max}$. Experimental results in § 3.5 justify this expectation.

Complexity Analysis Models

Corollaries 3.3.2 and 3.3.6 give bounds on the weights of the separator-index ranges. However, we need bounds on the sizes of these separator-index ranges for computational complexity analysis of the proposed CCP algorithms. Here, the size $\Delta S_p = SH_p - SL_p + 1$ denotes the number of tasks within the pth range $[SL_p, SH_p]$. Miguet and Pierson [67] propose the model $w_i = \theta(w_{avg})$ for $i = 1, 2, \ldots, N$ to prove that their H1 and H2 heuristics allocate $\theta(N/P)$ tasks to each processor. Here, $w_{avg} = W_{tot}/N$ denotes the average task weight. This assumption means that the weight of each task is not too far away from the average task weight. Using Corollaries 3.3.2 and 3.3.5, this model induces $\Delta S_p = O(P\, w_{max}/w_{avg})$. Moreover, this model can be exploited to induce the optimistic bound $\Delta S_p = O(P)$. However, we find their model too restrictive, since the minimum and maximum task weights can deviate substantially from w_{avg}. Here, we establish a looser and more realistic model on task weights, so that for any subchain $\mathcal{T}_{i,j}$ with weight $W_{i,j}$ sufficiently larger

```
DP+ (𝒲, N, P, SL, SH)
  B[1,i] ← 𝒲[i]  for  i = SL_1, SL_1 + 1, ..., SH_1;
  B[1, SH_1+1] ← ∞;
  for p ← 2 to P do
    j ← SL_{p-1};
    for i ← SL_p to SH_p do
      if 𝒲[i] − 𝒲[j] > B[p−1, j] then
        repeat j ← j + 1 until 𝒲[i] − 𝒲[j] ≤ B[p−1, j];
        if 𝒲[i] − 𝒲[j−1] < B[p−1, j] then
          j ← j − 1;
          B[p,i] ← 𝒲[i] − 𝒲[j];
        else
          B[p,i] ← B[p−1, j];
      else
        B[p,i] ← B[p − 1, j];
    B[1, SH_p+1] ← ∞;
  return B_opt ← B[P, N];
```

Figure 3.6: Dynamic-programming algorithm with static seperator-index bounding.

than w_{max}, the average task weight within subchain $\mathcal{T}_{i,j}$ satisfies $\Omega(w_{avg})$. That is, $\Delta_{i,j} = j - i + 1 = O(W_{i,j}/w_{avg})$. This model, referred to here as model $\mathcal{M}$, directly induces $\Delta S_p = O(P\, w_{max}/w_{avg})$, since $\Delta W_p \leq \Delta W_{P/2} = P\, w_{max}/2$ for $p = 1, 2, \ldots, P-1$.

3.3.2 Dynamic-Programming Algorithm with Static Separator-Index Bounding

The proposed DP algorithm, referred to here as the DP+ algorithm, exploits bounds on the separator indices for an efficient solution. Figure 3.6 illustrates the proposed DP+ algorithm, where input parameters SL and SH denote the index bound arrays, each of size P, computed according to Corollary 3.3.5 with $B_f = B_{RB}$. Note that $SL_P = SH_P = N$, since only $B[P, N]$ needs to be computed in the last row. As seen in Figure 3.6, only B_j^p values for $j = SL_p, SL_p+1, \ldots, SH_p$ are computed at each row p by exploiting Corollary 3.3.5, which ensures existence of an optimal partition $\Pi_{opt} = \langle s_0, s_1, \ldots, s_P \rangle$ with $SL_p \leq s_p \leq SH_p$. Thus, these B_j^p values will suffice for correct computation of B_i^{p+1} values for $i = SL_{p+1}, SL_{p+1}+1, \ldots, SH_{p+1}$ at the next row $p+1$.

As seen in Figure 3.6, explicit range checking is avoided in this algorithm for utmost efficiency. However, the j-index may proceed beyond SH_p to SH_p+1 within the *repeat-until-loop* while computing B_i^{p+1} with $SL_{p+1} \leq i \leq SH_{p+1}$ in two cases. In both cases, functions $W_{j+1,i}$ and B_j^p intersect in the open interval (SH_p, SH_p+1), so that $B_{SH_p}^p < W_{SH_p+1,i}$ and $B_{SH_p+1}^p \geq W_{SH_p+2,i}$. In the first case, $i = SH_p+1$, so that $W_{j+1,i}$ and B_j^p intersect in $(i-1, i)$, which implies that $B_i^{p+1} = W_{i-1,i}$, with $j_i^{p+1} = SL_p$, since $W_{i-1,i} < B_i^p$, as mentioned in § 3.2.2. In the second case,

$i > SH_p+1$, for which Corollary 3.3.5 guarantees that $B_i^{p+1} = W_{SL_p+1,i} \leq B^p_{SH_p+1}$, and thus we can safely select $j_i^{p+1} = SL_p$. Note that $W_{SL_p+1,i} = B^p_{SH_p+1}$ may correspond to a case leading to another optimal partition with $j_i^{p+1} = s_{p+1} = SH_p + 1$. As seen in Figure 3.6, both cases are efficiently resolved simply by storing ∞ to $B^p_{SH_p+1}$ as a *sentinel*. Hence, in such cases, the condition $W_{SH_p+1,i} < B^p_{SH_p+1} = \infty$ in the *if-then* statement following the *repeat-until-loop* is always true, so that the j-index automatically moves back to SH_p. The scheme of computing $B^p_{SH_p+1}$ for each row p, which seems to be a natural solution, does not work since correct computation of $B^{p+1}_{SH_{p+1}+1}$ may necessitate more than one B_j^p value beyond the SH_p index bound.

A nice feature of the DP approach is that it can be used to generate all optimal partitions by maintaining a $P \times N$ matrix to store the minimum j_i^p index values defining the B_i^p values at the expense of increased execution time and an increase in the asymptotic space requirement. Recall that index bounds SL and SH computed according to Corollary 3.3.5 restrict the search space for at least one optimal solution. The index bounds can be computed according to Lemma 3.3.4 for this purpose, since the search space restricted by Lemma 3.3.4 includes all optimal solutions.

The running time of the proposed DP+ algorithm is $O(N + P \log N) + \sum_{p=1}^{P} \theta(\Delta S_p)$. Here, $O(N)$ cost comes from the initial prefix-sum operation on the $\mathcal{W}$ array, and $O(P \log N)$ cost comes from the running time of the RB heuristic and computing the separator-index bounds SL and SH according to Corollary 3.3.5. Under model $\mathcal{M}$, $\Delta S_p = O(P w_{max}/w_{avg})$, and hence the complexity is $O(N + P \log N + P^2\, w_{max} \,/\, w_{avg})$. The algorithm becomes linear in N when the separator-index ranges do not overlap, which is guaranteed by the condition $w_{max} = O(2W_{tot}/P^2)$.

3.3.3 Iterative Refinement Algorithms

In this work, we improve the MS algorithm and propose a novel CCP algorithm, namely the *bidding* algorithm, which is run-time efficient for small-to-medium number of processors. The main difference between the MS and the bidding algorithms is as follows: the MS algorithm moves along a sequence of feasible bottleneck values, whereas the bidding algorithm moves along a sequence of infeasible bottleneck values so that the first feasible bottleneck value becomes the optimal value.

Improving the MS Algorithm

The performance of the MS algorithm [65] strongly depends on the initial partition. The initial partition proposed by Manne and Sorevik [65] satisfies the leftist partition constraint, but it leads to very poor run-time performance. Here, we propose using the partition generated by *PROBE(B^*)* as an initial partition. This partition is also a leftist partition, since moving any separator to the left will not decrease the load of the bottleneck processor. This simple observation leads to significant improvement in run-time performance of the algorithm. Also, using a heap as a priority queue does not give better run-time performance than using a running maximum despite its superior asymptotic complexity. In our implementation, we use a running maximum.

Bidding Algorithm

This algorithm increases the bottleneck value gradually, starting from the ideal bottleneck value B^*, until it finds a feasible partition, which happens to be an optimal one. Consider a partition $\Pi_t = \langle s_0, s_1, \ldots, s_P \rangle$ constructed by *PROBE(B_t)* for an infeasible B_t. After detecting the infeasibility of this B_t value, the point is to determine the next larger bottleneck value B to be investigated. Clearly, the separator indices of the partitions to be constructed by future *PROBE(B)* calls with $B > B_t$ will never be to the left of the respective separator indices of Π_t. Moreover, at least one of the separators should move right in hope of feasibility, since the load of the last processor determines infeasibility of the current B_t value (i.e., $L_P > B_t$). To avoid missing the smallest feasible bottleneck value, the next larger B value is selected as the minimum of processor loads that will be obtained by moving the end-index of every processor to the right by one position. That is, the next larger B value is equal to $\min\{\min_{1 \le p < P}\{L_p + w_{s_p+1}\}, L_P\}$. Here, we call the $L_p + w_{s_p+1}$ value the *bid* of processor $\mathcal{P}_p$, which refers to the load of $\mathcal{P}_p$ if the first task t_{s_p+1} of the next processor is augmented to $\mathcal{P}_p$. The bid of the last processor $\mathcal{P}_P$ is equal to the load of the remaining tasks. If the smallest bid B comes from processor $\mathcal{P}_b$, probing with new B is performed only for the remaining processors $\langle \mathcal{P}_b, \mathcal{P}_{b+1}, \ldots \mathcal{P}_P \rangle$ in the suffix $\mathcal{W}_{s_{b-1}+1:N}$ of the $\mathcal{W}$ array.

The bidding algorithm is presented in Figure 3.7. The innermost *while-loop* implements a linear probing scheme, such that the new positions of the separators are determined by moving them to the right, one by one. This linear probing scheme is selected because new positions of

```
BIDDING (W, N, P)
  s_p ← 0  for  p ← 0, 1, ..., P−1;   s_P ← N;
  BIDS[0].B ← L_r ← W_tot;
  B ← B*;   k ← 0;
  while L_r > B do
    repeat p ← p + 1;
      if s_p = 0 then
        s_p ← BINSRCH (W, s_{p−1} + 1, N, W[s_{p−1}] + B);
        L_p ← W[s_p] − W[s_{p−1}];
      else
        while L_p + w_{s_p+1} ≤ B do
          s_p ← s_p + 1;
          L_p ← L_p + w_{s_p};
      mybid ← L_p + w_{s_p+1};
      if mybid ≤ BIDS[p−1].B then
        BIDS[p].⟨B, q⟩ ← ⟨mybid, p⟩;
      else
        BIDS[p].⟨B, q⟩ ← BIDS[p−1].⟨B, q⟩;
      L_r ← W_tot − W[s_p];    rbid ← L_r / (P − p);
    until rbid > B or p = P − 1;
    if rbid < BIDS[p].B then
      B ← rbid;
    else
      ⟨B, k⟩ ←BIDS[p].⟨B, q⟩;
    p ← p − 1;
  return  B_opt ← B;
```

Figure 3.7: Bidding algorithm.

separators are likely to be in a close neighborhood of previous ones. Note that binary search is used only for setting the separator indices for the first time. After the separator index s_p is set for processor $\mathcal{P}_p$ during linear probing, the *repeat-until-loop* terminates if it is not possible to partition the remaining subchain $\mathcal{T}_{s_p+1,N}$ into $P-p$ processors without exceeding the current B value, i.e., $rbid = L_r/(P-p) > B$, where L_r denotes the weight of the remaining subchain. In this case, the next larger B value is determined by considering the best bid among the first p processors and *rbid*.

As seen in Figure 3.7, we maintain a prefix-minimum array *BIDS* for computing the next larger B value. Here, *BIDS* is an array of records of length P, where $BIDS[p].B$ and $BIDS[p].b$ store the best bid value of the first p processors and the index of the defining processor, respectively. $BIDS[0]$ helps the correctness of the running prefix-minimum operation.

The complexity of the bidding algorithm for integer task weights under model $\mathcal{M}$ is $O(P\log N + P\,w_{max} + P^2(w_{max}/w_{avg}))$. Here, $O(P\log N)$ cost comes from initial settings of separators through binary search. The B value is increased at most $B_{opt} - B^* < w_{max}$ times, and each time the next B value can be computed in $O(P)$ time, which induces the cost $O(P\,w_{max})$. The total area scanned by the separators is at most $O(P^2(w_{max}/w_{avg}))$. For non-integer task weights complexity can reach $O(P \log N + P^3(w_{max}/w_{avg}))$ in the worst case, which occurs when only one separator index moves to the right by one position at each B value. We should note here that using a min-heap for finding the next B value enables terminating a repeat-loop iteration as soon as a separator-index does not move. The trade-off in this scheme is the $O(\log P)$ cost incurred at each separator-index move due to respective key-update operations on the heap. We implemented this scheme as well and experimentally verified that it increases overall execution time in the CCP instances of § 3.5.

3.3.4 Parametric Search Algorithms

In this work, we apply theoretical findings given in § 3.3.1 for an improved probe algorithm. The improved algorithm, referred to here as the restricted probe (*RPROBE*), exploits bounds computed according to Corollary 3.3.5 (with $B_f = B_{RB}$) to restrict the search space for s_p separator values during binary searches on the $\mathcal{W}$ array. That is, *BINSRCH*($\mathcal{W}, SL_p, SH_p, Bsum$) in *RPROBE* searches $\mathcal{W}$ in the index range $[SL_p, SH_p]$ to find the index $SL_p \leq s_p \leq SH_p$ such that $\mathcal{W}[s_p] \leq Bsum$ and $\mathcal{W}[s_p{+}1] > Bsum$ via binary search. This scheme and Corollaries 3.3.2 and 3.3.6 reduce the complexity of an individual probe to $\sum_{p=1}^{P} \theta(\log \Delta_p) = O(P \log P + P \log(w_{max}/w_{avg}))$. Note that the complexity of *RPROBE* reduces to $O(P \log P)$ for sufficiently large P where $P = \Omega(w_{max}/w_{avg})$. Figures. 3.8, 3.9 and 3.10 illustrate *RPROBE* algorithms tailored for the respective parametric-search algorithms.

Approximate Bisection Algorithm with Dynamic Separator-Index Bounding

The proposed bisection algorithm, illustrated in Figure 3.8, searches the space of bottleneck values in range $[B^*, B_{RB}]$ as opposed to $[B^*, W_{tot}]$. In this algorithm, if *PROBE*(B_t) = *TRUE*, then the search space is restricted to $B \leq B_t$ values, and if *PROBE*(B_t) = *FALSE*, then the search space is restricted to $B > B_t$ values. In this work, we exploit this simple observation to propose and develop a dynamic probing scheme that increases the efficiency of successive *PROBE* calls by modifying separator index-bounds depending on success and failure of the probes. Let $\Pi_t = \langle t_0, t_1, \ldots, t_P \rangle$ be the partition constructed by *PROBE*(B_t). Any future *PROBE*(B) call with $B \leq B_t$ will set the s_p indices with $s_p \leq t_p$. Thus, the search space for s_p can be restricted to those indices $\leq t_p$. Similarly, any future *PROBE*(B) call with $B \geq B_t$ will set s_p indices $\geq t_p$. Thus, the search space for s_p can be restricted to those indices $\geq t_p$.

```
ε-BISECT+ (W, N, SL, SH, P)
  LB ← B*;
  UB ← B_RB;
  repeat
    B_t ← (UB + LB) / 2;
    if RPROBE (B_t) then
      for p ← 1 to P − 1 do SH_p ← s_p;
      UB ← B_t;
    else
      for p ← 1 to P − 1 do SL_p ← s_p;
      LB ← B_t;
  until UB ≤ LB + ε;
  return B_opt ← UB;
```

```
RPROBE (B)
  Bsum ← B;
  for p ← 1 to P−1 do
    s_p ← BINSRCH (W, SL_p, SH_p, Bsum);
    Bsum ← W[s_p] + B;
  if Bsum ≥ W_tot then
    return TRUE;
  else
    return FALSE;
```

Figure 3.8: Bisection as an ϵ-approximation algorithm with dynamic seperator-index bounding.

As illustrated in Figure 3.8, dynamic update of separator-index bounds can be performed in $\theta(P)$ time by a *for-loop* over SL or SH arrays, depending on failure or success, respectively, of *RPROBE(B_t)*. In our implementation, however, this update is efficiently achieved in $O(1)$ time through the pointer assignment $SL \leftarrow \Pi$ or $SH \leftarrow \Pi$ depending on failure or success of *RPROBE(B_t)*.

Similar to the ϵ-BISECT algorithm, the proposed ϵ-BISECT+ algorithm is also an ϵ-approximation algorithm for general workload arrays. However, both the ϵ-BISECT and ϵ-BISECT+ algorithms become exact algorithms for integer-valued workload arrays by setting $\epsilon = 1$. As shown in Lemma 3.2.1, $B_{RB} < B^* + w_{max}$. Hence, for integer-valued workload arrays the maximum number of probe calls in the ϵ-BISECT+ algorithm is $\log w_{max}$, and thus the overall complexity is $O(N + P \log N + \log(w_{max})(P \log P + P \log(w_{max}/w_{avg})))$ under model $\mathcal{M}$. Here, $O(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$, and $O(P \log N)$ cost comes from the running time of the RB heuristic and computing the separator-index bounds SL and SH according to Corollary 3.3.5.

Bisection as an Exact Algorithm

In this section, we will enhance the bisection algorithm to be an exact algorithm for general workload arrays by appropriate updating of lower and upper bounds after each probe. The idea is to move upper and lower bounds on the value of an optimal solution to a realizable bottleneck value, i.e., the total weight of a subchain of $\mathcal{W}$ after each probe. This reduces the search space to a finite set of realizable bottleneck values, as opposed to an infinite space of bottleneck values defined by a range $[LB, UB]$. Each bisection step is designed to eliminate at least one candidate value, and thus the algorithm terminates in finite number of steps to find the optimal bottleneck value.

After a probe *RPROBE(B_t)*, the current upper bound value UB is modified if *RPROBE(B_t)*

```
EXACT-BISECT (W, N, SL, SH, P)
  LB ← B*;   UB ← B_RD;
  repeat
    B_t ← (UB + LB) / 2;
    if RPROBE (B_t) then
      for p ← 1 to P − 1 do SH_p ← s_p;
      UB ← max_{1≤p≤P}{L_p};
    else
      for p ← 1 to P − 1 do SH_p ← s_p;
      LB ← min{min_{1≤p<P}{L_p + w_{s_p+1}}, L_P};
  until UB = LB;
  return B_opt ← UB;
```

```
RPROBE (B)
  Bsum ← B;
  for p ← 1 to P−1 do
    s_p ← BINSRCH (W, SL_p, SH_p, Bsum);
    Bsum ← W[s_p] + B;
    L_p ← W[s_p] − W[s_{p−1}];
  L_P ← W[N] − W[s_{P−1}];
  if L_P ≤ B then
    return TRUE;
  else
    return FALSE;
```

Figure 3.9: Exact bisection algorithm with dynamic seperator-index bounding.

succeeds. Note that *RPROBE*(B_t) not only determines the feasibility of B_t, but also constructs a partition Π with $cost(\Pi_t) \leq B_t$. Instead of reducing the upper bound UB to B_t, we can further reduce UB to the bottleneck value $B = cost(\Pi_t) \leq B_t$ of the partition Π_t constructed by *RPROBE*(B_t). Similarly, the current lower bound LB is modified when *RPROBE*(B_t) fails. In this case, instead of increasing the bound LB to B_t, we can exploit the partition Π_t constructed by *RPROBE*(B_t) to increase LB further to the smallest realizable bottleneck value B greater than B_t. How to compute this value is already described in our bidding algorithm:

$$B = \min\{\min_{1 \leq p < P}\{L_p + w_{s_p+1}\}, L_P\},$$

where L_p denotes the load of processor $\mathcal{P}_p$ in Π_t. Figure 3.9 presents the pseudocode of our algorithm.

Each bisection step divides the set of candidate realizable bottleneck values into two sets, and eliminates one of them. The initial set can have a size between 1 and N^2. Assuming the size of the eliminated set can be anything in the range 1–N^2, the expected complexity of the algorithm will be

$$T(N) = \frac{1}{N^2}\sum_{i=1}^{N^2} T(i) + O(P \log P + P \log(w_{max}/w_{avg})),$$

which has the solution $O(P \log P \log N + P \log N \log(w_{max}/w_{avg}))$. Here, $O(P \log P + P \log(w_{max}/w_{avg}))$ is the cost of a probe operation, and $\log N$ is the expected number of probes. Thus, the overall complexity becomes $O(N + P \log P \log N + P \log N \log(w_{max}/w_{avg}))$, where $O(N)$ cost comes from the initial prefix-sum operation on the $\mathcal{W}$ array.

Improving Nicol's Algorithm as a Divide-and-Conquer Algorithm

Theoretical findings of previous sections can be exploited at different levels to improve performance of Nicol's algorithm. A trivial improvement is to use the proposed restricted probe function instead of the conventional one. The careful implementation scheme given in Figure 3.5(b) enables use of dynamic separator-index bounding. Here, we exploit the idea behind the bisection algorithm to propose an efficient divide-and-conquer approach for Nicol's algorithm.

Consider the sequence of probes of the form $PROBE(W_{1,j})$ performed by Nicol's algorithm for processor $\mathcal{P}_1$ to find the smallest index $j = i_1$ such that $PROBE(W_{1,j}) = TRUE$. Starting from a naive bottleneck-value range $(LB_0 = 0,\ UB_0 = W_{tot})$, success and failure of these probes can narrow this range to (LB_1, UB_1). That is, each $PROBE(W_{1,j}) = TRUE$ decreases the upper bound to $W_{1,j}$ and each $PROBE(W_{1,j}) = FALSE$ increases the lower bound to $W_{1,j}$. Clearly, we will have $(LB_1 = W_{1,i_1-1},\ UB_1 = W_{1,i_1})$ at the end of this search process for processor $\mathcal{P}_1$. Now consider the sequence of probes of the form $PROBE(W_{i_1,j})$ performed for processor $\mathcal{P}_2$ to find the smallest index $j = i_2$ such that $PROBE(W_{i_1,j}) = TRUE$. Our key observation is that the partition $\Pi_t = \langle 0, t_1, t_2, \ldots, t_{P-1}, N\rangle$ to be constructed by any $PROBE(B_t)$ with $LB_1 < B_t = W_{i_1,j} < UB_1$ will satisfy $t_1 = i_1 - 1$, since $W_{1,i_1-1} < B_t < W_{1,i_1}$. Hence, probes with $LB_1 < B_t < UB_1$ for processor $\mathcal{P}_2$ can be restricted to be performed in $\mathcal{W}_{i_1:N}$, where $\mathcal{W}_{i_1:N}$ denotes the $(N - i_1 + 1)$th suffix of the prefix-summed $\mathcal{W}$ array. This simple yet effective scheme leads to an efficient divide-and-conquer algorithm as follows. Let $\mathcal{T}_i^{P-p}$ denote the CCP subproblem of $(P-p)$-way partitioning of the $(N-i+1)$th suffix $\mathcal{T}_{i,N} = \langle t_i, t_{i+1}, \ldots, t_N\rangle$ of the task chain $\mathcal{T}$ onto the $(P-p)$th suffix $\mathcal{P}_{p,P} = \langle \mathcal{P}_{p+1}, \mathcal{P}_{p+2}, \ldots, \mathcal{P}_P\rangle$ of the processor chain $\mathcal{P}$. Once the index i_1 for processor $\mathcal{P}_1$ is computed, the optimal bottleneck value B_{opt} can be defined by either W_{1,i_1} or the the bottleneck value of an optimal $(P-1)$-way partitioning of the suffix subchain $\mathcal{T}_{i_1,N}$. That is, $B_{opt} = \min\{B_1 = W_{1,i_1}, C(\Pi_{i_1}^{P-1})\}$. Proceeding in this way, once the indices $\langle i_1, i_2, \ldots, i_p\rangle$ for the first p processors $\langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_p\rangle$ are determined, then $B_{opt} = \min\{\min_{1\le b\le p}\{B_b = W_{i_{b-1},i_b}\}, C(\Pi_{i_p}^{P-p})\}$.

This approach is presented in Figure 3.10. At the bth iteration of the outer *for-loop*, given i_{b-1}, i_b is found in the inner *while-loop* by conducting probes on $\mathcal{W}_{i_{b-1}:N}$ to compute $B_b = W_{i_{b-1},i_b}$. The dynamic bounds on the separator indices are exploited in two distinct ways according to Theorem 3.3.7. First, the restricted probe function *RPROBE* is used for probing. Second, the

```
NICOL+ (𝒲, N, SL, SH, P)
  i_0 ← 1;   LB ← W_tot/P;   UB ← B_RB;
  for b ← 1 to P − 1 do
    ilow ← SL_b;   ihigh ← SH_b;
    while ilow < ihigh do
      imid ← (ilow + ihigh) / 2;
      B ← 𝒲[imid] − 𝒲[i_{b−1} − 1];
      if LB ≤ B < UB then
        if RPROBE (b, imid, B) then
          for p ← b + 1 to P − 1 do  SH_p ← s_p;
          UB ← B;
          ihigh ← imid;
        else
          for p ← b + 1 to P − 1 do  SL_p ← s_p;
          LB ← B;
          ilow ← imid + 1;
      elseif B ≥ UB
        ihigh ← imid;
      else
        ilow ← imid + 1;
    i_b ← ihigh;
    B_b ← 𝒲[i_b] − 𝒲[i_{b−1} − 1];
  B_P ← 𝒲[N] − W[i_{P−1} − 1];
  return min_{1≤b≤P} {B_b};

RPROBE (b, imid, B)
  Bsum ← 𝒲[imid] + B;
  for p ← b + 1 to P − 1 do
    s_p ← BINSRCH (𝒲, SL_p, SH_p, Bsum);
    Bsum ← W[s_p] + B;
  if Bsum ≥ W_tot then
    return TRUE;
  else
    return FALSE;
```

Figure 3.10: Nicol's algorithm with dynamic separator-index bounding.

search spaces for the bottleneck values of the processors are restricted. That is, given i_{b-1}, the binary search for i_b over all subchain weights of the form $W_{i_{b-1}+1,j}$ for $i_{b-1} < j \leq N$ is restricted to $W_{i_{b-1}+1,j}$ values for $SL_b \leq j \leq SH_b$.

Under model $\mathcal{M}$, the complexity of this algorithm is $O(N+P\log N+w_{max}(P\log P+P\log(w_{max}/w_{avg})))$ for integer task weights, because the number of probes cannot exceed w_{max}, since there are at most w_{max} distinct bound values in the range $[B^*, B_{RB}]$. For non-integer task weights, the complexity can be given as $O(N+P\log N+w_{max}(P\ \log P)^2+w_{max}P^2\log P\log(w_{max}/w_{avg}))$, since the algorithm makes $O(w_{max}P\log P)$ probes. Here, $O(N)$ cost comes from the initial prefix-sum operation on $\mathcal{W}$, and $O(P\log N)$ cost comes from the running time of the RB heuristic and computing the separator-index bounds SL and SH according to Corollary 3.3.5.

3.4 Load Balancing Applications

Here we describe load-balancing applications used to test the performance of the proposed algorithms.

3.4.1 Parallel Sparse Matrix Vector Multiplication

Sparse matrix vector multiplication (SpMxV) is one of the most important kernels in scientific computing. Parallelization of repeated SpMxV computations requires partitioning and distribution of the matrix. Two possible 1D sparse-matrix partitioning schemes are *rowwise striping* (RS) and *columnwise striping* (CS). Consider parallelization of SpMxV operations of the form $y = A\,x$ in an iterative solver, where A is an $N \times N$ sparse matrix and y and x are N-vectors. In RS, processor $\mathcal{P}_p$ owns the pth row stripe A_p^r of A and is responsible for computing $y_p = A_p^r\,x$, where y_p is the pth stripe of vector y. In CS, processor $\mathcal{P}_p$ owns the pth column stripe A_p^c of A and is responsible for computing $y^p = A_p^c\,x$, where $y = \sum_{p=1}^{P} y^p$. All vectors used in the solver are divided conformally with row (column) partitioning in the RS (CS) scheme, to avoid communication. RS and CS schemes require communication before or after local SpMxV computations, thus they can also be considered as *pre* and *post* communication schemes, respectively. In RS, each task $t_i \in \mathcal{T}$ corresponds to the atomic task of computing the inner-product of row i of matrix A with column vector x. In CS, each task $t_i \in \mathcal{T}$ corresponds to the atomic task of computing the sparse DAXPY operation $y = y + x_i a_{*i}$, where a_{*i} is the ith column of A. Each nonzero entry in a row and column of A incurs a multiply-and-add operation, thus the computational load w_i of task t_i is the number of nonzero entries in row i (column i) in the RS (CS) scheme. This defines how load balancing problem for rowwise and columnwise partitioning of a sparse matrix with a given ordering can be modeled as a CCP problem.

In RS (CS), by allowing only row (column) reordering, load balancing problem can be described as the number partitioning problem, which is NP-Hard [34]. By allowing both row and column reordering, the problem of minimizing communication overhead while maintaining load balance can be described as graph and hypergraph partitioning problems [14, 46], which are NP-Hard [35, 64] as well. However, possibly high preprocessing overhead involved in these models may not be justified in some applications. If the partitioner is to be used as part of a run-time library for a parallelizing compiler for a data-parallel programming language [96, 98], row and column reordering lead to high memory requirement due to the irregular mapping table and extra level of indirection in locating distributed data during each multiply-and-add operation [97]. Furthermore, in some applications, the natural row and column ordering of the sparse matrix may already be likely to induce small communication overhead (e.g., banded matrices).

The proposed CCP algorithms are surprisingly fast, so that the initial prefix-sum operation dominates their execution times in sparse-matrix partitioning. In this work, we exploit the standard *compressed row storage* (CRS) and *compressed column storage* (CCS) data structures for sparse matrices to avoid the prefix-sum operation. In CRS, an array *DATA* of length NZ stores nonzeros of matrix A, in row-major order, where $NZ = W_{tot}$ denotes the total number of nonzeros in A. An index array *COL* of length NZ stores the column indices of respective nonzeros in array *DATA*. Another index array *ROW* of length $N+1$ stores the starting indices of respective rows in the other two arrays. Hence, any subchain weight $W_{i,j}$ can be efficiently computed using $W_{i,j} = ROW[j+1] - ROW[i]$ in $O(1)$ time without any preprocessing overhead. CCS is similar to CRS with rows and columns interchanged, thus $W_{i,j}$ is computed using $W_{i,j} = COL[j+1] - COL[i]$.

A Better Load Balancing Model for Iterative Solvers

The load balancing problem for parallel iterative solvers has usually been stated considering only the SpMxV computations. However, linear vector operations (i.e., DAXPY and inner-product computations) involved in iterative solvers may have a considerable effect on parallel performance with increasing sparsity of the coefficient matrix. Here, we consider incorporating vector operations into the load-balancing model as much as possible.

For the sake of discussion, we will investigate this problem for a coarse-grain formulation [6, 7, 89] of the *conjugate-gradient* (CG) algorithm. Each iteration of CG involves, in order of computational dependency, one SpMxV, two inner-products, and three DAXPY computations. DAXPY computations do not involve communication, whereas inner-product computations necessitate a post global *reduction* operation [62] on results of the two local inner-products. In rowwise striping, pre-communication operations needed for SpMxV and the global reduction operation constitute pre- and post-synchronization points, respectively, for the aggregate of one local SpMxV and two local inner-product computations. In columnwise striping, the global reduction operation in the current iteration and post-communication operations needed for SpMxV in the next iteration constitute the pre- and post-synchronization points, respectively, for an aggregate of three local DAXPY and one local SpMxV computations. Thus, columnwise striping may be favored for a wider coverage of load balancing. Each vector entry incurs a multiply-and-add operation during each linear vector operation. Hence, DAXPY computations can easily be incorporated into the load-balancing model for the CS scheme by adding a cost of 3 to the computational weight w_i of atomic task t_i representing column i of matrix A. The initial prefix-sum operation can still be avoided by computing a subchain weight $W_{i,j}$ as $W_{i,j} = COL[j+1] - COL[i] + 3(j-i+1)$ in constant time. Note that two local inner-product computations still remain uncovered in this balancing model.

3.4.2 Sort-First Parallel Direct Volume Rendering

Direct volume rendering (DVR) methods are widely used in rendering unstructured volumetric grids for visualization and interpretation of computer simulations performed for investigating physical phenomena in various fields of science and engineering. A DVR application contains two interacting domains: object space and image space. Object space is a 3D domain containing the volume data to be visualized. Image space (screen) is a 2D domain containing pixels from which rays are shot into the 3D object domain to determine the color values of the respective pixels. Based on these domains, there are basically two approaches for data parallel DVR: image-space and object-space parallelism, which are also called as sort-first and sort-last parallelism according to the taxonomy based on the point of data redistribution in the rendering pipeline [68]. Pixels or pixel blocks constitute the atomic tasks in sort-first parallelism, whereas volume elements (primitives) constitute the atomic tasks in sort-last parallelism.

In sort-first parallel DVR, screen space is decomposed into regions and each region is assigned to a separate processor for local rendering. The primitives, whose projection areas intersect more than one region, are replicated. Sort-first parallelism has an advantage of processors generating complete images for their local screen subregion, but it faces load-balancing problems in the DVR of unstructured grids due to uneven on-screen primitive distribution.

Image-space decomposition schemes for sort-first parallel DVR can be classified as *static* and *adaptive* [63]. Static decomposition is a view-independent scheme, and the load-balancing problem is solved implicitly by scattered assignment of pixels or pixel blocks. Load-balancing performance of this scheme depends on the assumption that neighbor pixels are likely to have equal workload since they are likely to have similar views of the volume. As the scattered assignment scheme assigns adjacent pixels or pixel blocks to different processors, it disturbs image-space coherency and increases the amount of primitive replication. Adaptive decomposition is a view-dependent scheme, and the load-balancing problem is solved explicitly by using the primitive distribution on the screen.

In adaptive image-space decomposition, the number of primitives with bounding-box approximation is taken to be the workload of a screen region. Primitives constituting the volume are tallied to a 2D coarse mesh superimposed on the screen. Some primitives may intersect multiple cells. The inverse-area heuristic [69] is used to decrease the amount of error due to counting such primitives multiple times. Each primitive increments the weight of each cell it intersects by a value inversely proportional to the number of cells the primitive intersects. In this heuristic, if we assume that there are no shared primitives among screen regions, then the sum of the weights of individual mesh cells forming a region gives the number of primitives in that region. Shared primitives may still cause some errors, but such errors are much less than counting such primitives multiple times while adding mesh-cell weights.

Minimizing the perimeter of the resulting regions in the decomposition is expected to minimize the communication overhead due to the shared primitives. 1D decomposition, i.e., horizontal and vertical striping of the screen, suffer from unscalability. A Hilbert space-filling curve [76] is widely used for 2D decomposition of 2D non-uniform workloads. In this scheme [63], the 2D coarse mesh superimposed on the screen is traversed according to the Hilbert curve to map the 2D coarse mesh to a 1D chain of mesh cells. The load-balancing problem in this decomposition scheme then reduces to the CCP problem. Using a Hilbert curve as the space-filling curve is an implicit effort towards reducing the total perimeter, since the Hilbert curve avoids jumps during the traversal of the 2D coarse mesh. Note that the 1D workload array used for partitioning is a real-valued array, because of the inverse-area heuristic used for computing weights of the coarse-mesh cells.

3.5 Experimental Results

All algorithms were implemented in the *C* programming language. All experiments were carried out on a workstation equipped with a 133MHz *PowerPC* and 64 MB of memory. We have experimented with $P = 16$, 32, 64, 128, and 256 way partitioning of each test data.

The dataset for the sparse-matrix decomposition comes from matrices of linear programming test problems obtained from the *Netlib* suite [36] and the *IOWA Optimization Center* [49]. The sparsity patterns of these matrices are obtained by multiplying the respective rectangular constraint

Table 3.1: Properties of the test set.

SPARSE-MATRIX DATASET							DIRECT VOLUME RENDERING (DVR) DATASET					
name	# of tasks	workload: # of nonzeros				ex. time	name	# of tasks	workload			
		total	per row/col (task)			SpMxV			total	per task		
	N	W_{tot}	w_{avg}	w_{min}	w_{max}	msecs		N	W_{tot}	w_{avg}	w_{min}	w_{max}
NL	7039	105089	14.93	1	361	22.55	blunt256	17303	303K	17.54	0.020	1590.97
cre-d	8926	372266	41.71	1	845	72.20	blunt512	93231	314K	3.36	0.004	661.50
CQ9	9278	221590	23.88	1	702	45.90	blunt1024	372824	352K	0.94	0.001	411.04
ken-11	14694	82454	5.61	2	243	19.65	post256	19653	495K	25.19	0.077	3245.50
mod2	34774	604910	17.40	1	941	124.05	post512	134950	569K	4.22	0.015	1092.00
world	34506	582064	16.87	1	972	119.45	post1024	539994	802K	1.49	0.004	1546.78

matrices with their transposes. Hence, load balancing in rowwise and columnwise decompositions are equivalent since the resulting matrices are symmetric. Table 3.1 summarizes the properties of the test matrices. Note that the number of tasks in the sparse-matrix (SpM) dataset also refers to the number of rows and columns of the respective matrix. The table also displays the execution times of single SpMxV operations.

The dataset for image-space decomposition comes from sort-first parallel DVR of curvilinear grids *blunt-fin* and *post* representing the results of computational fluid dynamic simulations, which are commonly used in volume rendering. The raw grids consist of hexahedral elements and are converted into an unstructured tetrahedral data format by dividing each hexahedron into 5 tetrahedrons. Triangular faces of tetrahedrons constitute the primitives mentioned in § 3.4.2. Three distinct 1D workload arrays are constructed both for *blunt-fin* and *post* as described in § 3.4.2 for coarse meshes of resolutions 256×256, 512×512, and 1024×1024 superimposed on a screen of resolution 1024×1024. The properties of these six workload arrays are displayed in Table 3.1. The number of tasks is much less than coarse-mesh resolution because of the zero-weight tasks, which can be compressed to leave only nonzero-weight tasks. In Table 3.1, w_{avg}, w_{min} and w_{max} columns display the average, minimum, and maximum task weights.

The following abbreviations are used for the CCP algorithms: H1 and H2 refer to Miguet and Pierson's [67] heuristics described in § 3.2.1. RB refers to the recursive-bisection heuristic described in § 3.2.1. DP refers to the $O((N-P)P)$-time dynamic-programming algorithm given in Figure 3.1. MS refers to Manne and Sorevik's [65] iterative-refinement algorithm given in Figure 3.2. ϵBS refers to the ϵ-approximate bisection algorithm given in Figure 3.4. NC- and NC refer to the straightforward and careful implementations of Nicol's [70] parametric-search algorithm given in Figure 3.5(a) and Figure 3.5(b), respectively. Abbreviations ending with "+" are used to represent our improved versions of these algorithms. That is, DP+, ϵBS+, NC+ refer to our algorithms given

Table 3.2: Percent load imbalance values.

SPARSE-MATRIX DATASET						DVR DATASET					
CCP instance		HEURISTICS			OPT	CCP instance		HEURISTICS			OPT
name	K	H1	H2	RB		name	K	H1	H2	RB	
	16	2.60	2.44	1.20	0.35		16	4.60	1.20	0.49	0.34
	32	5.02	5.75	3.44	0.95		32	6.93	2.61	1.94	1.12
NL	64	8.95	9.01	5.60	2.37	blunt256	64	14.52	9.44	9.44	2.31
	128	33.13	27.16	22.78	4.99		128	38.25	24.39	16.67	4.82
	256	69.55	69.55	60.78	14.25		256	96.72	37.03	34.21	34.21
	16	2.27	0.98	0.53	0.45		16	0.95	0.98	0.98	0.16
	32	4.19	4.42	3.74	0.63		32	1.38	1.38	1.18	0.33
cre-d	64	7.12	4.92	4.34	1.73	blunt512	64	2.87	2.69	1.66	0.53
	128	25.57	18.73	16.70	2.88		128	5.62	8.45	4.62	0.97
	256	37.54	26.81	35.20	10.95		256	14.18	14.33	9.34	2.28
	16	1.85	1.85	0.58	0.58		16	0.94	0.57	0.95	0.10
	32	5.65	2.88	2.24	0.90		32	1.89	1.21	0.97	0.14
CQ9	64	13.25	11.49	7.64	1.43	blunt1024	64	4.99	2.16	1.44	0.26
	128	33.96	32.22	22.34	3.51		128	10.06	4.25	2.47	0.57
	256	58.62	58.62	58.62	14.72		256	19.65	9.68	9.68	0.98
	16	3.74	2.01	0.98	0.21		16	1.10	1.43	0.76	0.56
	32	3.74	4.67	3.74	1.18		32	3.23	3.98	3.23	1.11
ken-11	64	13.17	13.17	13.17	1.29	post256	64	17.28	11.04	10.90	3.10
	128	13.17	16.89	13.17	6.80		128	45.35	29.09	29.09	8.29
	256	50.99	50.99	50.99	7.11		256	67.86	67.86	67.86	67.86
	16	0.06	0.06	0.06	0.03		16	0.49	1.25	0.33	0.33
	32	0.19	0.19	0.19	0.07		32	0.94	1.61	0.90	0.58
mod2	64	7.42	2.72	2.18	0.18	post512	64	4.85	5.33	4.85	0.94
	128	16.15	6.29	2.46	0.41		128	18.03	14.55	10.15	1.72
	256	19.47	19.47	18.92	1.23		256	30.03	25.29	25.29	3.73
	16	0.27	0.18	0.09	0.04		16	0.70	0.70	0.53	0.20
	32	0.63	0.37	0.27	0.08		32	1.49	1.49	1.41	0.54
world	64	4.73	4.73	4.73	0.28	post1024	64	2.85	2.85	1.49	0.91
	128	6.37	6.37	6.37	0.76		128	13.15	11.79	9.10	1.11
	256	27.99	27.41	27.41	1.11		256	40.50	13.37	14.19	2.54
AVERAGES OVER K											
	16	1.76	1.15	0.74	0.36		16	1.44	1.01	0.64	0.28
	32	3.99	3.39	2.88	0.76		32	2.64	2.05	1.61	0.64
	64	9.01	6.78	5.69	1.43		64	7.89	5.58	4.96	1.31
	128	19.97	17.66	14.09	3.36		128	21.74	15.42	12.02	2.91
	256	43.89	38.91	36.04	9.18		256	44.82	27.93	26.76	18.60

in Figure 3.6, Figure 3.8, Figure 3.10, respectively, and MS+ refers to the algorithm described in § 3.3.3. BID refers to our bidding algorithm given in Figure 3.7. EBS refers to our exact bisection algorithm given in Figure 3.9. Both ϵBS and ϵBS+ algorithms are effectively used as exact algorithms for the SpM dataset with $\epsilon = 1$ for integer-valued workload arrays in the SpM dataset. However, the performance of these two algorithms was not tested on the DVR dataset, since they remain approximation algorithms because of the real-valued task weights in DVR.

Table 3.2 compares load-balancing quality of heuristics and exact algorithms. In this table, percent load-imbalance values are computed as $100 \times (B_{max} - B^*)/B^*$, where B_{max} denotes the bottleneck value of the respective partition and $B^* = W_{tot}/P$ denotes ideal bottleneck value. OPT values refer to load-imbalance values of optimal partitions produced by exact algorithms, i.e., $B_{max} = B_{opt}$. Table 3.2 clearly shows that considerably better partitions can be obtained in both

SpM and DVR datasets by using exact load-balancing algorithms instead of heuristics. The quality gap between exact algorithms and heuristics increases with increasing P. The only exceptions to these observations are 256-way partitioning of DVR instances *blunt256* and *post256*, for which the RB heuristic achieves optimal partitions.

Table 3.3 compares the performances of the static separator-index bounding schemes discussed in § 3.3.1. The values displayed in this table are the sums of the sizes of the separator-index ranges normalized with respect to N, i.e., $\sum_{p=1}^{P-1} \Delta S_p/N$, where $\Delta S_p = SH_p - SL_p + 1$. For each CCP instance, $(N-P)P$ represents the size of the search space for the separator indices and the total number of table entries referenced and computed by the DP algorithm. Columns labeled L1, L3 and C5 display total range sizes obtained according to Lemma 3.3.1, Lemma 3.3.3 and Corollary 3.3.5, respectively. As seen in Table 3.3, proposed practical scheme C5 achieves substantially better separator-index bounds than L1, despite its worse worst-case behavior (see Corollaries 3.3.2 and 3.3.6). Comparison of columns L3 and C5 shows the substantial benefit of performing left-to-right and right-to-left probes with B^* according to Lemma 3.3.4. Comparison of $(N-P)P$ and the C5 column reveals the effectiveness of the proposed separator-index bounding scheme in restricting the search space for separator indices in both SpM and DVR datasets. As expected, the performance of index bounding decreases with increasing P because of decreasing N/P values. The numbers for the DVR dataset are better than those for the SpM dataset because of the larger N/P values. In Table 3.3, values less than 1 indicate that the index bounding scheme achieves non-overlapping index ranges. As seen in the table, scheme C5 reduces the total separator-index range sizes below N for each CCP instance with $P \leq 64$ in both the SpM and DVR datasets. These results show that the proposed DP+ algorithm becomes a linear-time algorithm in practice.

The efficiency of the parametric-search approach depends on two critical issues: the number of probes and the cost of each probe. The dynamic index bounding schemes proposed for the parametric-search algorithms reduce the cost of an individual probe. Table 3.4 illustrates the performance of the proposed parametric-search algorithms in reducing the number of probes. For Table 3.4, in order to compare performances of EBS and ϵBS+ on the DVR dataset, we forced the ϵBS+ algorithm to find optimal partitions by running it with $\epsilon = w_{min}$ and then improving the resulting partition using the BID algorithm. Column ϵBS+&BID refers to this scheme, and

Table 3.3: Sizes of separator-index ranges normalized with respect to N.

SPARSE-MATRIX DATASET							DVR DATASET					
CCP instance							CCP instance					
name	K	(N-K)K	L1	L3	C5		name	K	(N-K)K	L1	L3	C5
	16	15.97	0.32	0.13	0.036			16	15.99	0.38	0.07	0.003
	32	31.86	1.23	0.58	0.198			32	31.94	1.22	0.30	0.144
NL	64	63.43	4.97	2.24	0.964		blunt256	64	63.77	5.10	3.81	0.791
	128	125.69	19.81	19.60	4.305			128	127.06	21.66	12.88	2.784
	256	246.73	77.96	82.83	22.942			256	252.23	101.68	50.48	10.098
	16	15.97	0.16	0.04	0.019			16	16.00	0.09	0.17	0.006
	32	31.89	0.76	0.90	0.137			32	31.99	0.35	0.26	0.043
cre-d	64	63.55	2.89	1.85	0.610		blunt512	64	63.96	1.64	0.69	0.144
	128	126.18	11.59	15.12	2.833			128	127.83	6.65	4.59	0.571
	256	248.69	46.66	57.54	16.510			256	255.30	28.05	16.71	2.262
	16	15.97	0.30	0.03	0.023			16	16.00	0.05	0.09	0.010
	32	31.89	1.21	0.34	0.187			32	32.00	0.18	0.18	0.018
CQ9	64	63.57	4.84	2.96	0.737		blunt1024	64	63.99	0.77	0.74	0.068
	128	126.25	19.20	18.66	3.121			128	127.96	3.43	2.53	0.300
	256	248.96	74.84	86.80	23.432			256	255.82	13.92	20.36	1.648
	16	15.98	0.28	0.11	0.024			16	15.99	0.35	0.04	0.021
	32	31.93	1.10	1.13	0.262			32	31.95	1.50	0.52	0.162
ken-11	64	63.73	4.42	7.34	0.655		post256	64	63.79	6.12	4.26	0.932
	128	126.89	17.60	14.56	4.865			128	127.17	26.48	23.68	4.279
	256	251.56	69.18	85.44	13.076			256	252.68	124.76	90.48	16.485
	16	15.99	0.16	0.00	0.002			16	16.00	0.13	0.02	0.003
	32	31.97	0.55	0.04	0.013			32	31.99	0.56	0.14	0.063
mod2	64	63.88	2.14	1.22	0.109		post512	64	63.97	2.33	2.41	0.413
	128	127.53	8.62	2.59	0.411			128	127.88	9.33	10.15	1.714
	256	254.12	34.49	39.02	2.938			256	255.52	37.08	46.28	6.515
	16	15.99	0.15	0.01	0.004			16	16.00	0.17	0.07	0.020
	32	31.97	0.59	0.06	0.020			32	32.00	0.54	0.27	0.087
world	64	63.88	2.30	2.81	0.158		post1024	64	63.99	2.19	0.61	0.210
	128	127.53	9.23	7.19	0.850			128	127.97	8.77	9.48	0.918
	256	254.11	36.97	53.15	2.635			256	255.88	34.97	27.28	4.479
AVERAGES OVER K												
	16	15.97	0.21	0.06	0.018			16	16.00	0.20	0.08	0.011
	32	31.88	0.86	0.68	0.136			32	31.98	0.73	0.28	0.096
	64	63.52	3.43	2.63	0.516			64	63.91	3.03	2.09	0.426
	128	126.07	13.68	12.74	2.731			128	127.65	12.72	10.55	1.428
	256	248.26	54.28	57.55	14.089			256	254.57	56.74	41.93	6.915

L1, L3 and C5 denote separator-index ranges obtained according to results of Lemma 3.3.1, Lemma 3.3.3 and Corollary 3.3.5.

Table 3.4: Number of probes performed by parametric search algorithms.

SPARSE-MATRIX DATASET								DVR DATASET						
CCP instance								CCP instance						
name	K	NC-	NC	NC+	ϵBS	ϵBS+	EBS	name	K	NC-	NC	NC+	ϵBS+&BID	EBS
	16	177	21	7	17	6	7		16	202	18	6	16 + 1	9
	32	352	19	7	17	7	7		32	407	19	7	19 + 1	10
NL	64	720	27	5	16	7	6	blunt256	64	835	18	10	21 + 1	12
	128	1450	40	14	17	8	8		128	1686	25	15	22 + 1	13
	256	2824	51	14	16	8	8		256	3556	203	62	23 + 1	11
	16	190	20	6	19	7	5		16	245	20	8	17 + 1	9
	32	393	25	11	19	9	8		32	502	24	13	18 + 1	10
cre-d	64	790	24	10	19	8	6	blunt512	64	1003	23	11	18 + 1	12
	128	1579	27	10	19	9	9		128	2023	26	12	20 + 1	13
	256	3183	37	13	18	9	9		256	4051	23	12	21 + 1	13
	16	182	19	3	18	6	5		16	271	21	10	16 + 1	13
	32	373	22	8	18	8	7		32	557	24	12	18 + 1	12
CQ9	64	740	29	12	18	8	8	blunt1024	64	1127	21	11	18 + 1	12
	128	1492	40	12	17	8	8		128	2276	28	13	19 + 1	13
	256	2971	50	14	18	9	9		256	4564	27	16	21 + 1	15
	16	185	21	6	17	6	6		16	194	22	9	18 + 1	7
	32	364	20	6	17	6	6		32	409	19	8	20 + 1	12
ken-11	64	721	48	9	17	7	7	post256	64	823	17	11	22 + 1	12
	128	1402	61	8	16	7	7		128	1653	19	13	22 + 1	14
	256	2783	96	7	17	8	8		256	3345	191	102	23 + 1	13
	16	210	20	6	20	5	4		16	235	24	9	16 + 1	10
	32	432	26	8	19	5	5		32	485	23	12	18 + 1	11
mod2	64	867	24	6	19	8	8	post512	64	975	22	13	20 + 1	14
	128	1727	38	7	20	7	7		128	1975	25	17	21 + 1	14
	256	3444	47	9	20	9	9		256	3947	29	20	22 + 1	15
	16	211	22	6	19	5	4		16	261	27	10	17 + 1	13
	32	424	26	5	19	6	6		32	538	23	13	19 + 1	14
world	64	865	30	12	19	9	9	post1024	64	1090	22	12	17 + 1	14
	128	1730	44	10	19	8	8		128	2201	25	15	21 + 1	16
	256	3441	44	11	19	10	10		256	4428	28	16	22 + 1	16
AVERAGES OVER K														
	16	193	20.5	5.7	18.5	5.8	5.2		16	235	22.0	8.7	16.7 + 1.0	10.2
	32	390	23.0	7.5	18.3	6.8	6.5		32	483	22.0	10.8	18.7 + 1.0	11.5
	64	784	30.3	9.0	18.0	7.8	7.3		64	976	20.5	11.3	19.3 + 1.0	12.7
	128	1564	41.7	10.2	18.0	7.8	7.8		128	1969	24.7	14.2	20.8 + 1.0	13.8
	256	3108	54.2	11.3	18.0	8.8	8.8		256	3982	83.5	38.0	22.0 + 1.0	13.8

values after "+" denote the number of additional bottleneck values tested by the BID algorithm. As seen in Table 3.4, exactly one final bottleneck-value test was needed by BID to reach an optimal partition in each CCP instance.

In Table 3.4, comparison of the NC- and NC columns shows that dynamic bottleneck-value bounding drastically decreases the number of probes in Nicol's algorithm. Comparison of the ϵBS and ϵBS+ columns in the SpM dataset shows that using B_{RB} instead of W_{tot} for the upper bound on the bottleneck values considerably reduces the number of probes. Comparison of the ϵBS+ and EBS columns reveals two different behaviors on SpM and DVR datasets. The discretized dynamic bottleneck-value bounding used in EBS produces only minor improvement on the SpM dataset because of the already discrete nature of integer-valued workload arrays. However, the effect of discretized dynamic bottleneck-value bounding is significant on the real-valued workload arrays of

Table 3.5: Partitioning times for sparse-matrix dataset as percents of SpMxV times.

SPARSE-MATRIX DATASET													
		HEURISTICS			EXACT ALGORITHMS								
CCP instance					EXISTING				PROPOSED				
name	K	H1	H2	RB	DP	MS	ϵBS	NC	DP+	MS+	ϵBS+	NC+	BID
NL	16	0.09	0.09	0.09	93	119	0.93	1.20	0.40	0.44	0.40	0.40	0.22
	32	0.18	0.18	0.13	177	194	1.77	2.17	1.73	1.73	0.80	0.80	0.44
	64	0.35	0.35	0.31	367	307	3.15	5.85	5.81	4.12	1.86	1.77	1.82
	128	0.89	0.84	0.71	748	485	6.30	17.12	19.87	26.92	4.26	7.32	4.21
	256	1.51	1.55	1.37	1461	757	11.09	40.31	91.80	96.41	9.80	15.70	21.06
cre-d	16	0.03	0.01	0.01	33	36	0.33	0.37	0.12	0.06	0.10	0.11	0.03
	32	0.04	0.06	0.06	71	61	0.65	0.93	0.47	1.20	0.29	0.33	0.10
	64	0.12	0.12	0.08	147	98	1.22	1.69	1.66	1.83	0.61	0.71	0.21
	128	0.28	0.29	0.14	283	150	2.41	3.59	5.47	10.94	1.68	1.68	0.61
	256	0.53	0.54	0.25	571	221	4.20	10.22	27.05	26.02	3.30	4.46	1.20
CQ9	16	0.04	0.04	0.04	59	73	0.48	0.59	0.20	0.13	0.15	0.13	0.17
	32	0.09	0.09	0.07	127	120	0.94	1.31	0.94	0.72	0.41	0.41	0.28
	64	0.20	0.17	0.15	248	195	1.85	3.18	3.01	4.47	1.00	1.44	0.68
	128	0.44	0.44	0.31	485	303	3.18	8.52	9.65	18.82	2.44	3.05	2.51
	256	0.92	0.92	0.63	982	469	6.51	20.33	69.56	72.2	5.32	7.45	14.92
ken-11	16	0.10	0.10	0.10	238	344	1.27	1.88	0.56	0.61	0.46	0.41	0.25
	32	0.20	0.20	0.15	501	522	2.29	3.10	3.82	4.78	1.22	1.17	1.63
	64	0.46	0.46	0.36	993	778	4.48	13.08	9.41	24.78	3.10	3.46	2.29
	128	1.17	1.17	0.71	1876	1139	9.21	39.59	50.13	50.03	6.41	6.62	17.4
	256	2.14	2.14	1.42	3827	1706	17.76	119.13	129.01	241.48	14.91	14.50	29.11
mod2	16	0.02	0.02	0.01	92	119	0.27	0.34	0.07	0.05	0.06	0.06	0.03
	32	0.04	0.04	0.02	188	192	0.51	0.78	0.19	0.18	0.16	0.15	0.07
	64	0.11	0.10	0.06	378	307	1.04	1.91	0.86	2.18	0.51	0.55	0.23
	128	0.24	0.23	0.11	751	482	2.28	5.92	2.61	3.72	1.06	1.23	0.53
	256	0.48	0.48	0.23	1486	756	4.26	11.14	12.11	50.04	2.91	3.43	3.66
world	16	0.02	0.02	0.02	99	122	0.29	0.33	0.08	0.05	0.07	0.08	0.03
	32	0.04	0.04	0.04	198	196	0.59	0.88	0.26	0.21	0.18	0.19	0.08
	64	0.12	0.11	0.09	385	306	1.16	1.70	1.09	4.84	0.64	0.54	0.35
	128	0.24	0.23	0.19	769	496	2.19	5.32	4.48	10.15	1.31	1.23	1.77
	256	0.47	0.47	0.36	1513	764	4.06	11.83	11.54	69.99	3.46	3.50	2.48
AVERAGES OVER K													
	16	0.05	0.05	0.05	102	136	0.60	0.78	0.24	0.22	0.21	0.20	0.12
	32	0.10	0.10	0.08	210	214	1.12	1.53	1.23	1.47	0.51	0.51	0.43
	64	0.23	0.22	0.18	420	332	2.15	4.57	3.64	7.04	1.29	1.41	0.93
	128	0.54	0.53	0.36	819	509	4.26	13.34	15.37	20.1	2.86	3.52	4.51
	256	1.01	1.01	0.71	1640	779	7.98	35.49	56.84	92.69	6.62	8.17	12.07

the DVR dataset.

Tables 3.5 and 3.6 display the execution times of CCP algorithms on the SpM and DVR datasets, respectively. In Table 3.5, the execution times are given as percents of single SpMxV times. For the DVR dataset, actual execution times (in msecs) are split into prefix-sum times and partitioning times. In Tables 3.5 and 3.6, execution times of existing algorithms and their improved versions are listed in the same order under each respective classification, so that improvements can be seen easily. In both tables, BID is listed separately since it is a new algorithm. Results of both ϵBS and ϵBS+ are listed in Table 3.5, since they are used as exact algorithms on SpM dataset with $\epsilon = 1$. Since neither ϵBS nor ϵBS+ can be used as an exact algorithm on the DVR dataset, EB—being an exact algorithm for general workload arrays—is listed separately in Table 3.6.

As seen in Tables 3.5 and 3.6, the RB heuristic is faster than both H1 and H2 for both SpM

Table 3.6: Partitioning times (in msecs) for DVR dataset.

DVR DATASET													
			HEURISTICS			EXACT ALGORITHMS							
CCP instance		prefix				EXISTING			PROPOSED				
name	K	sum	H1	H2	RB	DP	MS	NC	DP+	MS+	NC+	EBS	BID
	16		0.02	0.02	0.03	68	49	0.36	0.21	0.53	0.10	0.14	0.03
	32		0.05	0.05	0.05	141	77	0.77	0.76	0.76	0.21	0.27	0.24
blunt256	64	1.95	0.11	0.12	0.09	286	134	1.39	3.86	8.72	0.64	0.71	0.78
	128		0.24	0.25	0.20	581	206	3.66	12.37	29.95	1.78	1.84	2.33
	256		0.47	0.50	0.37	1139	296	52.53	42.89	0.78	13.96	3.11	56.69
	16		0.03	0.03	0.03	356	200	0.55	0.25	0.91	0.14	0.16	0.09
	32		0.07	0.07	0.07	792	353	1.31	1.23	2.13	0.44	0.43	0.22
blunt512	64	13.45	0.18	0.19	0.14	1688	593	2.65	3.68	9.28	0.94	1.10	0.45
	128		0.39	0.40	0.28	3469	979	5.84	15.01	46.72	2.29	2.63	1.65
	256		0.74	0.75	0.54	7040	1637	9.70	57.50	164.49	5.59	5.95	10.72
	16		0.03	0.03	0.03	1455	780	0.75	0.93	4.77	0.25	0.28	0.19
	32		0.12	0.11	0.08	3251	1432	1.81	2.02	8.92	0.60	0.62	0.31
blunt1024	64	59.12	0.27	0.27	0.19	6976	2353	3.50	7.35	22.28	1.27	1.37	1.39
	128		0.50	0.51	0.37	14337	3911	9.14	33.01	69.38	3.36	3.56	16.21
	256		0.97	1.00	0.68	29417	6567	16.59	180.09	538.10	8.48	8.98	16.29
	16		0.02	0.03	0.03	79	61	0.46	0.18	0.13	0.10	0.11	0.24
	32		0.05	0.05	0.05	157	100	0.77	1.05	1.74	0.26	0.31	0.76
post256	64	2.16	0.10	0.11	0.10	336	167	1.37	5.26	9.72	0.61	0.70	4.67
	128		0.25	0.26	0.18	672	278	2.90	22.94	55.10	1.65	1.77	23.96
	256		0.47	0.48	0.37	1371	338	45.16	84.78	0.77	13.04	3.62	299.32
	16		0.02	0.02	0.03	612	454	0.63	0.20	0.44	0.14	0.16	1.21
	32		0.07	0.07	0.07	1254	699	1.30	2.49	1.95	0.38	0.42	3.36
post512	64	20.55	0.18	0.18	0.13	2559	1139	2.58	16.93	38.73	0.97	1.07	8.53
	128		0.38	0.39	0.28	5221	1936	5.84	68.25	156.15	2.27	2.44	31.54
	256		0.70	0.73	0.53	10683	3354	12.67	256.48	726.54	5.44	5.39	140.88
	16		0.03	0.04	0.03	2446	1863	0.91	2.88	5.73	0.17	0.21	2.63
	32		0.09	0.08	0.08	5056	2917	1.61	13.57	17.37	0.51	0.58	12.73
post1024	64	69.09	0.25	0.26	0.17	10340	4626	3.56	35.05	33.25	1.35	1.47	32.53
	128		0.51	0.53	0.36	21102	7838	7.90	156.34	546.92	2.95	3.28	76.96
	256		0.95	0.98	0.67	43652	13770	16.30	764.59	1451.87	7.55	7.02	300.70
AVERAGES (normalized w.r.t. RB times) OVER K (prefix-sum time not included)													
	16		0.83	0.94	1.00	27863	18919	20.33	25.83	69.50	5.00	5.89	24.39
	32		1.10	1.06	1.00	23169	12156	18.47	47.37	72.82	5.83	6.46	39.02
	64		1.30	1.35	1.00	22636	9291	17.88	82.81	145.19	7.00	7.81	53.81
	128		1.35	1.39	1.00	22506	7553	20.46	168.36	481.19	8.60	9.31	86.81
	256		1.35	1.39	1.00	24731	6880	59.10	390.25	772.99	19.55	10.51	286.77
AVERAGES (normalized w.r.t. RB times) OVER K (prefix-sum time included)													
	16		1.00	1.00	1.00	32	23	1.08	1.04	1.09	1.01	1.02	1.03
	32		1.00	1.00	1.00	66	36	1.15	1.21	1.29	1.04	1.05	1.13
	64		1.00	1.00	1.00	135	58	1.27	1.97	2.90	1.11	1.12	1.55
	128		1.01	1.01	1.00	275	94	1.62	4.75	10.53	1.28	1.30	3.25
	256		1.02	1.02	1.00	528	142	7.98	14.64	13.71	2.95	1.55	26.73

and DVR datasets. As also seen in Table 3.2, RB finds better partitions than both H1 and H2 for both datasets. These results reveal RB's superiority to H1 and H2.

In Tables 3.5 and 3.6, relative performance comparison of existing exact CCP algorithms shows that NC is two orders of magnitude faster than both DP and MS for both SpM and DVR datasets, and ϵBS is considerably faster than NC for the SpM dataset. These results show that among existing algorithms the parametric-search approach leads to faster algorithms than either the dynamic-programming or the iterative-improvement approaches.

Tables 3.5 and 3.6 show that our improved algorithms are significantly faster than the respective existing algorithms. In the dynamic-programming approach, DP+ is two-to-three orders of magnitude faster than DP so that DP+ is competitive with the parametric-search algorithms. For the SpM dataset, DP+ is 630 times faster than DP on average in 16-way partitioning, and this ratio decreases to 378, 189, 106, and 56 with increasing number of processors. For the DVR dataset, if the initial prefix-sum is not included, DP+ is 1277, 578, 332, 159, and 71 times faster than DP for $P = 16$, 32, 64, 128, and 256, respectively, on average. This decrease is expected because the effectiveness of separator-index bounding decreases with increasing P. These experimental findings agree with the variation in the effectiveness of separator-index bounding values seen in Table 3.3. In the iterative refinement approach, MS+ is also one-to-three orders of magnitude faster than MS, where this drastic improvement simply depends on the scheme used for finding an initial leftist partition.

As Tables 3.5 and 3.6 reveal, significant improvement ratios are also obtained for the parametric search algorithms. On average, NC+ is 4.2, 3.5, 3.1, 3.7, and 3.7 times faster than NC for $P=16$, 32, 64, 128, and 256, respectively, for the SpM dataset. For the DVR dataset, if the initial prefix-sum time is not included, NC+ is 4.2, 3.2, 2.6, 2.5, and 2.7 times faster than NC for $P=16$, 32, 64, 128, and 256, respectively. For the SpM dataset, ϵBS+ is 3.4, 2.5, 1.8, 1.6, and 1.3 times faster than ϵBS for $P=16$, 32, 64, 128, and 256, respectively. These improvement ratios in the execution times of the parametric search algorithms are below the improvement ratios in the numbers of probes displayed in Table 3.4. Overhead due to the RB call and initial settings of the separator indices contributes to this difference in both NC+ and ϵBS+. Furthermore, costs of initial probes with very large bottleneck values are very cheap in ϵBS.

In Table 3.5, relative performance comparison of the proposed exact CCP algorithms shows that BID is the clear winner for small to moderate P values (i.e., $P \leq 64$) in the SpM dataset. Relative performance of BID degrades with increasing P so that both ϵBS+ and NC+ begin to run faster than BID for $P \geq 128$ in general, where ϵBS+ becomes the winner. For the DVR dataset, NC+ and EBS are clear winners as seen in Table 3.6. NC+ runs slightly faster than EBS for $P \leq 128$, but EBS runs considerably faster than NC+ for $P=256$. BID can compete with these two algorithms only for 16 and 32 way partitioning of grid *blunt-fin* (for all mesh resolutions). As seen in Table 3.5, BID takes less than 1% of a single SpMxV time for $P \leq 64$ on average.

For the DVR dataset (Table 3.6), the initial prefix-sum time is considerably larger than the actual partitioning time of the EBS algorithm in all CCP instances except 256-way partitioning of *post256*. As seen in Table 3.6, EBS is only 12% slower than the RB heuristic in 64-way partitionings on average.

These experimental findings show that the proposed exact CCP algorithms should replace heuristics.

3.6 Partitioning Chains for Heterogeneous Systems

Discussions until this section have focused on chain-partitioning algorithms for homogeneous parallel systems, where the execution time of a given task is the same on all processors. However, heterogeneous systems are becoming increasingly important because of the growing popularity of build-it-yourself parallel computers, epitomized by Beowulf-class machines [10]. Parallel systems built in this way are very likely to exhibit processor heterogeneity. Two distinct problems arise in partitioning chains for heterogeneous systems, referred to here as the chains-on-chains partitioning problem for heterogeneous systems (CCP-HET) and the chain-partitioning problem for heterogeneous systems (CP-HET). In the CCP-HET problem, a chain of tasks is to be mapped onto a chain of processors, i.e., the pth subchain in a partition is assigned to the pth processor. In the CP-HET problem, a chain of tasks is to be mapped onto a *set*—as opposed to a chain—of processors, i.e., processors can be permuted for subchain assignment. The following two sections discuss these two problems.

3.6.1 CCP Problem on Heterogeneous Systems

In the CCP-HET problem, a chain $\mathcal{T}=\langle t_1, t_2, \ldots, t_N\rangle$ of N tasks, with weights $\mathcal{W}=\langle w_1, w_2, \ldots, w_N\rangle$, is to be mapped onto a chain $\mathcal{P}=\langle \mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P\rangle$ of P processors with associated execution speeds $\mathcal{E}=\langle e_1, e_2, \ldots, e_P\rangle$. The execution time of task t_i on processor $\mathcal{P}_p$ is equal to w_i/e_p. The cost $C(\Pi)$ of a partition Π is determined by the maximum execution time among all processors. The objective is to find a partition $\Pi=\{s_0=0, s_1, \ldots, s_P=N\}$, where $s_p \leq s_{p+1}$ for $0 \leq p < P$, with minimum cost

$$C(\Pi) = \max_{1\leq p\leq P} \frac{W_{s_{p-1}+1, s_p}}{e_p}.$$

```
HPROBE(B)
  s_0 ← 0;    p ← 1;
  Bsum ← B × e_1;
  while p ≤ P and Bsum < W_tot do
    s_p ← BINSRCH (𝒲, s_{p-1}, N, Bsum);
    Bsum ← 𝒲[s_p] + B × e_p;
    p ← p + 1;
  if Bsum ≥ W_tot then
    return TRUE ;
  else
    return FALSE;
```

Figure 3.11: Probe function modified for heterogeneous processors.

All the algorithms presented in § 3.2 and 3.3 can be enhanced to solve CCP-HET optimally, without altering their complexities. For the CCP-HET algorithms, the weight of a task should be replaced with execution time of a task on a processor. We will briefly describe guidelines for some of the algorithms.

We can bound the value of an optimal solution for CCP-HET using the same techniques we did for CCP. A lower bound LB is achieved when all processors are equally loaded,

$$LB = \frac{W_{tot}}{sum_{p=1}^{P} e_p}.$$

An upper bound UB can be found in practice with a fast and effective heuristic. We can give

$$UB = LB + \frac{w_{max}}{\min_{p=1}^{P} e_p}$$

as a theoretically robust bound for many greedy heuristics. As a constructive proof for this result, assume each processor is minimally loaded to surpass LB. The worst case occurs when we have to include the maximally weighted task to surpass LB, and this happens for the slowest processor. Notice that the load of the last processor will always be less than or equal to LB.

A probe function *HPROBE*(B) for heterogeneous systems is presented in Figure 3.11. In this algorithm, the execution speed of each processor is taken into account before binary search, when we compute the load of a processor. We scale the total weight of tasks to be assigned to a processor, proportional to its execution speed. This probe function can be used to bound separator indices according to Lemma 3.3.1 and Corollary 3.3.5 as well.

The recursion given in (3.1) for the dynamic programming solution can be restated as:

$$B_i^p = \min_{p-1 \le j < i} \{\max\{B_j^{p-1}, \frac{W_{j+1,i}}{e_p}\}\}.$$

This equation defines an $O(N^2P)$ algorithm, but the observations given in § 3.2.2 for reducing the complexity to $O(NP)$ are still valid. All that needs to be done is to replace subchain weight computations $\mathcal{W}[i]-\mathcal{W}[j]$ in the DP and DP+ algorithms presented in Figures 3.1 and 3.6 with $(\mathcal{W}[i]-\mathcal{W}[j])/e_p$. It is worth noting that an optimal solution for heterogeneous systems might leave a processor empty. Thus the j index should scan the whole 1–N range as opposed to the p–$N-P+p$ range as in the case of homogeneous processors.

To enhance the bidding algorithm for heterogeneous systems, we have to be able to compute the next smallest candidate bottleneck value after a failed probe, and move the separators according to this new value. The bid of a processor $\mathcal{P}_p$ can be computed as the total chain weight with one more task added, divided by the execution speed e_p. Let $\Pi = \{s_0, s_1, \ldots, s_P\}$ be the current solution. The bid of processor $\mathcal{P}_p$ is $W_{s_{p-1}+1,s_p+1} \,/\, e_p$ for $p=1,2,\ldots,P-1$. The bid of the last processor can be computed as the remaining task weight divided by the execution speed of the last processor, i.e., $W_{s_{P-1}+1,N} \,/\, e_P$. Once we decide on the next candidate bottleneck value, we have to adjust the separator indices by adding/removing tasks to/from processors. While adding (removing) a task to (from) a processor we add (subtract) the weight of the task divided by the execution speed of the respective processor.

Lower and upper bounds, together with the *HPROBE*(B) function are sufficient for the ϵ-approximate bisection algorithm. For the exact bisection algorithm, we need to move upper and lower bounds to realizable bottleneck values. As in CCP, our probe function constructs a solution, and hence we already have a realizable upper bound when a probe succeeds. When a probe fails, we can use the bid value described in the previous paragraph to restrict the search space to realizable bottleneck values. Each probe eliminates at least one candidate bottleneck value, and thus the bisection algorithm terminates in a finite number of steps with an optimal solution.

The proposed techniques can be enhanced further to solve the CCP problem when the execution time of a task is an arbitrary function of the processor to which the task is assigned (provided that the execution time is positive). The presented algorithms can still be adapted for this problem without altering their complexities. In this case, we no longer have a definition of the weight of a task w_i, but have definition of the weight w_i^p of a task t_i for a processor $\mathcal{P}_p$ to be used for computing the load of a processor.

3.6.2 Chains Partitioning for Heterogeneous Systems

In this section, we study the problem of partitioning a chain of tasks onto a set of processors. The solution to this problem is not only separators on the task chain, but also processor-to-subchain assignments. Thus, we define a mapping $\mathcal{M}$ as a partition $\Pi = \langle s_0 = 0, s_1, \ldots, s_P = N \rangle$ of the given task chain $\mathcal{T} = \langle t_1, t_2, \ldots t_N \rangle$ with $s_p \leq s_{p+1}$ for $0 \leq p < P$, and a permutation $\langle \pi_1, \pi_2, \ldots, \pi_P \rangle$ of given set of P processors $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P\}$. According to this mapping, the pth task subchain $\langle t_{s_{p-1}+1}, \ldots, t_{s_p} \rangle$ is executed on processor $\mathcal{P}_{\pi_p}$. The cost $C(\mathcal{M})$ of a mapping $\mathcal{M}$ is the largest subchain computation time, determined by the subchain weight and the execution speed of the assigned processor, i.e.,

$$C(\mathcal{M}) = \max_{1 \leq p \leq P} \frac{W_{s_{p-1}+1, s_p}}{e_{\pi_p}}.$$

The corresponding decision problem for the chain-partitioning problem for heterogeneous systems is as follows.

Given a chain of tasks $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$, a weight $w_i \in Z^+$ for each $t_i \in \mathcal{T}$, a set of processors $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_P\}$ such that $P < N$, and an execution rate $e_p \in Z^+$ for each $\mathcal{P}_p \in \mathcal{P}$, decide if there exists a mapping $\mathcal{M}$ of chain $\mathcal{T}$ onto the set of processors $\mathcal{P}$ such that $C(\mathcal{M})$ is less than or equal to a specified value.

Lemma 3.6.1 *The chain-partition problem for heterogeneous systems is NP-Hard.*

Proof: We will use reduction from the 3-Partition (3P) problem. A pseudo-polynomial transformation suffices, because 3P problem is NP-Complete in the strong sense (i.e., there is no pseudo-polynomial time algorithm for the problem unless P=NP). The 3P problem is stated in [34] as follows.

Given a finite set $\mathcal{A}$ of $3m$ elements, a bound $B \in Z^+$, and a cost $c_i \in Z^+$ for each $a_i \in \mathcal{A}$, such that each c_i satisfies $B/4 < c_i < B/2$ and such that $\sum_{a_i \in \mathcal{A}} c_i = mB$, decide if $\mathcal{A}$ can be partitioned *into* m *disjoint sets* $S_1, S_2, \ldots, S_m$ *such that* $\sum_{a_i \in S_p} c_i = B$ for $p = 1, 2, \ldots, m$.

For a given instance of the 3P problem, the corresponding chain-partitioning problem is constructed as follows.

- The number of tasks N is $m(B+1)-1$. The weight of every $(B+1)$st task is B, (i.e., $w_i = B$ for $i \bmod (B+1) = 0$), and the weights of all other tasks are 1.
- The number of processors P is $4m-1$. The first $m-1$ processors have execution speeds of B, (i.e., $e_p = B$ for $p = 1, 2, \ldots, m-1$), and remaining processors have execution speeds equal to the costs of items in the 3P problem (i.e., $e_p = c_{p-m+1}$ for $p = m, \ldots, 4m-1$).

We claim that there is a solution to the 3P problem if and only if there is a mapping $\mathcal{M}$ with cost $C(\mathcal{M}) = 1$ for the chain-partitioning problem. The following observations constitute the basis for our proof.

- The processors with execution speeds of B must be mapped to tasks with weight B to have a solution with cost $C(\mathcal{M}) = 1$, because the execution speeds of all other processors are $\leq B/2$. These processors (tasks) serve as divider processors (tasks).
- The total weight of the chain is $3m + (m-1)B = (B+3)m - B$. The sum of execution speeds of all processors is also $(m-1)B + 3m = (B+3)m - B$. This forces each processor to be assigned a load with value equal to its execution speed, to achieve a mapping with cost $C(\mathcal{M}) = 1$. To generalize: If a subchain of tasks with total weight W is mapped to a subset of processors with total execution speed E, then $E = W$ to have a solution with $C(\mathcal{M}) = 1$. Notice that the other direction of the proposal also holds due to unit-weight tasks between dividers.

As noted above, the divider processors should be assigned to the divider tasks. Between two successive divider tasks there is a subchain of B unit-weight tasks with total weight B, which must be assigned to a subset of processors with total execution speed B. Since there are m such subchains, the same grouping is also valid for grouping c_i values in the 3P problem.

□

Theorem 3.6.2 *Chain-partition problem for heterogeneous systems is NP-Complete.*

Proof: The cost of a given mapping can be computed in polynomial time, thus the problem is in NP. With Lemma 3.6.1, we can conclude that the chain-partition problem for heterogeneous systems is NP-Complete. □

3.7 Conclusions

We proposed runtime efficient chains-on-chains partitioning algorithms for optimal load balancing in one-dimensional decomposition of nonuniform computational domains. Our main idea was to run an effective heuristic, as a pre-processing step, to find a "good" upper bound on the optimal bottleneck value, and then exploit lower and upper bounds on the optimal bottleneck value to restrict the search space for separator-index values. This separator-index bounding scheme was exploited in a static manner in the dynamic-programming algorithm, drastically reducing the number of table entries computed and referenced. A dynamic separator-index bounding scheme was proposed for parametric search algorithms to narrow separator-index ranges after each probe. We enhanced the approximate bisection algorithm to be exact by updating lower and upper bounds into realizable values after each probe. We proposed a new iterative-refinement scheme, that is very fast for small-to-medium numbers of processors. We also showed that the proposed algorithms can be used for heterogeneous parallel systems with minor modifications, and proved NP-Completeness of the chain-partitioning problem for heterogeneous systems, where processor permutation is allowed in subchain-to-processor assignment.

We investigated two distinct applications for experimental performance evaluation: 1D decomposition of irregularly sparse matrices for parallel matrix-vector multiplication, and decomposition for image-space parallel volume rendering. Experiments on the sparse matrix dataset showed that 64-way decompositions can be achieved in 100 times less time than a single matrix vector multiplication time, while producing 4 times better load imbalance values than the most effective heuristic, on average. Experimental results on the volume rendering dataset showed that exact algorithms can produce 3.8 times better 64-way decompositions than the most effective heuristic, while being only 11% slower, on average.

Acknowledgments

This work has been performed under the supervision of Cevdet Aykanat from Bilkent University. We also appreciate Tahsin Kurç's provision of some of our test problems.

Chapter 4

A Divide–and–Conquer Algorithm for Identifying Strongly Connected Components

A *strongly connected component* of a directed graph is a maximal subset of vertices containing a directed path from each vertex to all others in the subset. The vertices of any directed graph can be partitioned into a set of disjoint strongly connected components. This decomposition is a fundamental tool in graph theory with applications in compiler analysis, data mining, scientific computing, and other areas.

The definitive serial algorithm for identifying strongly connected components, due to Tarjan [92], is built on a depth first search of the graph. For a graph with n vertices and m edges, this algorithm runs in the optimal $O(m)$ time and is widely used in textbooks as an example of the power of depth first search [18].

For large problems, a parallel algorithm for identifying strongly connected components would be useful. One application of particular interest to us is discussed below. Unfortunately, depth first search (DFS) seems to be difficult to parallelize. Reif shows that a restricted version of the problem (lexicographical DFS) is $\mathcal{P}$-Complete [87]. However, Aggarwal and Anderson, and Aggarwal et al. describe randomized NC algorithms for finding a DFS of undirected and directed graphs, respectively [2, 3]. The expected running time of this latter algorithm is $O(\log^7 n)$ and it requires an impractical $n^{2.376}$ processors. To our knowledge, the deterministic parallel complexity of DFS for general, directed graphs is an open problem. Chaudhuri and Hagerup studied the problem for acyclic [15], and planar graphs [40], respectively, but our interest is in general graphs.

More practically, DFS is a difficult operation to parallelize and we are aware of no algorithms or implementations that perform well on large numbers of processors. Consequently, Tarjan's algorithm is not suitable for finding strongly connected components in parallel.

Alternatively, there are several parallel algorithms for the strongly connected components problem (SCC) that avoid the use of depth first search. Gazit and Miller devised an NC algorithm for SCC, which is based upon matrix–matrix multiplication [37]. This algorithm was improved by Cole and Vishkin [17], but still requires $n^{2.376}$ processors and $O(\log^2 n)$ time. Kao developed a more complicated NC algorithm for planar graphs that requires $O(\log^3 n)$ time and $n/\log n$ processors [58]. More recently, Bader proposed an efficient parallel implementation of SCC for planar graphs [8], which uses a clever *packed–interval* representation of the boundary of a planar graph. When n is much larger than the number of processors p, Bader's approach scales as $O(n/p)$. But Bader's approach does not apply to general graphs.

Our interest in the SCC problem is motivated by the discrete ordinates method for modeling radiation transport. Using this methodology, the object to be studied is modeled as a union of polyhedral finite elements. Each element is a vertex in our graph and an edge connects any pair of elements that share a face. The radiation equations are approximated by an angular discretization. For each angle in the discretization, the edges in the graph are directed to align with the angle. The computations associated with an element can be performed if all its predecessors have been completed. Thus, for each angle, the set of computations is sequenced as a topological sort of the directed graph. A problem arises if the topological sort cannot be completed—i.e., the graph has a cycle. If cycles exist, the numerical calculations must be modified—typically by using old information along one of the edges in the cycle, thereby removing the dependency. Thus, identifying strongly connected components quickly is essential. Since radiation transport calculations are computationally and memory intensive, parallel implementations are necessary for large problems. Also, since the geometry of the grid can change after each time step for some applications, the SCC problem must be solved in parallel.

Efficient parallel implementations of the topological sort step of the radiation transport problem have been developed for *structured* grids, oriented grids that have no cycles [9, 26]. Some initial attempts to generalize these techniques to unstructured grids show promise [74, 84]. It is these latter efforts that motivated our interest in the SCC problem.

In the next section we describe a simple divide-and-conquer algorithm for finding strongly connected components. In §4.2 we show that this algorithm has an expected serial complexity of $O(m \log n)$. In §4.3 we present a modification of the algorithm and show that it has worst case serial complexity of $O(m \log n)$. This modification has been designed by Lisa Fleischer, and we are including it in this thesis for completeness.

Our approach has good potential for parallelism for two reasons. First, the divide–and–conquer paradigm generates a set of small problems that can be solved independently by separate processors. Second, the basic step in our algorithm is a reachability analysis, which is similar to topological sort in its parallelizability. So we expect the current techniques for parallelizing radiation transport calculations to enable our algorithm to perform well too. A preliminary version of this work can be found in [30], and a journal version has been submitted as [31].

4.1 A Divide–and–Conquer Algorithm

Before describing our algorithm, we introduce some notation. Let $G = (V, E)$ be a directed graph with vertices V and directed edges E. An edge $(i, j) \in E$ is directed from i to j. We denote the set of strongly connected components of G by $SCC(G)$. Thus $SCC(G)$ is a partition of V. We also use $SCC(G, v)$ to denote the (unique) strongly connected component containing vertex v. We denote by $V \setminus X$ the subset of vertices in V that are not in a subset X. The size of vertex set X is denoted by $|X|$.

A vertex v is *reachable* from a vertex u if there is a sequence of directed edges $(u, x_1), (x_1, x_2), \ldots,$ (x_k, v) from u to v. We consider a vertex to be reachable from itself. Given a vertex $v \in V$, the *descendants* of v, $Desc(G, v)$, is the subset of vertices in G that are reachable from v. Similarly, the *predecessors* of v, $Pred(G, v)$, is the subset of vertices from which v is reachable. The set of vertices that are neither reachable from v nor reach v is called the *remainder*, denoted by
$Rem(G, v) = V \setminus (Desc(G, v) \cup Pred(G, v))$.

Given a graph $G = (V, E)$ and a subset of vertices $V' \subseteq V$, the *induced subgraph* $G' = (V', E')$ contains all edges of G connecting vertices of V', i.e., $E' = \{(u, v) \in E : u, v \in V'\}$. We will use $\langle V' \rangle = G' = (V', E')$ to denote the subgraph of G induced by vertex set V'. The following lemma is an immediate consequence of the definitions.

Lemma 4.1.1 *Let $G = (V, E)$ be a directed graph, with $v \in V$ a vertex in G. Then $Desc(G, v) \cap Pred(G, v) = SCC(G, v)$.*

Lemma 4.1.2 *Let G be a directed graph with vertex v. Any strongly connected component of G is a subset of $Desc(G, v)$, a subset of $Pred(G, v)$ or a subset of $Rem(G, v)$.*

Proof: Let u and w be two vertices in the same strongly connected component of G. By definition, u and w are reachable from each other. The proof involves establishing $u \in Desc(G, v) \iff w \in Desc(G, v)$ and $u \in Pred(G, v) \iff w \in Pred(G, v)$, which then implies $u \in Rem(G, v) \iff w \in Rem(G, v)$. Since the proofs of these two statements are symmetric, we give just the first: If $u \in Desc(G, v)$ then u must be reachable from v. But then w must also be reachable from v, so $w \in Desc(G, v)$. $\square$

With this background, we can present our algorithm, which we call DCSC (for Divide–and–Conquer Strong Components). The algorithm is sketched in Figure 4.1. The basic idea is to select a random vertex v, which we will call a *pivot* vertex, and find its descendant and predecessor sets. The intersection of these sets is $SCC(G, v)$ by Lemma 4.1.1. After this step, the remaining vertices are divided into three sets $Desc(G, v)$, $Pred(G, v)$, and $Rem(G, v)$. By Lemma 4.1.2, any additional strongly connected component must be entirely contained within one of these three sets, so we can divide the problem and recurse.

$DCSC(G)$
    If G is empty **then Return**;
    Select v uniformly at random from V;
    $SCC \leftarrow Pred(G, v) \cap Desc(G, v)$;
    Output SCC;
    $DCSC(\langle Pred(G, v) \setminus SCC \rangle)$;
    $DCSC(\langle Desc(G, v) \setminus SCC \rangle)$;
    $DCSC(\langle Rem(G, v) \rangle)$;

Figure 4.1: Divide–and–conquer algorithm for strongly connected components.

As mentioned above, this algorithm is amenable to practical parallelization on two levels. First, the recursive invocations are completely independent and so can execute independently. Second, the searches for predecessors and descendants allow for much more parallelism than does a depth first search.

4.2 Serial Complexity of Algorithm DCSC

The cost of algorithm DCSC can be described by four terms, one each from the three recursive invocations and a fourth from the determination of predecessors and descendants. Let $n(G)$ and $m(G)$ denote the number of vertices and edges in graph G, respectively. We will use n and m to denote the number of vertices and edges in the original graph, when it is clear from the context. Let $T(G)$ be the runtime of the algorithm on a graph G. For a pivot vertex i, let $P_i = Pred(G,i) \setminus SCC(G,i)$, $D_i = Desc(G,i) \setminus SCC(G,i)$ and $R_i = Rem(G,i)$ be the graphs for the recursive calls. The recursive expression for the runtime is

$$T(G) = T(R_i) + T(D_i) + T(P_i) + \Theta(m(G) - m(R_i)). \tag{4.1}$$

Clearly, $T(G) = \Omega(m+n)$, since we eventually must look at all vertices and edges. Also, in the worst case $T(G) = O(mn)$, since each iteration takes at most linear time in the number of edges and reduces the number of vertices by at least 1. We show here that the expected behavior of $T(G)$ is $O((m+n)\log n)$. We will accomplish this by showing that each edge is visited $O(\log n)$ times on average. The expected case analysis will require bounds on the sizes of the predecessor and descendant sets. The following two results provide these bounds.

Lemma 4.2.1 *For a directed graph G, there is a numbering π of the vertices from 1 to n in which the following is true. All elements $u \in Pred(G,v) \setminus Desc(G,v)$ satisfy $\pi(u) < \pi(v)$; and all elements $u \in Desc(G,v) \setminus Pred(G,v)$ satisfy $\pi(u) > \pi(v)$.*

Proof: If G is acyclic, then a topological sort provides a numbering with this property. If G has cycles, then each strongly connected component can be contracted into a single vertex, and the resulting acyclic graph can be numbered via topological sort. Assume a strongly connected component with k vertices was assigned a number j in this ordering. Assign the vertices within the component the numbers $(j, \ldots, j+k-1)$ arbitrarily and increase all subsequent numbers by $k-1$. $\square$

It is important to note that we do not need to construct an ordering with this property; we just need to know that it exists.

Corollary 4.2.2 *Given a directed graph G and a vertex numbering π from Lemma 4.2.1, then $n(Pred(G,v) \setminus SCC(G,v)) < \pi(v)$ and $n(Desc(G,v) \setminus SCC(G,v)) \leq n(G) - \pi(v)$ for all vertices v.*

Theorem 4.2.3 *Algorithm DCSC has expected time complexity $O((m+n)\log n)$.*

Proof: We will bound the expected number of visits to an edge by $\log n$, and the theorem follows. Edges in the predecessor set P and descendant set D are visited a constant c times at each recursive invocation of DCSC, and edges in R are not visited. Let $X(G)$ be the average number of times an edge is visited in algorithm DCSC. Then

$$X(G) = \frac{1}{m}\{m_p(X(P)+c) + m_d(X(D)+c) + m_r X(R) + m_s c\}, \tag{4.2}$$

where m_p, m_d, and m_r, respectively, are the number of edges in the predecessor, descendant, and remainder sets, and m_s is the number of edges in the strongly connected component and edges between different components.

Since DCSC is a randomized algorithm, we will perform an expected case analysis by summing the cost over all n possible pivot vertices and dividing by n. We will show that the expected value of $X(G)$, $EX(G) \leq a\log n$ for some constant a. As the inductive hypothesis, we will assume this bound holds for all graphs with fewer than n vertices.

$$\begin{aligned}
EX(G) &= \frac{1}{n}\sum_{1\leq i\leq n}\frac{1}{m}\{m_p(EX(P_i)+c) + m_d(EX(D_i)+c) + m_r EX(R_i) + m_s c\} \\
&\leq \frac{1}{n}\sum_{1\leq i\leq n}\frac{1}{m}(m_p + m_d + m_r + m_s)\max\{EX(P_i)+c, EX(D_i)+c, EX(R_i), c\} \\
&\leq \frac{1}{n}\sum_{1\leq i\leq n}\max\{EX(P_i)+c, EX(D_i)+c, EX(R_i)\}.
\end{aligned}$$

Using the inductive hypothesis and Corollary 4.2.2, we find that

$$EX(n) = \frac{1}{n}\sum_{1\leq i\leq n}\max\{a\log i + c, a\log(n-i)+c, a\log n\}.$$

Using symmetry,

$$EX(n) = \frac{2}{n} \sum_{n/2 \le i \le n} \max\{a \log i + c, a \log n\}.$$

The first term dominates when $i > \frac{n}{2^{\frac{c}{a}}}$. Let $d = \frac{1}{2^{\frac{c}{a}}}$. We will choose a so that $1/2 \le d \le 1$.

$$\begin{aligned}
EX(n) &= \frac{2}{n} \sum_{n/2 \le i \le n} \max\{a \log i + c, a \log n\} \\
&= \frac{2}{n} (\sum_{n/2 \le i < dn} \max\{a \log i + c, a \log n\} + \sum_{dn \le i \le n} \max\{a \log i + c, a \log n\}) \\
&= \frac{2}{n} (\sum_{n/2 \le i < dn} a \log n + \sum_{dn \le i \le n} a \log i + c). \qquad (4.3)
\end{aligned}$$

To bound the second term, we will split the summation into three pieces, each of span $(1-d)n/3$.

$$\begin{aligned}
\sum_{dn \le i \le n} a \log i + c &= \sum_{dn}^{\frac{2d+1}{3}n} (a \log i + c) + \sum_{\frac{2d+1}{3}n}^{\frac{d+2}{3}n} (a \log i + c) + \sum_{\frac{d+2}{3}n}^{n} (a \log i + c) \\
&\le \sum_{dn}^{\frac{2d+1}{3}n} (a \log(\frac{2d+1}{3}n) + c) + \sum_{\frac{2d+1}{3}n}^{\frac{d+2}{3}n} (a \log(\frac{d+2}{3}n) + c) + \sum_{\frac{d+2}{3}n}^{n} (a \log n + c) \\
&\le \frac{(1-d)n}{3}(a \log n + a \log \frac{2d+1}{3} + c) + \frac{(1-d)n}{3}(a \log n + a \log \frac{d+2}{3} + c) + \\
&\quad \frac{(1-d)n}{3}(a \log n + c) \\
&\le (1-d)na \log n + \frac{(1-d)an}{3}(\log \frac{2d+1}{3} + \log \frac{d+2}{3} + \frac{3c}{a}).
\end{aligned}$$

We want $\log \frac{2d+1}{3} + \log \frac{d+2}{3} + \frac{3c}{a} \le 0$, or equivalently,

$$\begin{aligned}
\log \frac{3}{2d+1} + \log \frac{3}{d+2} &\ge \frac{3c}{a} \\
\frac{9}{(2d+1)(d+2)} &\ge (2^{\frac{c}{a}})^3 \\
\frac{9}{2d^2 + 3d + 2} &\ge \frac{1}{d^3} \\
9d^3 &\ge 2d^2 + 3d + 2,
\end{aligned}$$

which holds for $d = \frac{8}{9}$. Setting $a = \frac{c}{\log \frac{9}{8}}$, we find that

$$\sum_{dn \leq i \leq n} a \log i + c \leq (1-d) n a \log n. \tag{4.4}$$

Combining equations (4.3) and (4.4),

$$\begin{aligned} EX(n) &= \frac{2}{n}\Big(\sum_{n/2 \leq i < dn} a \log n + \sum_{dn \leq i \leq n} a \log i + c\Big) \\ &\leq \frac{2}{n}\Big\{\frac{(2d-1)n}{2} a \log n + (1-d) n a \log n\Big\} \\ &= \frac{2}{n}\frac{n}{2} a \log n \\ &= a \log n. \end{aligned}$$

The analysis above assumes that $m > n$. If this is not the case, we can use the same analysis to count the expected number of vertex visits to achieve an $O(n \log n)$ bound for the algorithm.

$\square$

4.3 A Worst Case $O(m \log n)$-time Algorithm

In this section, we modify the algorithm from §4.1 to achieve a worst case $O(m \log n)$ time complexity. The basic idea of the modification is to avoid completing long searches for predecessors and descendants. Assume without loss of generality that the predecessor set has fewer edges than the descendant set. If we knew this, we could first identify all the predecessors, and then limit the successor search to visit only predecessor vertices. The intersection of these two searches is a strongly connected component. Once this is done, we can recurse by dividing the graph into two pieces. The first piece is the predecessor set minus the strongly connected component. The second set is the full graph minus the predecessors. By Lemma 4.1.2, there can be no strongly connected components that cross between these two sets. In practice, we don't know whether we should search for predecessors or successors first. So instead, we will search for both simultaneously and abandon the second search when the first one finishes. The resulting algorithm is sketched in Figure 4.2.

Theorem 4.3.1 *Algorithm WDCSC presented in Figure 4.2 runs in worst case $O(m \log n)$-time.*

```
WDCSC(G)
    If G is empty then Return;
    Select v uniformly at random from V;
    Simultaneously search for Pred(G, v) and Desc(G, v);
    If Pred(G, v) finishes first
        Search for Desc(G, v) only in Pred(G, v);
    Else If Desc(G, v) finishes first
        Search for Pred(G, v) only in Desc(G, v);
    SCC ← Pred(G, v) ∩ Desc(G, v);
    Output SCC;
    If Pred(G, v) finished first
        WDCSC(⟨Pred(G, v) \ SCC⟩);
        WDCSC(⟨V \ Pred(G, v)⟩);
    Else If Desc(G, v) finished first
        WDCSC(⟨Desc(G, v) \ SCC⟩);
        WDCSC(⟨V \ Desc(G, v)⟩);
```

Figure 4.2: A worst case $O(m \log n)$ time algorithm to find strongly connected components

Proof: Let $R(m)$ be the run time of this algorithm on a graph with m edges. We will use p_i, d_i and s_i to denote the total number of edges adjacent to $P_i \setminus SCC(G,i)$, $D_i \setminus SCC(G,i)$ and $SCC(G,i)$, respectively. By symmetry, we can assume $p_i \leq d_i$. Then $R(m)$ can be expressed as

$$R(m) \leq R(p_i) + R(m - p_i - s_i) + \Theta(p_i + s_i), \tag{4.5}$$

since the amount of work in one iteration is proportional to the size of the minimum of the predecessor set and the descendant set. That is, the amount of work is twice $p_i + s_i$ until the algorithm completes one search, and then is at most s_i. Note that by hypothesis $p_i \leq m/2$.

We will show that $R(m) = O(m \log m)$ by induction, considering two cases depending on the size of p_i. Since $m = O(n^2)$, this leads to our claimed bound.

Case 1: $p_i + s_i \leq m/2$.

If $p_i \leq m/2$, then we can upper bound $R(m)$ by substituting $a = p_i + s_i$ in (4.5) to get

$$R(m) \leq R_2(m,a) := R(a) + R(m-a) + \Theta(a).$$

By our assumptions in this case, we have that $1 \leq a \leq m/2$. We show that this recursion is $O(m \log m)$ by first showing that this holds for $a = 1, m/2$, and then showing that R_2, as a function of a in the range $[1, m-1]$, is convex. Thus, its value in an interval is bounded from above by its value at the endpoints of the interval.

For $a = 1$, we have that $T_2(m) = \Theta(m)$. When $a = m/2$, the recursion becomes $R(m) = 2R(m/2) + \Theta(m)$, which is well-known to lead to $R(m) = O(m \log m)$. We suppose, by induction, that $R(r) = c_1 m \log m$ for an appropriate constant c_1 for $r < m$, and that $\Theta(r) = c_2 r$. Thus, the first derivative of $R_2(m, a)$ with respect to a is

$$c_1(\log a - \log(m - a)) + c_2.$$

The second derivative is

$$c_1(\frac{1}{a} + \frac{1}{m-a}),$$

which is positive for $a \in [1, m-1]$. Thus, $R_2(m, a)$ is convex for $a \in [1, m/2]$, and hence $R(m) = O(m \log m)$ in this case.

Case 2: $p_i + s_i > m/2$.

In this case, we can bound $R(m)$ by

$$\begin{aligned} R(m) &= R(p_i) + R(m - p_i - s_i) + \Theta(p_i + s_i) \\ &\leq 2R(m/2) + \Theta(m). \end{aligned}$$

since $p_i \leq m/2$. As before, we have a well–known recursion whose solution is $R(m) = O(m \log m)$.

$\square$

4.4 Conclusions

The simple divide–and–conquer algorithm described in 4.1 has recently been implemented by McLendon, et al. [66], and good parallel performance is observed for problems arising from the radiation transport application described at the beginning of this chapter. Several open questions remain. Despite our conviction that our divide–and–conquer approach is amenable to effective parallelization, we have not shown that it leads to an improved NC algorithm. Specifically, the reachability analyses at the core of our method can be solved in NC via Gazit and Miller's matrix product approach [37], but their technique can be used to find strongly connected components directly. Another question of interest to us is the potential for a linear time algorithm for strongly connected components that does not rely on depth first search. Any such algorithm could be a candidate for an improved parallel approach.

Acknowledgments

We benefited from general discussions about algorithms for parallel strongly connected components with Steve Plimpton, Will McLendon and David Bader. We are also indebted to Bob Carr, Cindy Phillips, Bill Hart and Sorin Istrail for discussions about the analysis.

Chapter 5

Communication Support for Adaptive Computation

This chapter addresses two problems associated with redistributing data among processors. The first problem is about determining the interprocessor communication pattern necessary after new subdomains are assigned. The second problem is about actually migrating data to implement a remapping subject to memory constraints. Partial results of this work have been reported in [80, 81].

5.1 Determining Communication Patterns

This section considers the problem of determining the communication pattern after each processor is assigned a portion of a problem. We assume that each processor can determine what data it needs to receive, but does not know which processor owns that data. This problem can arise in various applications. As a concrete example we will consider the very important case of matrix–vector multiplication, where the matrix is partitioned by rows. To perform matrix–vector multiplication a processor can examine its rows, find all the columns for which it has a nonzero and thus identify the components of the vector that it needs. But it does not know which processors own these components. To perform the communication, each processor must know what data it will receive from and what data it will send to every other processor. The purpose of this work is to design efficient algorithms to determine this communication pattern.

A solution to this problem is useful for two reasons. First, the alternative is to have the partitioner (or some other code) determine this information and to have it included in the problem description. This alternative burdens the routines that set up the problem and adds additional complexity to software interfaces. Providing an efficient parallel solution to this problem simplifies the use of parallel computers. Second, dynamic load balancing redistributes tasks amongst processors, and so changes the communication pattern. Although the repartitioning tool could carry some data along with it to facilitate determination of the new pattern, a good solution to the general problem will relieve the partitioner of this responsibility.

A simple solution to the problem of determining the communication pattern is to have each processor exchange messages with all others. For instance, each processor can create a list of the items it needs and the processors can pass these lists around a ring. When a processor receives a list, it checks to see if it owns the requested items. Unfortunately, this all–to–all communication is slow and not scalable. It is possible to employ higher-dimensional variants of the ring algorithm, such as a 2D-torus algorithm where we first run the ring algorithm on subsets of processors to aggregate lists and then exchange aggregated lists with the other processors. But these variants still perform all–to–all communication, only in a somewhat more efficient way. They also require more memory on each processor. In this work, we propose a rendezvous algorithm, called the *distributed directory* algorithm, which avoids excessive communication while still requiring only a small amount of memory.

5.1.1 Solution Methods

In this section we will discuss the *ring* algorithm, its extensions to higher dimensions, and the distributed directory algorithm we propose.

The ring algorithm circulates queries among processors in a ring fashion. At each step i, a processor p

(1) receives a list from processor $p-1$,

(2) processes the list to determine what data it must send to $p-i$,

(3) sends the remainder of this list to processor $p+1$.

For P processors, after $P-1$ steps, every processor will have seen every other query and complete sending information—what each processor has to send to every other processor—will be generated. We need the receiving information as well, which is easy to generate given the sending information. This algorithm requires each processor to have memory to store two lists at a time.

To decrease the number of steps, we can use higher-dimensional variants of the ring algorithm. For instance, in 2D-torus algorithm, we can think of the processors as being assigned to a 2D grid (even if the actual network topology is different). We can run the ring algorithm on each row of the grid, aggregating lists. These larger lists are then circulated via a ring algorithm in each column of the grid. This requires two ring algorithms of $\sqrt{P}-1$ steps, but the lengths of queries in the second step can be $\sqrt{P}$ times larger, since it is a collection of queries from all processors in a row. We can further increase the dimensions, and go all the way up to hypercube or even all-to-all-communication in one step to further trade increased message size for fewer steps. In a k-dimensional torus algorithm each processor,

(1) uses the ring algorithm to aggregate lists from processors that have the same k dimension as itself,

(2) processes these lists to generate sending information,

(3) makes a recursive call by replacing dimensions as $k-1$ and replacing its own list with the collective list.

In general, a k-dimensional torus algorithm requires k ring operations, where each ring has $P^{1/k}$ processors, so the total number of steps is $kP^{1/k}$. Unfortunately, the final steps involve an aggregation of $P^{(k-1)/k}$ individual lists. Going to higher dimensions increases not only the memory requirement, but also the sizes of messages being transferred.

To avoid moving excessive volumes of data, we propose using a distributed *directory*, which is a rendezvous algorithm. In this algorithm each processor is responsible for maintaining directory information for a subset of the components. The algorithm is presented in Figure 5.1.

The algorithm assumes that each item has some unique identifier associated with it, which can be used to determine a directory processor that will be informed about the item. This directory processor is the meeting place where the owner and the requesters can *rendezvous*. Throughout the algorithm, processors work both as clients, who have queries to be resolved and directory servers that answer queries.

The algorithm begins by creating a distributed data structure for answering queries. The directory is distributed across the same set of processors as those that want to figure out the communication pattern. First, each processor identifies its items that might be queried by an-

Directory Creation:
As a client:
Identify directory owner for all my items that might be queried;
Submit these items to directory;
As a directory:
Receive Submissions;
Build hash table of submitted lists;

Query Resolution:
As a client:
Identify directory owner of items I need;
Submit requests to directory;
As a directory:
Receive queries;
Check hash table to determine owners of requested items;
Inform requesters of results;
As a client:
Receive query results;

Figure 5.1: Distributed directory algorithm

other processor. For each such item, it determines which directory processor should own it via a hash function. After sending its data to directory processors, each processor works as part of the directory, collecting submitted lists and organizing them into a data structure to answer queries.

With this directory, query resolution is straightforward. Each processor identifies the items it needs to know the owner of. For each such item, it determines the directory processor that owns it via the same hash function used to create the directory. As a directory processor, it then receives queries, looks up the owner of the requested data, and returns the answer to the requester.

One of the critical issues for performance is to limit the size of the directory. We want to maintain the directory for only those items that will be queried. For symmetric matrices (or symmetric communication in general) rows that need to be queried are those rows that require communication for themselves (equivalently, the boundary vertices on the graph of the matrix). Thus, only items that will actually be queried need to be placed in the directory.

If communication is not symmetric, it is not possible to determine the minimal set of items that need to be submitted to the directory. One solution would be for processors to submit their entire domains to the directory. However, typically only a very small portion of a processor's domain will be queried, thus submitting the whole domain is inefficient, especially as the domain size for each processor grows. Notice that the ring algorithm and its variants do not suffer from this problem, and they can be adopted without modifications. In practice however, unsymmetric matrices often have a lot of symmetry in them. We can exploit this observation to reduce the size of the problem by first using the directory algorithm. The directory algorithm will resolve queries associated with the symmetric matrix structure communication. We can use a ring–variant algorithm to resolve any remaining queries (i.e., items with unsymmetric communication). As the experiments in the next section reveal, the time to run the directory algorithm is much smaller than all ring variants. Thus, we can afford a preprocessing step that reduces the problem size to improve the efficiency of the ring algorithm.

5.1.2 Experimental Results

We have tested our algorithms for determining communication pattern for the *ocean* matrix, a symmetric matrix of dimension 143,437 with 962,623 nonzeros. We ran our experiments on the ASCI Red parallel computer at Sandia National Labs. Our results are presented in Figure 5.2 for 4, 16, 64, 256, and 1024 processors. The matrix was partitioned using the multilevel algorithm in Chaco. For these experiments the items (matrix rows) had a global numbering that we used as the identifier. For a hash function, we simply assigned the first N/P items to the first directory processor. The number of the rows of the matrix contains some implicit locality structure that this hash function exploits. Thus, a processor probably communicates with only a subset of directory processors. A random hash function might improve load balance in the directory, but it would likely increase the number of messages involved in constructing and querying the directory.

As expected, the directory algorithm scales well with increasing number of processors whereas performance of the ring algorithm deteriorates. The 2D algorithm performs better than the 1D ring, but still is not scalable. Further increasing the dimensions of the ring algorithm slightly improves performance, but the gain that can be achieved by increasing number of processors is limited, as proved by the similarity between the performances of the 3D and 4D versions for 256 and 1024 processors. Recall that these ring variants also require more space than the distributed directory. We have also experimented with MPI's all-to-all communication routine (`MPI_Alltoallv`), since

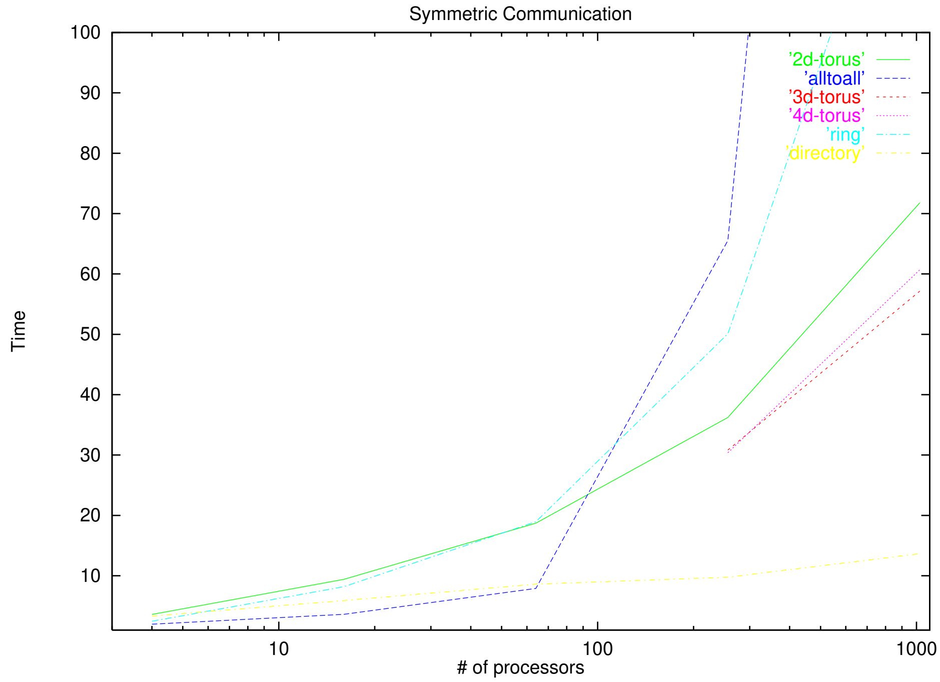


Figure 5.2: Performance of different algorithms for symmetric communication

an MPI level implementation might be more efficient. For small number of processors, MPI calls perform well, but as the number of processors increases, its runtime explodes.

Our experimental results for unsymmetric matrices are presented in Figure 5.3. The results show that submitting the whole domain can be an efficient solution for the problem. Reducing the problem size by using the directory algorithm does not help much since the overhead due to setup times dominates the costs. As in the symmetric case, using the 2D-torus algorithm performs better than the ring algorithm, but its scalability is limited.

5.2 Memory-Constrained Interprocessor Communication

In many parallel computations, the workload needs to be periodically redistributed among the processors. On a distributed memory computer, this generally requires data structures associated with the computations to be transferred between processors. Many examples of this phenomena occur in scientific computing, including adaptive mesh refinement, particle simulations with short- or long-range forces, state-dependent physics models, and multi-physics or multi-phase simulations.

A number of algorithms and software tools have been developed to repartition the work among

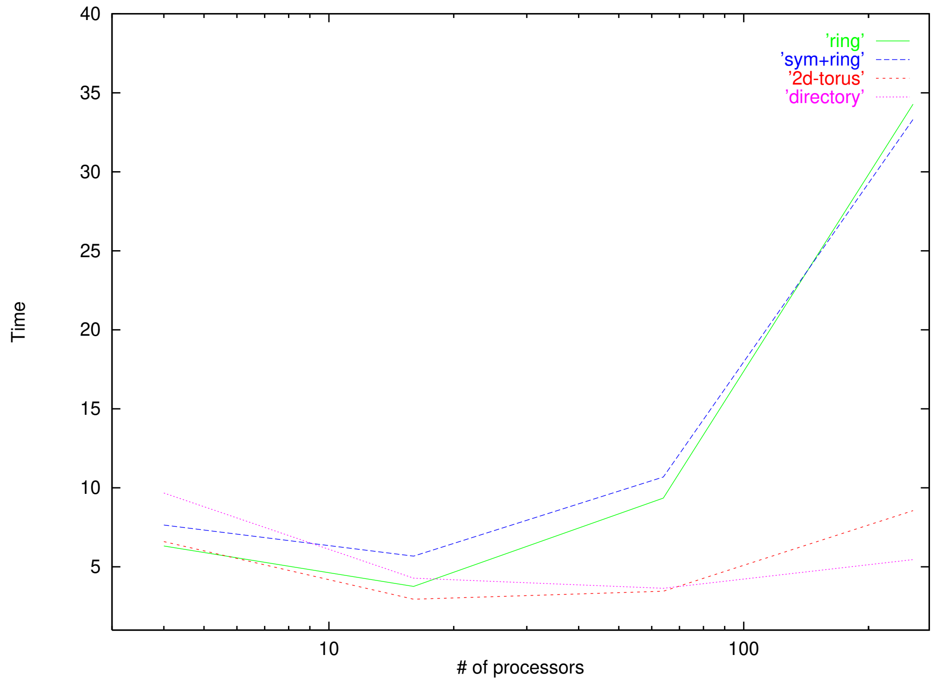


Figure 5.3: Performance of different algorithms for unsymmetric communication

processors (see for example, [20, 44] and references therein). However, the mechanics of actually moving large amounts of data has received much less attention. When the processors have sufficient memory, the simplest way to transmit the data is quite effective. Each processor can execute the following steps.

(1) Allocate space for my incoming data,

(2) Post an asynchronous receive for my incoming data,

(3) Barrier,

(4) Send all my outgoing data,

(5) Free up space consumed by my outgoing data,

(6) Wait for all my incoming data to arrive.

The barrier in step (3) ensures that no messages arrive until the processor is ready to receive them, so no buffering is needed.

Unfortunately, this protocol can fail when memory is limited. It requires a processor to have sufficient memory to hold both the outgoing and the incoming data simultaneously, since incoming messages can arrive before space for outgoing data is freed. An alternative way to view this issue is that for a period of time, the data being transferred consumes space on both the sending and receiving processors. A protocol that alleviates this problem is desirable for three reasons. First, since many scientific calculations are memory limited, reserving space for this communication operation limits the size of the calculations that can be performed. Second, the amount of memory required by this protocol is unpredictable, because the data remapping requirements depend upon the computation. Thus, one is forced to set aside a conservative amount of space, which is likely to be wasteful. Third, a general purpose tool for dynamic load balancing should be robust in the presence of limited memory. The construction of such a tool inspired our interest in this problem [23].

To address these problems, we propose a simple modification to the above scheme. Instead of sending all of the data at once, we will send it in phases. After each phase, processors can free up the memory of the data they have sent. That memory is now available for the next communication phase. Since each phase can be expensive, it is important to limit the total number of phases.

More formally, consider a set of P processors. The amount of data that needs to be communicated between processors is a *transfer request*. We will assume that the request is *feasible* – that the end result of satisfying the transfer request does not violate any processor's memory constraints. We will let T_{ij} denote the total volume of data to be transferred from processor i to processor j.

We now wish to perform the requested transfer in a sequence of phases. Let t_{ij}^l denote the volume of data transfer from processor i to processor j in phase l, and let A_i^l be the memory available to processor i at the beginning of phase l. We will also use R_i^l and S_i^l to denote the total volume of data received and sent by processor i in phase l (i.e., $R_i^l = \sum_{j=1}^k t_{ji}^l$ and $S_i^l = \sum_{j=1}^k t_{ij}^l$).

At each step, the constraint of finite memory requires that $R_i^l \leq A_i^l$ for $i = 1, 2, \dots k$. The available memory after each phase can be computed as $A_i^{l+1} = A_i^l + S_i^l - R_i^l$. Our objective is to find a schedule of transfers that obeys the memory constraint and satisfies the transfer request in a minimal number of phases. We will call this the *minimum phase remapping problem.* Note that there is a corresponding decision problem: can a given transfer request be completed in a specified number of phases?

In §5.2.1 we show that the problem of determining whether a given transfer can be completed in a specified number of phases is NP-Complete. The remainder of this section focuses on formulations and approximation algorithms that could be used in practice. In §5.2.2 we present a reduction of our problem to multi-commodity flow. We present a continuous relaxation of the problem in §5.2.3, and a practical approximation algorithm in §5.2.4. Finally, we discuss some practical considerations in §5.2.5.

Despite its practical importance, we are unaware of any previous work on efficient data transfers with limited memory. Some standard collective communication operations can be implemented in ways that limit memory usage, but the general problem we consider seems to be new. Cypher and Konstantinidou designed memory efficient message passing protocols [21]. However, their work addressed exchange of tokens as opposed to variable sized messages, and they did not explicitly consider the effect of finite memory in the processors. Their work conceptually divides a process into communication and application processes. Communication processes receive unit-size messages and copy them to application processes. It is assumed that application processes have enough memory, and the goal is to limit the memory requirement of the communication processes.

5.2.1 Complexity

In this section we show that determining whether a given transfer can be completed in a specified number of phases is NP-Complete. Our proof uses a reduction from the Hamiltonian Circuit problem. Recall that a Hamiltonian Circuit is a cycle in a graph that visits each vertex once. The directed Hamiltonian Circuit problem is known to be NP-Complete [59]. Given an instance of the Hamiltonian Circuit problem, the basic idea of our reduction is to construct an instance of the data transfer problem, in which there is but a single unit of usable memory. This unit is a *token* that is passed between processors, and possession of the token allows a processor to receive data in the next phase. In our construction, a solution to the data remapping problem occurs if and only if the token can be passed in a cycle among all the processors, which implies the existence of a Hamiltonian Circuit.

While the token is being passed in a cycle, the processors must not perform any other data transfers. But when the cycle is completed, processors must be able to finish all their other communication operations. To see how this can be done, consider the Hamiltonian Circuit problem posed in the left portion of Figure 5.4. From this instance, we construct the data remapping problem in the right portion of the figure. The data remapping problem contains the original graph as its *core* (represented in the figure with dark lines) after replacing vertices with processors and replacing edges with unit-volume data transfers. It also contains a *chain* of processors to the left. The bottom processor in this chain has free memory that will percolate upwards with each phase, finally allowing all the data transfers to be completed. Given a Hamiltonian Circuit Graph $G = (V, E)$, we construct a data remapping problem with $\mathcal{P}$ as the set of processors and T as the set of transfer requests as follows.

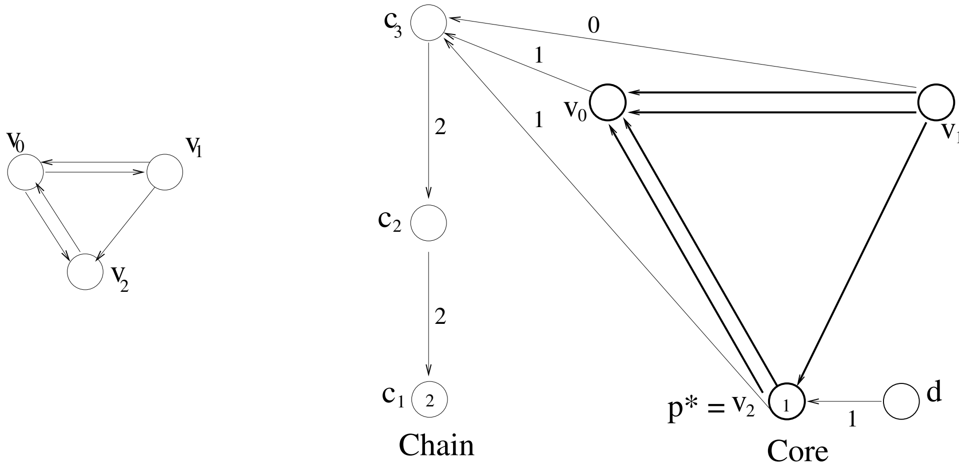


Figure 5.4: Construction for NP-Completeness proof.

- $\mathcal{P}$ has a processor p_i for each vertex v_i of V. We will refer to these processors as core processors. Each edge of E is a unit-volume transfer in T.
- Add a chain of $|V|$ processors $\{c_1, \ldots, c_{|V|}\}$ to $\mathcal{P}$. Also, add transfer requests (c_{i+1}, c_i) to T, each with volume $|E| - |V|$.
- Add a transfer request from each core processor p_i to the top of the chain $c_{|V|}$. This transfer has volume equal to one less than the in-degree of v_i in G.
- Add a dummy processor d and a unit-weight transfer connecting d to an arbitrary processor p^* in the core.
- Give $|E| - |V|$ units of free memory to processor c_1 and 1 unit of free memory to p^*. All other processors have no free memory.

Consider what happens as the data remapping occurs. In the first phase, c_1 will send its data to c_2, moving the free memory one step up in the chain. After $|V|-1$ phases, this free memory will have arrived at $c_{|V|}$, the top of the chain. Meanwhile, the single unit of free memory (the token), which started at p^*, will have meandered about, enabling some data to be transferred.

In phase $|V|$, processor $c_{|V|}$ has enough free memory to receive all of the data that needs from the core processors. During this phase, the token can take one more step. The messages sent to $c_{|V|}$ free up memory in the core processors. Specifically, at the completion of phase $|V|$, each core processor p_i has indegree$(p_i) - 1$ units of free memory (one processor might also have an additional unit of free memory from the token).

In phase $|V| + 1$, core processor p_i can now receive all the data that it needs, minus 1. The complete set of transfers to p_i can be completed in this phase if and only if one of the data transfers to p_i has previously been handled by the token. If there is a processor that was not visited by the token in phases 1 to $|V|$, then that processor cannot receive all its data in $|V| + 1$ phases. But the only way for the token to visit all the core processors in $|V|$ phases is to complete a Hamiltonian Circuit of the core graph. Note that the token must end up where it started, at processor p^*, to enable the transfer from d to occur during phase $|V| + 1$. This argument leads to the following result.

Theorem 5.2.1 *Determining whether an instance of the data remapping problem can complete in a specified number of phases is NP-Complete.*

Proof: Given an instance of the Hamiltonian Circuit problem $G = (V, E)$, construct a data remapping problem as described above. As sketched above, the data remapping problem finishes in $|V| + 1$ phases if the core graph has a Hamiltonian Circuit.

The total amount of data that must be transferred is $|V|(|E| - |V|) + |E| + 1$. The first term comes from the data being sent to the chain and within the chain, the second term reflects data being redistributed within the core, and the last term is the transfer from d to p^*. This quantity equals $(|E| - |V| + 1)(|V| + 1)$. Since there are only $(|E| - |V| + 1)$ units of free memory, the transfers can complete in $(|V| + 1)$ phases only if all the free memory is used at every phase. Thus, the transfers must proceed as discussed above.

If the core graph does not have a Hamiltonian Circuit, then one of its processors will not have been visited by the token by the end of phase $|V|$. That unvisited processor, p_i, still needs to receive indegree(v_i) data, but has only indegree(v_i) $- 1$ units of available memory, so the data transfers cannot complete in $|V| + 1$ phases.

Notice that the construction of the data remapping problem is polynomial, so we can conclude that the data remapping problem is NP-Hard. A given solution can be verified in polynomial time, thus the problem is in NP. □

5.2.2 Multi-commodity Flow Formulation

In this section, we present a multi-commodity flow (MCF) formulation to determine whether a given transfer can complete in a specified number of phases [4]. Once we can solve the decision problem, the number of phases in an optimal solution can be determined using parametric search. This formulation enables use of MCF technology to solve the minimum phase data remapping problem optimally. This might be helpful for three reasons. First, some MCF problems can be solved relatively quickly, despite their intractability in the general case. Second, the continuous version of the MCF problem can be solved in polynomial time and the solution can be used as a heuristic for the integer problem. Finally, MCF solvers will find an optimal solution if runtime is not an issue.

In our MCF formulation, each processor corresponds to a commodity. Let P be the number of processors and L be the number of phases. We must decide if a remapping can complete in L phases. As depicted in Figure 5.5, our MCF graph contains a sequence of components, one for each phase. Each component allows for the communication that occurs in the corresponding phase.

The MCF graph $G = (V, E)$ has $2PL$ vertices. Each processor is represented by $2L$ vertices: two processors (one sender and one receiver) at each phase. We will use r_i^l and s_i^l to denote receiver and sender, respectively, for processor i in phase l. A sender vertex of the first phase is the source of a commodity with volume equal to the total volume of the data originally stored by this processor. A receiver vertex in the last phase is a destination for a set of commodities that corresponds to data to be stored by this processor after remapping is complete.

In the MCF graph, there is an edge from r_i^l to s_i^l for $l = 1, \ldots, L$ and $i = 1, \ldots, P$. The capacity

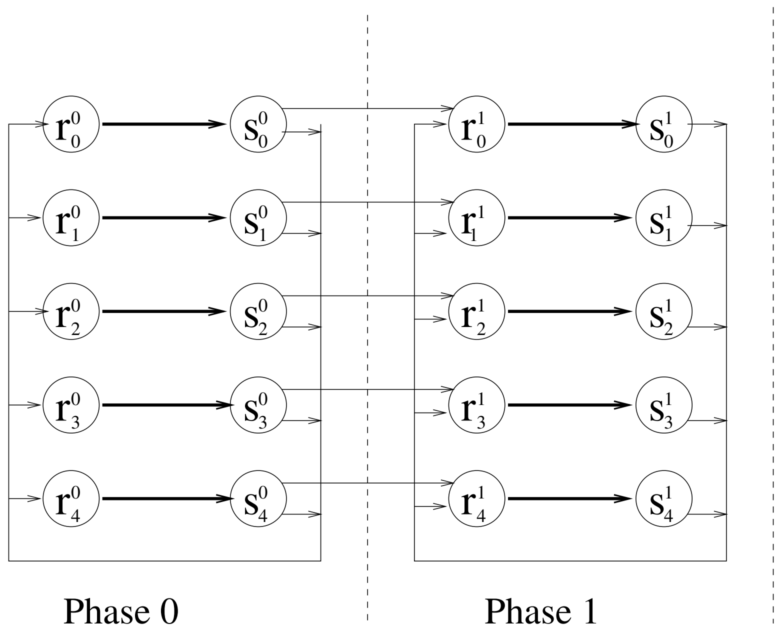


Figure 5.5: MCF graph for 5 processors and 2 phases.

of an edge is equal to the total memory on the respective processor. There are also edges from each sender vertex s_i^l to all other receiver vertices r_j^l in the same phase to enable data exchange between any pair of processors in a phase. These edges have infinite capacities.

With this construction, all processors first receive the data in a phase and then send their messages. This corresponds to first allocating space for the data to be received and then sending the outgoing data. The edges from receivers to senders within a phase guarantee that there is sufficient space to allocate memory for the incoming data before releasing the space for the data being shipped out, so that the memory constraints are guaranteed to be satisfied.

Finally, there is an edge (with infinite capacity) from each sender s_i^l to the receiver in the next phase r_i^l for $l = 1, \ldots, L-1$. The flow on these edges corresponds to data that is already in the memory of a processor at the beginning of a phase. The graph for $P = 5$ and $L = 2$ is depicted in Figure 5.5.

Theorem 5.2.2 *There exists a solution to the remapping problem if and only if there exists a solution to the MCF formulation.*

Proof: We can replace a data transfer from processor i to processor j in phase l with flow on edge (s_i^l, r_j^l) of equal volume. As argued above, memory constraints on the processors are satisfied if and only if the capacity constraints on the edges are satisfied in G. Thus, the feasibility of one solution implies the feasibility of the other. $\square$

In this formulation the number of commodities is equal to the number of processors, and the graph has $2PL$ vertices and P^2L edges. The number of vertices and edges can be reduced for a more efficient formulation. First, we can replace the crossbar between senders and receivers in a phase l with a vertex v^l and edges from all senders of phase l to v^l and edges from v^l to all receivers of phase l. Second, we can merge the senders of phase l with receivers of phase $l+1$. The graph after these reductions is depicted in Figure 5.6. This improved formulation has $PL+L+P$ vertices and $(3L+1)P$ edges.

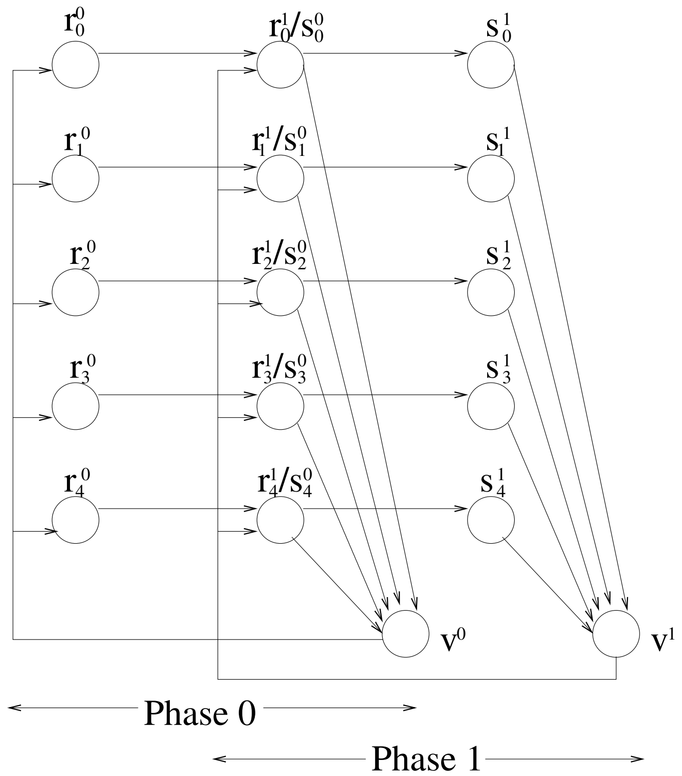


Figure 5.6: MCF graph after reduction.

5.2.3 Continuous Relaxation

Although the multi-commodity flow formulation from §5.2.2 provides a methodology for solving instances of the minimum phase remapping problem, runtime can still be exponential in the problem size. In this section, we describe an efficient solution for an approximation to the remapping problem. In the approximation, integral constraints on the volume of data transfers are relaxed to allow continuous values. Naturally, the volume of transfer between two processors in a phase must be an integer. But integer solutions near the continuous ones can be used as heuristics. Note that the unit of data transfer is only a byte, whereas the volume of data being transferred is often on the order of megabytes. So, conversion from a continuous solution to an integer solution will often be a small perturbation, and so heuristics based upon this idea may be generally effective. However, bad cases for this heuristic exist, as discussed at the end of this section.

As defined in the introduction, T_{ij} denotes the total volume of data to be communicated from processor i to processor j, and t_{ij}^{l} denotes the volume of data transferred from processor i to processor j in phase l. The memory available to processor i at the beginning of phase l is denoted by A_i^l. We also use R_i and S_i to denote the total volume of data received and sent by processor i during remapping.

Let $L = \lceil \frac{T}{M} \rceil$ be the lower bound on the number of phases. We will divide each message into L equal pieces, i.e., $t_{ij}^{0} = t_{ij}^{1} = \ldots = t_{ij}^{L-1} = \frac{T_{ij}}{L}$ and send a piece at each phase. If the memory constraints are satisfied, then the data transfers will complete in precisely L phases. However, there is no guarantee that memory constraints will not be violated. As a solution to this problem, we will use preprocessing and postprocessing phases to ensure feasibility of the phases in between.

Lemma 5.2.3 *If the following conditions are satisfied, the continuous version of the remapping problem can be completed in* $L = \lceil \frac{T}{M} \rceil$ *phases.*

(i) $S_i = R_i$ *for all processors.*

(ii) $A_i^0 \geq \frac{R_i}{L}$.

Proof: At each phase, processor i will receive $\frac{R_i}{L}$ units of data. By the second condition, each processor has sufficient memory for the first phase. By the first condition, each processor ships out $\frac{S_i}{L} = \frac{R_i}{L}$ units of data at each phase, which frees up sufficient memory for the next phase. □

Lemma 5.2.4 *A solution for a continuous version of the data remapping problem for transfer request* $\mathcal{R}$ *can be performed via the following three steps.*

1. *one preprocessing phase.*

2. a new transfer request $\mathcal{R}'$, where $S_i = R_i$ and $A_i^0 \geq \frac{R_i}{L}$.

3. one postprocessing phase.

Proof: In the preprocessing phase we will reorganize the data to satisfy conditions *(i)* and *(ii)* from Lemma 5.2.3, and define a new mapping of the data. After the new mapping is complete, a single postprocessing phase will be sufficient to get all of the data to the correct processor.

In the preprocessing step, all processors i with $R_i < S_i$ will transfer some of their outgoing data to processors j in which $R_j > S_j$, so that in subsequent phases $R_i = S_i$. Note that if the transfer request is feasible, then $R_j - S_j > A_j^0$. Thus, this rearrangement can be completed in a single phase.

Next, as second part of the preprocessing step, processors i with $A_i < \frac{R_i}{L}$ will transfer some of their outgoing data to processors j with $A_i > \frac{R_i}{L}$. To avoid disturbing the first property, the sending processors will also pass equal amounts of receiving assignment. Once again, this step can be completed in one phase, since by construction, the receiving processors have sufficient space.

Notice that the actual data being transferred is irrelevant—we are just trying to balance the numbers—so a send and receive operation can cancel each other. This enables merging of the two steps above into one phase.

After the new transfer request $\mathcal{R}'$ is realized, we must correct for the transfer of receiving assignments. This correction is the purpose of the postprocessing phase. Under the transfer of receiving assignments, each processor is either a sender or a receiver of such assignments. So, during postprocessing, each processor will either receive or send data, but not both. Since the initial remapping is feasible, each processor has enough memory for the data to be received, thus the postprocessing can be completed in one phase. $\square$

The complexity of constructing the solution for the preprocessing phase is linear in the number of processors. To see this, divide the processors into two lists: those with $R_i < S_i$ and those $R_j > S_j$. Now step through the lists together, transferring sending responsibility from a processor in the i list to one in the j list. Each transfer balances R_i and S_i for a processor in one of the lists. The same can be applied to balance initial available memories. Notice that the preprocessing step uniquely describes the postprocessing phase, and remapping for $\mathcal{R}'$ is straightforward.

Theorem 5.2.5 *Given a transfer request $\mathcal{R}$, the continuous version of the data remapping problem can be completed in $\lceil \frac{T}{M} \rceil + 2$ phases.*

Proof: By Lemma 5.2.4, $\mathcal{R}$ can be completed by pre- and postprocessing steps, along with a transfer request $\mathcal{R}'$ satisfying conditions of Lemma 5.2.3. Notice that the total volume of data to be transferred, T' in $\mathcal{R}'$ is no greater than T in $\mathcal{R}$, and the total available memory in the system does not change: $M = M'$. Hence by Lemma 5.2.3, $\mathcal{R}'$ can be completed in $\lceil \frac{T'}{M} \rceil \leq \lceil \frac{T}{M} \rceil$ phases. Together with one preprocessing and one postprocessing phases, remapping can be completed in $\lceil \frac{T}{M} \rceil + 2$ phases. $\square$

It is worth noting that a good solution for this continuous approximation may not yield a good solution for the true discrete problem. For instance, consider the example depicted in Figure 5.7.

Figure 5.7: Catastrophic instance for continuous relaxation.

This example consists of two groups of processors, with no communication between the groups, and there is only one unit of available memory. Available memory must be possessed by each component in turn, and this requires temporarily moving some data from one component to the other to transfer the free memory, as will be discussed in more detail in the next section. In the preprocessing step described in the proof of Lemma 5.2.4, this available memory will be divided into two groups of processors, but the fractional transfers that follow give no insight into the correct way to orchestrate the data transfers for this instance. Specifically, in the continuous solution all processors are identical, so no information is gleaned about the necessity of working on components in turn.

5.2.4 Efficient Approximation Algorithms

In this section, we describe the basics of a family of efficient algorithms that provide solutions, in which the number of phases is at most 1.5 times that of an optimal solution. The algorithm is motivated by some simple observations. First, the maximum amount of data that can be transferred in a phase is equal to the total amount of free memory in the parallel machine. Let M be the total available memory in the parallel machine, and let T be the total volume of data to be moved. Note that M does not change between phases.

Lemma 5.2.6 *The minimum number of phases in a solution is* $\lceil \frac{T}{M} \rceil$.

This bound can be achieved only if available memory is used to receive messages at each phase. Thus, free memory is wasted if it resides on a processor that has no data to receive. Our algorithm works by redistributing free memory to processors that can use it. Equivalently, data is *parked* on a processor with free memory, which it cannot use, to free up memory on processors that can use it. We will park only data that must be transferred eventually.

Parking

Parking aims to utilize memory that would otherwise be wasted. Consider a processor that has received all its data and still has available memory. This memory cannot be utilized in subsequent phases, which decreases the total memory usable for communication, thus potentially increasing the number of phases. Instead, another processor can temporarily move some of its data to this processor to free up space for messages. An example is illustrated in Figure 5.8. In this simple example, the top two processors want to exchange 100 units of data, but each has only one unit of available memory. A simplistic approach will require 100 phases. However, the third processor has 100 units of free memory. By *parking* data on this third processor (i.e. transferring free memory to another processor), the number of phases can be reduced to three.

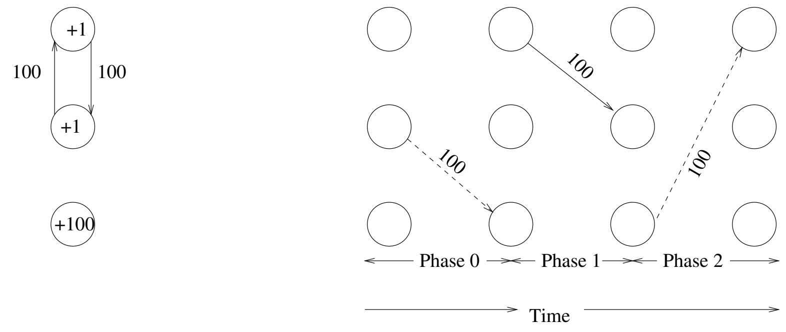


Figure 5.8: Example of the utility of parking.

More formally, if a processor has k units of data left to receive and m units of free memory, then it has *parking space* of $\max(0, m-k)$ units. A processor has data to park if the incoming data exceeds available memory, and the quantity of this parkable data is $\max(0, k-m)$ units. The parkable data consists of data that eventually must be sent to another processor. Note that if the transfer request is feasible, then a processor must send out $\max(0, k-m)$ units. Any processor that has parking space can store parkable data from another processor, thereby maximizing the amount of usable free memory. This parked data merely takes an extra step on the way to its final destination. Exploiting this observation will allow us to construct an approximation algorithm.

In our algorithm, we merely store data in a parking space, and then forward it to its correct destination when the destination processor has available memory. Note that it is inconsequential which processor owns the parked data. In other words, parking spaces are indistinguishable. What potentially effects performance is which processors shunt their data to a parking space.

Lemma 5.2.7 *It is sufficient to park data at most once to obtain an optimal solution.*

Proof: Assume there is a solution that parks some data D twice. Let p_1 and p_2 be the first and second processors on which D is parked. After data is moved from p_1 to p_2, if no other processor uses available memory at p_1, then there was never a need to move data to p_2. If another processor p_i, parks data on p_1, then we can rearrange the data movement with D staying in p_1, and p_i parking on p_2, due to indistinguishability of parking spaces. □

It is worth noting that parking is not just a heuristic but a requirement in some cases. Consider the example in Figure 5.8, modified so that there is no available memory in the top two processors. In this case, the transfer request is still feasible, but realizing the remapping requires parking.

An Approximation Algorithm

In this section, we describe an algorithm that obtains a solution with at most 1.5 times the optimal number of phases. The algorithm is quite generic and allows for a number of possible enhancements.

Algorithm 5.2.1

- *A processor receives as much data as it can in each phase (i.e., if a processor has available memory at the end of a phase, then this processor does not have any more data to receive).*

- *If the transfer request cannot be completed in the next phase, then park as much data as possible (i.e, park the minimum of the total parkable data and the total parking space).*

Note that many details about the algorithm are unspecified: If I have more incoming data than free memory, which messages should I receive in the current phase? If several processors want to park data, but limited parking spaces are available, which should succeed? We will show below that with any answers to these questions, the resulting algorithm generates a solution with no more than 1.5 times the optimal number of phases. Intelligent answers to these questions could be used to devise algorithms with better practical (or perhaps theoretical) performance.

Lemma 5.2.8 *The total volume of data transferred by Algorithm 5.2.1 is at most* $\lceil \frac{3T}{2} \rceil$.

Proof: Let T_p be the volume of data transferred through parking, and let T_d be the data transferred directly. Data is transferred either directly or through parking, thus $T = T_p + T_d$.

It is enough to park data once due to Lemma 5.2.7, thus parked data is moved twice, and the total volume of data moved is $2T_p + T_d = T + T_p$. Because each parked unit of data enables at least one direct transfer, the algorithm guarantees that $T_p \leq T_d$. Thus at most half of T can be transferred through parking, i.e., $T_p \leq \frac{T}{2}$, and the total volume of data moved is $T + T_p \leq T + \frac{T}{2} = \frac{3T}{2}$. □

Theorem 5.2.9 *Algorithm 5.2.1 constructs a solution with at most* $\lceil \frac{3T}{2M} \rceil + 1$ *phases.*

Proof: The algorithm makes use of all M units of available memory until the amount of parkable data is less than the amount of parking space. It then completes in at most two additional phases, one in which some data is parked, and a final phase in which each processor has enough memory to receive all its messages. By Lemma 5.2.8 we know that the total volume of data transferred in the algorithm is at most $\lceil \frac{3T}{2} \rceil$. With M units of transfer in all, but the last two phases, the process can be completed in at most $\lceil \frac{3T}{2M} \rceil + 2$ phases.

We will now decrease the bound to $\lceil \frac{3T}{2M} \rceil + 1$. Let l be the number of phases for the algorithm to complete the data remapping process. The total volume of data transferred is $(l-2)M + x$, where x is the volume of data transferred in the last two phases: $1 < x \leq 2M$. From Lemma 5.2.8 we know that

$$\lceil \frac{(l-2)M + x}{M} \rceil \leq \lceil \frac{3T}{2M} \rceil.$$

But simple algebra reveals that

$$l - 1 \leq \lceil \frac{(l-2)M + x}{M} \rceil.$$

Combining these inequalities

$$l - 1 \leq \lceil \frac{3T}{2M} \rceil,$$

and the result follows. $\square$

Combined with Lemma 5.2.6, Theorem 5.2.9 shows that Algorithm 5.2.1 is a 3/2 approximation algorithm for the minimum phase remapping problem. Without a better lower bound, this value of 3/2 is tight as illustrated by the example in Figure 5.9.

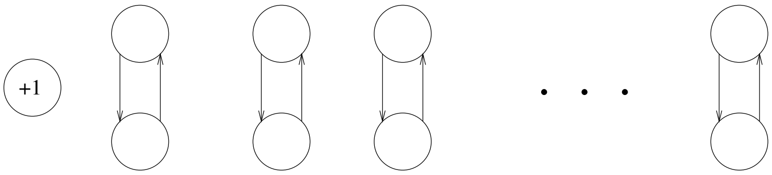


Figure 5.9: Example to show the tightness of the 1.5 bound.

This example consists of an odd number of processors P. All but one of them are organized in pairs which exchange a single unit of data. Only the unpaired processor has a single unit of available memory. The total volume of data to be moved is $T = P - 1$. The only way for a pair to exchange their data is first to park a unit elsewhere, so a total of $\frac{P-1}{2}$ units of parking are needed. Hence, the total volume of data transferred is $\frac{3(P-1)}{2} = \frac{3T}{2}$, and the number of phases is $\frac{3T}{2M}$, since $M = 1$.

5.2.5 Practical Considerations

The preceding sections of this paper have concentrated on the amount of available memory, ignoring how that memory is organized. However, the adaptive applications that require periodic data redistribution typically make extensive use of dynamic memory management. As a consequence, the available memory may not be contiguous, which limits the immediate practical utility of our theoretical analysis. In the following discussion, we assume that the message passing software requires outgoing/incoming messages to be read from/to a single, contiguous block of memory. This is typical of current implementations of MPI.

The complexity of memory layout adds several complications to our simple theoretical model.

First, outgoing data will not generally be contiguous, so space to buffer it on the sending processor will be needed. Second, if available memory is fragmented, then incoming messages can only be as large as the fragments, not the entire free memory (Heroic efforts to circumvent this problem by chopping up messages to fit in fragments are possible, but we ignore them here). Once data has arrived, it may need to be copied into non-contiguous portions of an application data structure. Finally, our analysis assumed that total available memory at the end of a phase could be updated simply by the difference between the total amounts of outgoing and incoming data. Although this may be true, it says nothing about fragment sizes, which are the true, practical limit on useful memory for message passing. Since the useful memory for future phases is unpredictable, algorithms like continuous relaxation that rely on pre-scheduling the messages become problematic.

Despite these caveats, there are instances in which our model and algorithms can be applied without modifications. Consider the case where the application controls memory and packs all data contiguously. Available memory will then consist of a single, contiguous block. If the application data structures allow for the memory to be reorganized to make outgoing message data contiguous, then no further buffering of outgoing data will be necessary. Once messages have arrived, memory can be locally reorganized to place them into the application data structures in the appropriate way, while keeping the available memory contiguous. In this way, the objections raised above are circumvented and our model and techniques are directly applicable.

However, most applications do not allow for such straightforward memory management techniques. For such applications, our theoretical model provides guidance, but not robust theoretical guarantees of performance. In Figure 5.10 we sketch an algorithm that performs robustly, but without the certainty of bounds on the number of phases. In this algorithm, the *neighbors* of a processor q are those processors that wish to send data to q. The basic idea is to have each processor determine its available memory at the beginning of each phase and to restrict the sizes of incoming messages to ensure that this memory is not exceeded. No assumptions are necessary about the amount of available memory in subsequent phases.

Several comments about this algorithm are in order. First, this template does not specify how the sender set or the sending allotments are determined. These choices could be made to mimic as closely as possible one of the algorithms from the earlier sections of this paper. Second, this

```
While data exchange is not complete do
   Determine my receive buffer and send buffer sizes;
   Send my send buffer size to neighbors;
   While there is available memory and data to receive do
      Choose a set of senders S who will send to me;
      Choose a sizes of data senders will send to me;
      Post asynchronous receive;
      Inform senders of their allotments;
   for each sending allotment I am given
      Construct and send message;
      Free up the memory for data sent;
```

Figure 5.10: Template of a robust algorithm for memory-constrained interprocessor communication

approach requires only local communication without any global control. Having said that, the sender and allotment selections are likely to be improved via global information. Third, parking is not included in this template. Parking requires global information and could be included if the message buffer sizes are collected once they are determined. Clearly, by adding such intelligence, worst case performance can be dramatically improved for cases like the one depicted in Figure 5.8. However, the global communication required for this intelligence will make each phase slower and so hurt performance for simpler (presumably more common) cases. We do not know which approach is ultimately superior.

5.3 Conclusions

We addressed two problems associated with redistributing data among processors. The first problem is that of determining the inter-processor communication pattern when each processor knows which components it wants to receive, but does not know which processor owns them. In a simple solution to this problem, processors exchange messages in a ring fashion, which requires all-to-all communication and is unscalable. We described a rendezvous algorithm for an efficient solution for this problem, and our experiments verified its effectiveness. The second problem is that of migrating data in the case of limited memory. The problem arises when processors do not have enough memory to allocate space for their incoming data before releasing the space for the outgoing data. In this case, the remapping operation must be decomposed into phases so that processors free up memory for the data they shipped out at end of a phase, making it available for the incoming data in the next phase. We studied how to complete the remapping operation in a minimum number of phases, a problem we call minimum phase remapping. We showed that the problem of determining whether a given transfer can be completed in a specified number of phases is NP-Complete. A reduction of the minimum phase remapping problem to the multi-commodity flow problem was presented. We showed how a continuous relaxation of the problem admitted a simple solution with two more phases than that of an optimal solution, but it might be difficult to obtain a good discrete solution from this continuous one. We also devised a practical approximation algorithm with a bound of 1.5 times the optimal solution.Finally, we addressed some practical considerations about the problem.

Both of these problems arise from our collaborative efforts to build general purpose libraries to support complicated parallel applications. Both of the features described here are now being added into Zoltan, a public–domain dynamic load balancing tool [23]. Our distributed directory algorithm has been also added to the Aztec linear solver package [95].

Chapter 6

Partitioning for Complex Objectives

Over the past decade graph partitioning has proven to be a critical tool for parallel computing. Unstructured computations can often be described as an undirected graph, in which vertices correspond to computations and edges reflect data dependencies. A graph partitioner can be employed to partition vertices of the graph into disjoint sets of equal cardinality (or weight), while limiting the number of edges crossing between sets. When each set is assigned to a different processor, the computational load is balanced, while the communication requirements are kept small.

Although this model is far from perfect (see, e.g. Hendrickson and Kolda [45]), it has been quite successful, particularly for solving differential equations on meshes. For the graph partitioning model to reflect the runtime of a parallel computation accurately, two assumptions must be satisfied. First, the number (or weight) of cut edges must approximate the communication cost. As discussed in [45], this assumption is often not true, but for mesh-based computations the approximation is acceptable. Second, the computation associated with a subdomain must be approximately equal to the total number (or weight) of vertices assigned to it. For several important numerical operations this assumption is valid — specifically matrix-vector multiplication or explicit calculations.

However, for some computations the work is not a simple sum of vertex contributions, but rather a complicated function of the partition. For such problems the work per subdomain (and hence load balance) cannot be assessed until the partition is computed. This leads to a chicken-and-egg problem: the work cannot be computed without the partition, but the partitioner cannot balance the load without knowing the work. Examples of this phenomena include the following.

- **Computation plus communication**: Traditional partitioners balance the computational work on each subdomain, but the communication requirements can vary widely between processors. If there is no synchronization between the computing and communicating phases, then it is better to balance the sum of the two. Unfortunately, the communication cost depends upon the partition.

- **Direct solves on each subdomain**: In domain decomposition methods, equations in each subdomain are approximately solved independently and then the subdomain solutions are combined. For some domain decomposition solvers, like the FETI class of methods [28], each subdomain is solved via a sparse, direct factorization. The work and memory requirements of a sparse direct solver are complicated functions of the subdomain, and are fairly expensive to compute.

- **Approximate factorization preconditioners**: Related to the problem of direct solves on each subdomain, a common class of parallel preconditioners uses an approximate factorization on each subdomain. Again, the work and memory associated with the approximate factorization are complex functions of the full subdomain.

- **Overlapped subdomains**: Traditional partitioners assign a vertex uniquely to a subdomain. For some kinds of operations, the work involves a larger class of vertices. One such example is overlapped Schwartz domain decomposition [91]. In this approach, the mesh elements (which correspond to graph vertices) are partitioned among processors, but the application of the preconditioner on each subdomain includes mesh elements that are near to but not in that subdomain. If some subdomains have many such near-by elements but others have few, then balancing the partitions will not balance the cost of the preconditioner. Again, the cost of a subdomain is a complicated function of the partition. As we will show in §6.3, this problem is closely related to balancing computation plus communication.

- **Parallel multifrontal solvers**: When performing a parallel, sparse factorization via a multifrontal method, the work and storage associated with a subdomain depend upon the size of its boundary with other processors [90]. But again, the boundary is a function of the partition.

In this chapter, we review the graph partitioning problem in § 6.1, then describe in §6.2 a general framework for addressing complex partitioning problems. In §6.3 we illustrate the application of these ideas to two of the listed complex partitioning problems—balancing overlapped subdomains, and balancing communication and computation. In § 6.4 and § 6.5, we discuss experimental results and future work for these two problems. Finally, we present some basic ideas for applying this framework to balance factored domains in § 6.6. Parts of this work have been published in [82].

6.1 Preliminaries

In this section, we will review the graph partitioning problem and one of the heuristics, namely Kernighan-Lin (KL), which has been the basis for many other research efforts.

6.1.1 Graph Partitioning

A graph $G = (V, E)$ is defined by a set of vertices V and a set of edges E. Each edge $e = (u, v) \in E$ is defined by a pair of vertices u, v. A vertex u is *adjacent* to vertex v if and only if they are connected by an edge, i.e. $(u, v) \in E$. We will use $Adj(v)$ to denote the set of vertices adjacent to vertex v.

$$Adj(v) = \{u : (u, v) \in E\}.$$

The *degree* of a vertex $deg(v)$ is the number of vertices adjacent to it,

$$deg(v) = |Adj(v)|.$$

A weight function $w(v)$ can be defined to map vertices to positive numbers. The weight of a set of vertices $w(V')$ is defined by the sum of weights of vertices in the set, i.e.,

$$w(V') = \sum_{v \in V'} w(v).$$

Similarly we can define a cost function $c(u, v)$ for edges and generalize it for sets of edges

$$\mathcal{C}(E') = \sum_{(u,v) \in E'} c(u, v).$$

A P-way partition $\mathcal{P} = \langle \mathcal{P}_1, \mathcal{P}_2 \ldots, \mathcal{P}_P \rangle$ of a graph is a partitioning of its vertices. An edge (u, v) is *internal* if its vertices are in the same part ($\exists \mathcal{P}_i \;\; u, v \in \mathcal{P}_i$), and it is *cut* otherwise. Cost of a cut C(P) is defined as the sum of costs of cut edges it produces.

A partition is *balanced* if all parts have about the same weight. Formally, a partition is balanced if and only if

$$\frac{W_{max} - W_{avg}}{W_{avg}} \leq \epsilon,$$

where W_{max} and W_{avg} are maximum and average part weights, respectively, and ϵ is a predetermined balance ratio.

With these definitions graph partitioning problem can be defined as *finding a balanced partition* $\mathcal{P}$ *with minimum cost* $\mathcal{C}(\mathcal{P})$.

6.1.2 Kernighan-Lin based Heuristics

The Kernighan-Lin (KL) heuristic [61] is a local search algorithm and has become the basis of many graph partitioning heuristics. We will describe the heuristic for bipartitioning, but it can be generalized for more than two parts. It starts with an initial balanced partition and tries to improve the quality of the partition by swapping vertices from different parts until it reaches a local optimum.

Choosing the pair of vertices is the critical part of the algorithm. The *gain* of a swap is defined as the change in the cost of the partition when two vertices are swapped. The KL heuristic chooses the pair with maximum gain and tentatively swaps them. Once moved, vertices are locked (not allowed to be swapped). The heuristic proceeds by choosing other pairs, keeping a record of all tentative swaps and their gains. The algorithm stops when all vertices are locked. Of all the swaps, only a sequence starting with the first move that result in the maximum overall gain are realized. We call this one pass of the algorithm. Subsequent passes start after unlocking all the vertices. The heuristic stops making passes when there is no gain, which means a local optimum solution has been found.

Fiduccia and Mattheyses [29] later introduced the concept of a *move*, where only one vertex is moved at a time as opposed to two vertices being swapped. This reduces the complexity of the neighborhood search of this local search algorithm, while maintaining the quality of the partitions.

(1) Generate starting partition using standard tools.
(2) Evaluate cost of partition with respect to desired objective.
(3) Determine global schedule for cost transfer.
(4) Until threshold cost reduction is achieved ...
    (4a) Select next vertex to move.
    (4b) Update cost (approximation? bound?) on affected processors.
(5) Evaluate new partition and perhaps go to (3).

Figure 6.1: Framework for addressing complex partitioning objectives.

Like all other local-search algorithms, performance of the KL heuristics degrades both in terms of speed and quality of partitions as the problem size grows, and becomes very sensitive to the initial solution. Multi-level techniques alleviate this problem. In 2-level partitioning for instance, we first match pairs of vertices that are likely to be in the same part. We then generate a coarser graph in which a vertex pair of the original graph is represented by a single vertex. This reduces the problem size and yields a more tractable problem. Once this smaller problem is solved, the solution is projected back to the original graph, which is used as an initial solution to a local search algorithm. We can use the same technique to partition the coarse graph which gives a multilevel algorithm. Initialized by Hendrickson and Leland [47], many popular graph partitioning tools are built on the multilevel idea [47, 60].

6.2 Partitioning for Complex Objectives

Our general approach to partitioning for complex objectives is outlined in Figure 6.1. The steps in this outline are discussed below. The application of this general framework to a specific complex objective is detailed in §6.3.

Our approach begins in step (1) by partitioning the graph using standard tools. There are several reasons for this. First, partitions generated by standard tools have properties that may be desirable for complex objectives. These properties include small interprocessor boundaries, small aspect ratios, and connected subdomains. Second, our task becomes that of perturbing an existing partition, which is easier than generating one from scratch. Third, we are interested in objective functions that cannot be computed until the partition is known, so using standard approaches gives us a viable starting point for our optimization.

Given a starting partition, we next evaluate it under the desired objective function in step (2). The result of this evaluation is the assignment of a *cost* to each subdomain. The cost could reflect the work to be performed on that subdomain, the memory associated with it, or any other appropriate value that needs to be balanced. Our goal is to minimize the cost of the most expensive subdomain. We will accomplish this by moving vertices from expensive subdomains to inexpensive ones.

With this cost in hand, in step (3) we determine a global schedule for moving cost between subdomains. This schedule is a directed graph whose vertices correspond to subdomains and whose edges are weighted by the desired transfer of cost between two subdomains. For the experiments described below we use a diffusion method to generate this schedule, a standard technique in dynamic load balancing [20]. Several caveats are necessary here. First, the total cost will not, in general, be conserved. That is, moving a vertex from one domain to another may reduce the cost on the first domain by a different amount than the cost is increased on the second subdomain. Second, the change in cost associated with a vertex addition or deletion may be expensive to compute exactly. Hence, as we move vertices we may have to employ approximations or bounds on the cost updates. Both these considerations compel us to treat the schedule as a guideline, not a rigid prescription.

Once the schedule is determined, we begin to move vertices between subdomains in step (4). We select vertices to move based upon criteria of partition quality. Thus, we may use edge-cut or aspect-ratio considerations to prioritize vertices. As a vertex is moved, we update the cost of the subdomains it is moving between. The details of this update are intimately tied to the objective function we are trying to optimize. But as discussed above, we may have to resort to heuristic approximations or upper and lower bounds to keep the cost of updates manageable.

The process of vertex transfers and cost updates continues until a significant fraction of our schedule has been achieved. If the cost updates for our objective are exact, we can continue movements until completion. But if they are inexact, then it may be counterproductive to continue moving vertices. In this case, we stop early, evaluate the new partition under the objective function, and repeat the schedule generation and vertex move process, as illustrated in step (5).

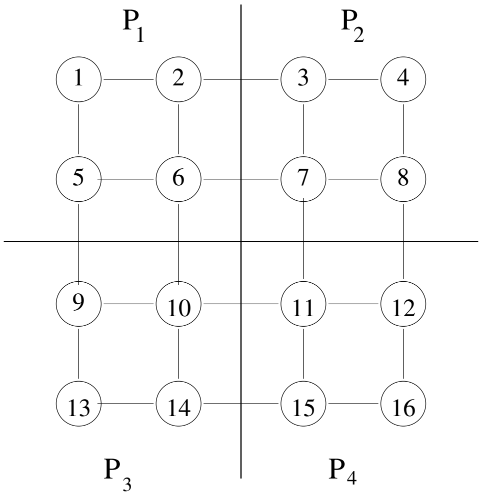


Figure 6.2: Example graph

6.3 Partitioning for Balanced Overlapped Subdomains

In this section we show how the general approach outlined above can be applied to two closely-related complex objective functions. Specifically, we will use it to generate partitions with the property that the overlapped domains are balanced, and to produce partitions in which the sum of computation plus communication costs are balanced. As discussed in the introduction, the former objective arises in overlapped Schwartz domain decomposition [91], while the latter is relevant for operations in which there is no synchronization between computation and communication phases.

6.3.1 Problem Definitions

Given a partition $\mathcal{P} = \langle \mathcal{P}_1, \mathcal{P}_2, \ldots \mathcal{P}_P \rangle$, an extended subdomain $\mathcal{P}_i^+$ for a part $\mathcal{P}_i$ and a single level of overlap is defined as

$$\mathcal{P}_i^+ = \mathcal{P}_i \cup \{v : v \notin \mathcal{P}_i;\ u \in \mathcal{P}_i;\ (v,u) \in E\}. \tag{6.1}$$

The first term corresponds to nodes owned by processor $\mathcal{P}_i$, while the second term reflects additional nodes in the extended domains, which we will refer to as *neighbor* nodes $N(\mathcal{P}_i)$. In the example of Figure 6.2, $\mathcal{P}_1 = \{1, 2, 5, 6\}$, and $\mathcal{P}_1^+ = \mathcal{P}_1 \cup \{3, 7, 9, 10\}$.

The cost of a domain D is defined as

$$cost(D) = \sum_{v \in D} w(v). \tag{6.2}$$

Our goal is to find a partition in which the maximal cost of the extended subdomains is kept small.

This problem is also closely related to the problem of balancing computation plus communication in matrix-vector multiplication or explicit stencil operations. Traditional partitioners merely balance the computational work, and the communication load may be unbalanced. With current speeds of processors making communication costs more significant, communication imbalance can limit performance. So by balancing the sum of computation plus communication cost, performance may be improved. To explain the similarity between the two problems better, we will first define $Recv(\mathcal{P}_i)$ and $Send(\mathcal{P}_i)$, for nodes processor i receives and sends, respectively, during matrix-vector multiplication. A processor receives data for all those nodes that are connected to one of its own, thus

$$Recv(\mathcal{P}i) = \{v : v \notin \mathcal{P}_i;\ u \in \mathcal{P}_i;\ (u,v) \in E\}.$$

Notice that $Recv(\mathcal{P}_i)$ is the same as the second term in Equation (6.1). Data for a node is sent to all processors that have a node connected to it. Since each node can be sent to multiple processors, we will define $Send(\mathcal{P}_i)$ as a multi-set.

$$Send(\mathcal{P}_i) = \biguplus_{1 \le j \ne i \le P} \{v : v \in \mathcal{P}_i;\ u \in \mathcal{P}_j;\ (u,v) \in E\},$$

where $\uplus$ denote the multi-set union operation. Note that $Send(\mathcal{P}_i)$ is identical to all the pieces of $Recv$ sets that other processors will be obtaining from processor $\mathcal{P}_i$. In the example from Figure 6.2, $Recv(\mathcal{P}_1) = \{3, 7, 9, 10\}$, and $Send(\mathcal{P}_1) = \{2, 6, 5, 6\}$. Notice that 6 is sent to processors 2 and 4 and thus appears twice in the multi-set.

The time to send a message can be computed as setup time plus unit transmission time multiplied by the length of the message. Thus, the total cost of a subdomain can be defined as follows, where β and μ are the costs of one unit of data transfer and setup time, respectively, normalized by the cost of computation.

$$cost(\mathcal{P}_i) = \mu(pconn(\mathcal{P}_i)) + \beta(|Send(\mathcal{P}_i)| + |Recv(\mathcal{P}_i)|) + \sum_{v \in \mathcal{P}_i} w(v). \tag{6.3}$$

Here, we use $pconn(\mathcal{P}_i)$ to denote the number of processors $\mathcal{P}_i$ is communicating with. It is worth noting that our algorithm allows us to have different costs for sending and receiving (both startup and bandwidth).

The first terms in Equation 6.3 correspond to total communication cost, while the last term corresponds to computational load. As discussed in § 6.2, our algorithm perturbs an existing partition, and it is unlikely that this perturbation will change the number of communicating pairs. Thus we can add the first term to the cost of each subdomain at the initial cost evaluation step. Although updating the cost due to this term is possible, it will not be included in following discussion.

A related problem is balancing memory requirements of processors, which might be important for memory-intensive applications. Each processor must store data items for vertices for which it does the computation and vertices it receives from other processors. If processors work on only vertices assigned to them, their memory requirement will be defined exactly by the cardinality of its extended domain.

For all these problems, the key information is the set of neighboring adjacency information that is contained in the *Send* and *Recv* sets. For overlapped partitioning, the cost associated with this information is the sum of vertex weights, while for balancing communication plus computation all that matters is the cardinality of these sets. But in either case, the same algorithmic structure is necessary to monitor these sets as vertices are moved. In the next section, we describe an algorithm to track this information efficiently and balance either objective.

6.3.2 Data Structures and Algorithms

As discussed in §6.2, we start with an initial partition obtained from a graph partitioning tool and then try to move to a nearby, balanced partition. Hopefully, the initial partition is a good approximation to minimize boundary size and thus the volume of extra work due to overlaps. In this section, we will discuss the application of the general framework sketched in Figure 6.1 to the specific problems of overlapped subdomains and communication plus computation. We show how to update the cost of partitions efficiently, how to update data structures after a vertex is moved, and how to choose the next vertex to move. As we show below, for this particular objective function the cost function can be updated exactly at small cost. So the additional complexities discussed in §6.2 associated with approximate values do not arise.

Let $part[v]$ denote the partition that vertex v is assigned to, and let $Adj(v)$ be the set of vertices adjacent to v. Let $conn(v,p)$ be the number of vertices in partition p that are either vertex v or neighbors of v,

$$conn(v,p) = \#(u : u \in Adj(v) \cup \{v\}; part[u] = p).$$

After moving a vertex, we need to update costs of the subdomains the vertex is moved to and from. When a vertex v is moved out of a partition, the cost of that partition is decreased by the total weight of vertices that are disconnected from the partition. Thus, for each vertex we will keep $pos(v)$, the set of vertices that will be disconnected from $part[v]$ if v moves to another subdomain,

$$pos(v) = \{u : u \in Adj(v) \cup \{v\}; conn(u, part[v]) = 1\}.$$

We also need to determine efficiently the increase in the cost of a partition p that vertex v is moved to. Let $nconn(v,p)$ be the neighbors of v that are currently connected to part p,

$$nconn(v,p) = \{u : u \in Adj(v) \cup \{v\}; conn(u,p) > 0\}.$$

The increase in the cost of a part p if v is moved to p is $cost((Adj(v) \cup \{v\}) \setminus nconn(v,p))$. For problems arising from the solution of differential equations, each vertex will be connected typically to only a small number of parts, so $nconn$ will be a sparse data structure.

Besides updating the costs of parts after moving a vertex, it is essential to update the data structures for the whole graph for subsequent moves. Figure 6.3 presents the pseudocode for updating these data structures efficiently. Notice that the modifications are limited to vertices of distance at most two from the moved vertex (neighbors and their neighbors). The locality of this update operation enables an efficient overall algorithm.

It is worth noting that we define pos and $nconn$ as sets for simplicity of presentation. For an efficient implementation, it is sufficient to keep the total weight of vertices for balancing overlapped partitions, and cardinality for balancing communication plus computation.

We also need a good metric to choose the vertex to be moved. In our implementation, we choose the vertex with maximum decrease in the total work. Alternatives are minimizing edge cut or aspect ratio. These alternatives could be employed without altering the data structure or the

$part[from] \leftarrow part[from] \setminus \{v\};$
$Recv[from] \leftarrow Recv[from] \setminus pos(v) \cup \{v\};$
$part[to] \leftarrow part[to] \cup \{v\};$
$Recv[to] \leftarrow Recv[to] \cup (Adj(v) \setminus nconn(v, to));$
$Send[from] \leftarrow Send[from] \setminus \biguplus_{j \neq from \,\&\, conn(v,j)>0} \{v\};$
$Send[to] \leftarrow Send[to] \cup \biguplus_{j \neq to \text{ and } conn(v,j)>0} \{v\};$
$pos(v) \leftarrow Adj(v) \setminus nconn(v, to);$
forall $u \in Adj(v) \cup \{v\}$
    decrement $conn(u, from)$;
    if $conn(u, from) = 0$
        remove one instance of u from $Send[pvect(u)]$;
        forall $t \in Adj(u)$
            remove u from $nconn(t, from)$;
    else if $conn(u, from) = 1$
        find $t \in Adj(u)$ s.t. $part[t] = from$;
        add t to $pos(t)$;
    increment $conn(u, to)$;
    if $conn(u, to) = 1$
        add one instance of u to $Send[pvect(u)]$;
        forall $t \in Adj(u)$
            add t to $nconn(t, to)$;
    else if $conn(u, to) = 2$
        find $t \in Adj(u)$ s.t. $part[t] = to$;
        remove u from $pos(t)$;
if $conn(v, to) = 1$
    remove one instance of v from $Send[to]$;

Figure 6.3: Algorithm for updating pos, $nconn$ and $conn$ fields after moving vertex v from partition $from$ to partition to.

basic algorithm.

6.3.3 Higher Level Overlaps

In the discussion so far we have limited our consideration to only one level of overlap, but sometimes two (or more) levels are used for better accuracy in the preconditioning step. Techniques described in the previous chapter can be used for higher level overlaps as well.

We defined $pos(v)$ to be the set of vertices that will be disconnected from $part[v]$ when v is moved to another partition. When we use m levels of overlap, then any vertex within a distance

m of v can be in this set, so the new definition of pos will be

$$pos(v) = \{u : dist(u, v) \leq m;\ conn(u, part[v]) = 1\},$$

where $dist(u, v)$ is the shortest distance between vertices u and v. Similarly, $nconn(v, p)$ can be defined as

$$nconn(v, p) = \{u : d(u, v) \leq m;\ conn(u, p) > 0\}.$$

The rest of the algorithm works without any changes.

Notice that we replaced only $Adj(v)$ in definitions pos and $nconn$ with the set of vertices within a certain distance. This makes it possible to use the data structures and algorithm defined in the previous section as is, by changing only the Adj fields, thus defining a new graph. This new adjacency information can be considered to be a modified graph. Specifically, to partition a graph G with m levels of overlap, we can define a new graph G^m with the same set of vertices, and edge set changed to connect a vertex to all vertices within a distance m in G. Using matrix notation, if A is the adjacency matrix of G, then A^m will be the adjacency matrix of G^m. Our previous algorithm can be applied without modification to G^m to obtain a balanced 1–level overlap partition for G^m, which gives a balanced m–level partition for G.

6.4 Experimental Results

We tried our approach on the set of matrices in Table 6.1, which have been collected from various applications that use overlapped Schwartz preconditioners. All the matrices come from PDE–based applications. For unsymmetric matrices we used the nonzero pattern of $A + A^T$, as suggested by Heroux [48].

The results of applying our methods to balance overlapped domains are shown in Table 6.2. Imbalance is computed as the ratio of the most heavily loaded processor to the average, minus 1. Recall that the total amount of work differs with changing partitions. Generally, our modifications increase the total work, because the initial partition is an approximation to minimize the total volume of extra work. Therefore, in parentheses we also show the percentage reduction in the load of the most heavily loaded processor, which we compute as (previous − current)/current. For all

Table 6.1: Properties of the test matrices.

Name	N	NNZ	Nonzeros per row		
			Max	Min	Avg
Braze	1344	142296	161	23	105.9
Defroll	6001	173718	97	6	29.0
DIE3D	9873	1723498	497	8	174.6
dday	21180	1033324	53	13	48.8
visco	23439	1136966	469	10	48.5
sls	36771	2702280	80	23	73.5
ocean	143437	819186	6	1	5.7

Table 6.2: Initial and improved percent imbalance for overlapped domains.

Name	$P = 4$		$P = 8$		$P = 16$		$P = 32$	
	Init.	Imp.	Init.	Imp.	Init.	Imp.	Init.	Imp.
Braze	6.93	1.86(4.98)	26.65	6.60(15.69)	54.11	26.92(14.72)	71.42	57.49(5.25)
Defroll	20.66	7.59(2.99)	28.43	13.80(0.74)	34.04	22.92(0.91)	51.02	42.95(0.00)
DIE3D	20.03	6.24(7.72)	23.36	15.11(1.71)	53.43	23.41(17.43)	82.36	65.59(2.66)
dday	3.06	0.59(2.48)	5.07	0.21(4.94)	7.96	1.98(5.71)	14.03	4.25(9.54)
visco	6.44	0.87(3.53)	8.05	3.65(2.00)	26.30	3.18(21.87)	36.99	18.15(13.69)
sls	5.57	1.51(3.35)	10.28	1.35(8.32)	17.96	2.21(13.57)	31.83	8.04(18.13)
ocean	2.21	0.02(2.20)	2.44	0.15(2.43)	5.66	0.58(5.65)	10.61	0.85(10.54)

the large problems we are able to reduce the maximum load significantly, which should directly translate into improved performance. Even seemingly simple matrices like the 6–point stencil of ocean exhibits a nontrivial amount of imbalance when partitioned onto enough processors. We are unable to provide much help for the small problems. Moving a vertex can significantly change the cost of a subdomain, so the granularity of these problems is too coarse for our algorithm to make much progress.

As can be seen in Table 6.3, the runtime of our algorithm is consistently much less than the time spent by the initial partitioner, for which we used the multilevel algorithm in Chaco [47]. In addition, the initial cost evaluation dominates the runtime of our algorithm, and this step is necessary for the application even if our improvement methodology is not invoked. Furthermore, the fraction of time spent in our algorithm decreases as the matrices become larger. This is expected, since our algorithms work on only the boundaries of the matrix, so our algorithms scale well for larger problems.

Our algorithms can be used for higher levels of overlaps, as discussed in § 6.3.3. Table 6.4

Table 6.3: Run-times normalized to graph partitioning times

Name	$P=4$	$P=8$	$P=16$	$P=32$
Braze	0.22	0.33	0.31	1.00
Defroll	0.55	0.36	0.25	0.16
DIE3D	0.87	0.61	0.73	0.82
dday	0.19	0.14	0.12	0.14
visco	0.09	0.37	0.24	0.24
sls	0.47	0.44	0.26	0.25
ocean	0.13	0.14	0.14	0.14

Table 6.4: 2-level overlaps

Name	$P=4$		$P=8$		$P=16$		$P=32$	
	Init.	Imp.	Init.	Imp.	Init.	Imp.	Init.	Imp.
dday	5.77	1.36(3.90)	8.61	0.16(8.55)	13.25	2.80(9.45)	21.85	4.55(16.75)
sls	9.19	1.45(6.73)	15.09	0.53(14.76)	29.65	4.77(20.96)	50.38	11.75(27.20)
ocean	3.76	0.79(2.63)	3.82	1.52(2.32)	10.18	1.29(10.17)	18.15	1.85(18.14)

presents results for 2-level overlap on a smaller set of matrices. For this experiment we reduced the test set to relatively sparser matrices, since two-level overlap significantly increases extended subdomain size. As the results show, the proposed techniques can effectively reduce load imbalance for 2-level overlaps.

Using the same set of test problems, we have tried to balance the sum of communication plus computation time. The results are presented in Table 6.5. We used the costs for message startup and transmission times taken from Sandia's cluster computer consisting of DEC Alpha processors connected by Myrinet. Specifically, we used a value for μ of 2000, and for β of 3.2. Again, we see significant overall improvement for all but the smallest problems.

6.5 Future Work for Partitioning with Overlaps

When using overlapped subdomain preconditioners, dense rows in the matrix cause significant problems. A dense row creates one huge unit task with its own weight and weights of all its neighbors. For example, matrices arising from circuit simulations typically have a few very dense rows corresponding to power, ground, or clock connections. Some matrices arising from differential equations have dense rows corresponding to constraints like globally conserved quantities.

Even when there is no very dense row, partitioning and maintaining scalability becomes difficult

Table 6.5: Initial and improved percent imbalance for communication plus computation.

Name	$P=4$		$P=8$		$P=16$		$P=32$	
	Init.	Imp.	Init.	Imp.	Init.	Imp.	Init.	Imp.
Braze	8.03	1.25(6.10)	25.15	1.30(21.77)	46.14	1.03(40.56)	48.59	14.76(28.10)
Defroll	17.90	0.49(15.65)	18.21	2.28(13.88)	25.83	0.50(21.44)	39.08	1.58(31.65)
dday	1.03	0.07(0.94)	2.64	0.10(2.27)	6.59	0.6(5.88)	12.97	1.78(10.59)
visco	1.76	0.35(1.19)	3.40	1.28(1.83)	13.76	2.42(10.84)	25.94	1.29(24.01)
sls	2.28	0.13(2.10)	3.67	0.16(3.44)	11.44	2.07(8.83)	16.00	1.99(12.74)
ocean	2.38	0.10(2.25)	3.97	0.55(3.46)	9.35	0.91(8.91)	22.16	0.54(22.50)

when the average degree of vertices is very high. Such matrices might arise in finite element simulations with elements that have many degrees of freedom. The movement of a single high-degree vertex can significantly change the work assigned to a subdomain. Thus, perfect load balance is very difficult to attain. Even a single level of overlap may not be desirable for such problems, since the extended subdomains may be much larger than the non–extended subdomains. If the extensions due to overlap form a significant portion of the total load on a processor, then little will be gained by applying more processors. Equivalently, the total work grows as the problem is divided into smaller and smaller subdomains, which can limit the scalability of such approaches.

To maintain scalability for these relatively dense matrices, we are considering 1/2–level overlaps, which use vertex separators instead of edge separators as in m–level overlaps. For simplicity, assume we have only two processors. We will decompose the graph into three sets $\mathcal{P}_1$, $\mathcal{P}_2$ and S so that S is a separator in this graph (i.e., there are no edges between vertices in $\mathcal{P}_1$ and $\mathcal{P}_2$). In the preconditioning phase, processors work on $\mathcal{P}_1 \cup S$ and $\mathcal{P}_2 \cup S$, so S is the overlap domain. For the matrix–vector product phase we distribute the nodes in S to processors.

This scheme will limit the increase in total work, thus improving scalability, because a smaller set of the vertices is duplicated. Inevitably, a dense row will be duplicated in many subdomains, but unlike the case of 1–level overlaps, it will not duplicate its neighbors into the partition it is assigned to. In graph partitioning terminology, 1–level overlaps correspond to duplicating wide separators, whereas 1/2–level overlaps correspond to duplicating narrow separators.

In general, preconditioner quality improves with more overlap, so we anticipate that our 1/2–level overlap idea will reduce the numerical convergence rate. But by significantly improving the parallel performance, we anticipate an overall improvement in runtime. We are currently investigating the numerical feasibility of this 1/2–level overlap idea.

A second area that we are investigating is the parallelization of our approach. Most of the steps in Figure 6.1 are straightforward to parallelize. The partition evaluation in step (2) is naturally parallel. The construction of a global schedule for cost transfer in step (3) is a standard problem in dynamic load balancing, so we can exploit existing methodologies. With this schedule, the set of interprocessor transfers can be divided into independent problems and executed in parallel. Our current (serial) implementation first transfers cost to the sinks in the directed transfer graph, which allows for the transfer schedule to be updated with exact cost information. This will not be possible in parallel. We plan to add this functionality to the Zoltan dynamic load balancing tool [24].

6.6 Partitioning for Factorization

The key to the efficiency and robustness of an iterative solver is an effective preconditioner. In the most general setting, a preconditioner M is a matrix close to the original matrix A, so that $M^{-1}A \approx I$. Solving the system $M^{-1}Ax = M^{-1}b$ is much easier for an iterative method than solving $Ax = b$, because of its smaller condition number. While some techniques (e.g., sparse approximate inverses) rely on directly computing M^{-1}, many preconditioners compute M, and each iteration of the solver involves solution of a system $My = z$. Thus we want M not only to be close to A, but also to yield an easy-to-solve linear system. This preconditioning step is often the most time consuming part of an iterative solver.

For performance reasons, many parallel solvers use block preconditioners, where each processor applies preconditioning to its submatrix, and the nonzeros off the processor subdomains are ignored. Considering the dominance of the preconditioning phase in the overall cost of iterative solvers, preconditioners must be the leading factor for partitioning to achieve good load balance. In this work, we study how we can achieve balance for partitioning for incomplete and complete factorizations.

One of the most popular preconditioning strategies for iterative solvers involves computing an *incomplete factor* of the matrix. The idea behind this approach is to calculate a lower triangular matrix that has a tractable amount of nonzeros, but still is close to the exact factor of the original matrix. The number of nonzeros in the exact factor can be significantly larger than in the original

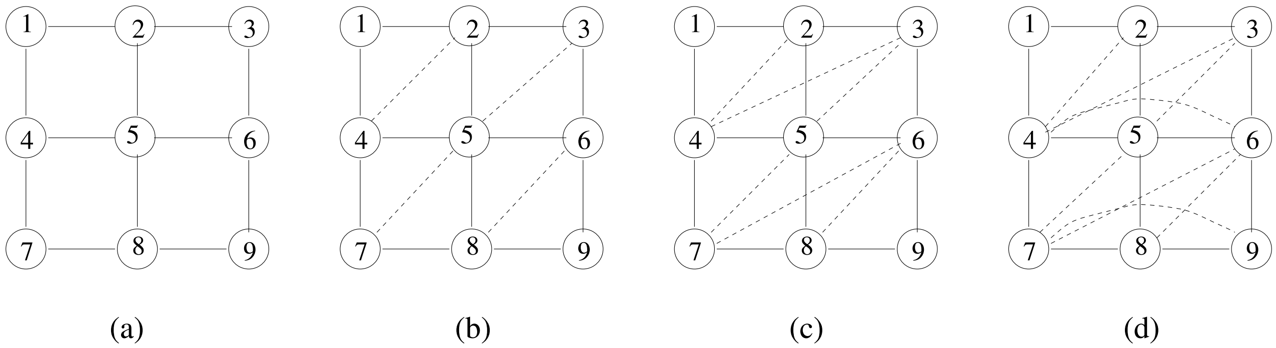


Figure 6.4: Nonzero structures for different preconditioners. Dashed lines represent fill edges. (a) Original matrix (b) 1-level fill (c) 2-level fill (d) Complete factorization

matrix.

From a graph theoretical point of view, eliminating a row in Cholesky factorization can be considered as replacing a vertex with a clique of its neighbors. This will introduce new edges in the graph (nonzeros in the matrix), which we call *fill* edges. These fill edges change the adjacencies in the graph, and will in turn add more edges in subsequent eliminations. The number of new edges in the factor depends strongly on the order in which rows are eliminated and can grow enormously if a proper ordering is not used. Thus ordering rows/columns of a matrix to preserve sparsity has been the subject of numerous research efforts.

Incomplete factorizations drop some of the fill edges to keep the factored matrix sparse. One way is to drop entries that are relatively small in magnitude, since they are unlikely to have a significant effect on the overall solution. Another way is to limit the depth of fill, which does not allow fill edges keep adding more fill to the graph. The simplest way is to factor the matrix, but do not add any nonzeros (fill edges). This method is called *no-fill* factorization. In this method, the number of nonzeros is small, but closeness to the exact factor is dubious. One can allow one-level of fill (add fill edges to the graph but do not let that edge add any more edges) to get closer to the original factor at the cost of increased number of nonzeros. This can be generalized into k-level fill, where a new edge causes new fills in the next $k-1$ elimination steps it is involved in. In this case, there will be an edge between two vertices in the graph of the factor if and only if there is a path of length at most k between the two vertices through lower numbered vertices. And in the extreme case, k is chosen as the dimension of the matrix, so that the exact factor is computed. Figure 6.4 presents an example of nonzero structures for different preconditioners.

Balanced partitions for no-fill preconditioners can be found using conventional graph partitioners, because the load per row can be predicted a priori. For higher levels of fill however, the load per row depends on the ordering, which is computed after subdomains are determined, and thus cannot be predicted a priori. Total work is defined not by individual vertices, but by the domain as a whole, and thus it is not possible to achieve good balance merely by assigning weights to vertices and then using standard graph partitioning techniques. It is not easy to compute the load changes when we move a vertex from one part to another, since it requires placing the vertex in the ordering of the new subdomain, and the effect of adding or removing a vertex are not necessarily local. Also, some ordering algorithms are nondeterministic, i.e., running the same algorithm on the same graph will yield different results, making the problem even more difficult.

One way to solve this problem is to find efficient and effective methods for upper/lower bounds on the increase/decrease in the amount of fill when a vertex is added/removed, if computing the exact change is not possible, or very expensive to compute. When we have these bounds, we can move a vertex from one part to another and approximate the load of a part after the vertex is added/removed. This also gives a tentative ordering of the vertices, avoiding nondeterminism of ordering algorithms.

It is easier to predict the change in fill if k is small, because fill does not propagate very much. In one-level fill for instance, it suffices to count non-adjacent pairs of neighbors of the vertex, and for two-level fill, it suffices to look at non-adjacent pairs within a distance of at most two. Obviously, this becomes more and more expensive as k increases; we have to traverse the whole graph for exact factorization. For exact factorization it is more efficient to use bounds on the change in the loads of subdomains. Below we describe an efficient heuristic that works for finite element meshes. It depends on two observations. The first is a well-known theorem about direct methods for sparse matrices [38]. Below we paraphrase this result.

Let v be a vertex in the elimination graph. At any point in elimination, v will have fill edges to all those vertices that it can reach through previously eliminated vertices.

The second observation is specific for finite-element meshes:

Any two neighbors of a vertex v are reachable through only the neighbors of v.

We will use these two observations to localize the affects of vertex addition in the factored

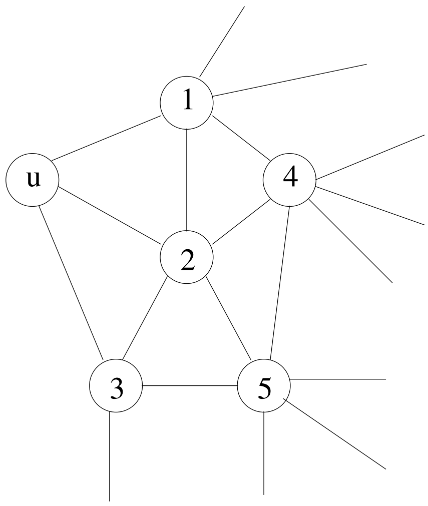


Figure 6.5: Example for bounding fill in vertex additions.

graph. Consider the example in Figure 6.5. Assume we want to insert vertex u, to the rest of the graph, which is already factored. If we insert vertex u right after all of its neighbors, then there will not be any changes in the factorization of the rest of the graph. Notice that neighbors of u are already connected, thus u does not add any reachabilities. In general, when we add a vertex u right after all of its neighbors in a given ordering, we do not add any reachabilities to the graph, since neighbors of u have already reached each other, thus addition of u does not make any other two vertices reach each other. This helps us to conclude that addition of u after all its neighbors will not change the rest of the graph, and we only need to consider the fill on u itself. With a similar reasoning, u can only reach to vertices that are already reached by its neighbors. Since neighbors of u are connected, the neighbor of u that is eliminated last, reaches all vertices a neighbor of u, thus u itself, can reach to. With this observation, the edges incident to u will be all edges incident its last neighbor in the ordering and edge to connect these two vertices. This gives us an effective and efficient-to-compute bound on the increase in the load of a domain when a vertex is added.

As for the lower bound, when a vertex is removed, total load decreases due to the load on this vertex and the fill generated due to the reachabilities this vertex provides. A trivial lower bound can be achieved by ignoring the second term, which will be very efficient but less effective. An alternative is to refactor the matrix, which will be less efficient but very effective. Using the observations above, refactorization can be restricted to a window, that starts with the vertex being eliminated and ends with its last neighbor in the ordering. A vertex will not cause any fills after its last neighbor, since it does not add any reachabilities anymore, thus there is no need to continue refactorization.

The lower and upper bounds described above can be used to predict the change in loads of parts when vertices are moved as the basis of an iterative improvement algorithm.

6.7 Conclusions

We addressed several problems where traditional graph partitioners fail to provide balanced partitions. In these problems, it is not possible to predict the load of a vertex a priori, which is crucial for the traditional partitioners. We presented a framework to solve these partitioning problems with complex objectives and applied it to achieve balanced overlapped domains and to balance computation plus computations. We designed data structures that can be updated locally, thus efficiently after moving a vertex to another part. Our experiments showed the effectiveness and efficiency of our methods. We also discussed how to solve the graph partitioning problem when incomplete/complete factorizations will be used in each subdomain. We presented methods to bound the change in the loads of domains, when a vertex moves from one part to another.

Acknowledgments

We are indebted to Mike Heroux for bringing the problem of partitioning for overlaps to our attention, and also for providing test problems. We also appreciate Edmond Chow's encouragement and his provision of one of our test problems.

Chapter 7

Exploiting Flexibly Assignable Work to Improve Load Balance

In many applications of parallel computing, the distribution of data among processors implies a corresponding distribution of work. However, there are important exceptions to this rule that arise for one of two reasons. First, some portions of the data may be replicated on multiple processors, any of which could perform the associated work. Second, tasks may involve multiple data items which may not all reside on the same processor. Thus, all the interacting data will need to be combined on a single processor before the computation can be completed. In principle, any processor could perform this task [57], but for the purposes of this chapter we will consider only those processors owning a portion of the relevant data—other options would increase the communication requirements.

Examples of such *flexibly assignable* work are common in scientific applications. In molecular dynamics simulations, a force is computed between any pair of particles that are close to each other. For large problems, these calculations are usually parallelized by dividing the particles among the processors [83]. If two close-by particles reside on different processors, then either processor could perform the computation. Kalé et al. [57] worked on heuristics to improve load balance by cleverly assigning such pairs in their molecular dynamics code, NAMD [56].

Another example arises in finite element simulations. These calculations consist of several computational phases, some of which are element based while others are node based. If, for instance, the mesh is partitioned so that processors own full elements, then nodes at the boundary between elements will be duplicated on at least two processors. Any of these processors could perform the node based operations for these shared nodes. If instead the mesh is partitioned by nodes, then some elements will be divided among multiple processors. Any of these processors could be employed to perform the element computation.

A third example comes from an important class of preconditioners known as overlapped Schwartz domain decomposition [82, 91]. In this preconditioning scheme, processors perform one calculation on subdomains that overlap each other, and another calculation on disjoint subdomains. With the overlapped domains, some portions of the data are duplicated on multiple processors. Any of these processors could perform the calculations for these duplicated objects in the disjoint portion of the computation.

The freedom to assign work to any of several processors raises the question of how best to exploit this flexibility. In this chapter we investigate using this freedom to improve load balance. That is, we want to give most of this flexibly assignable work to processors that would otherwise have too little to do. More formally, we address the following *task assignment problem.*

Given: *A set of tasks, consisting of their costs and the (possibly singleton) set of processors that can perform them.*

Find: *An assignment of tasks to processors that minimizes the number of tasks assigned to the maximally loaded processor.*

Despite its practical utility, to our knowledge, this problem has not been defined or addressed previously. After providing some basic definitions in §7.1, we investigate several combinatorial approaches to address the task assignment problem in §7.2. First, we describe a parametric search solution that uses a standard maximum-flow solver as a probe function. This solution is simple to implement and allows for the use of any flow solver as a black box.

Our second combinatorial algorithm involves a more detailed analysis of the structure of the problem. Specifically, we devise a form of maximum-flow/minimum-cut theorem for our objective function. This result gives significant insight into the structure of the problem, and we use it to devise an augmenting path algorithm that mimics the structure of Ford–Fulkerson methods for maximum flows. The result is an asymptotically more efficient approach, but one that cannot be built upon standard maximum-flow solvers. It is also worth noting that our approach solves the problem of finding a maximum flow with the property that the largest flow on any terminal edge is minimized.

These combinatorial algorithms are sufficient for many problems in which a serial computation can determine the assignment as a preprocessing step to a parallel calculation. However, in some instances the characteristics of the parallel computation change over time, and the assignment must be recomputed. Our combinatorial methods are not particularly amenable to parallelization. For this reason, in §7.3 we present a continuous approximation to the problem that leads to a more easily parallelized numerical approach. The continuous approximation is closely related to the diffusion methodology widely employed for determining work transfers in dynamic load balancing [20]. We show that our formulation can be reduced to a different numerical kernel known as a linearly-constrained least squares problem. We then show that there are efficient parallel approaches to solve this problem. Of course, the discretized solution to the continuous approximation may not be identical to the actual solution to the discrete problem.

Finally, in § 7.4 we discuss how the three techniques can be enhanced for heterogeneous systems, where processors compute at different speeds. Modifications are simple and do not change the complexities of the proposed solutions.

7.1 Preliminaries

A flow network is defined by a directed graph $G = (V, E)$, with a source vertex s, a terminal (sink) vertex t, and a capacity for each edge (i, j), which we denote by $c(i, j)$.

We will define a *flow* f to be a function $f : E \to Z^+$ from edges to integers, and use $f(i, j)$ to denote the volume of flow along edge (i, j). A flow must satisfy the capacity constraints on edges (i.e. $f(i, j) \leq c(i, j)$), and the flow conservation constraints

$$\sum_{(i,k)\in E} f(i,k) = \sum_{(j,i)\in E} f(j,i) \text{ for all } i \in V \setminus \{s,t\}.$$

The value of a flow $|f|$ is defined by the flow leaving the source s,

$$|f| = \sum_{(s,v)\in E} f(s,v).$$

A *maximum flow* (max-flow) is a flow that maximizes $|f|$.

A flow is *complete* if its value is equal to the cumulative capacity of edges leaving the source, i.e., $|f| = \sum_{(s,v)\in E} c(s,v)$. A graph that can support a complete flow will be called a *complete-flow* graph.

Given a graph G and flow f, the *residual graph* G_f has the same set of vertices as G and all edges in G (referred to here as *forward* edges), plus a matching set of *backward* edges that point in the opposite direction. The capacity of a forward edge is equal to its capacity in G minus the flow assigned to that edge in f. The capacity of a backward edge is equal to the flow on the corresponding forward edge.

In the max-flow problem, an augmenting path is defined as a path from s to t along which more flow can be pushed. The capacity of a path is defined by the minimum of the capacities of its edges. Any path in G_f from s to t with nonzero capacity is an augmenting path and can be used to increase the total flow.

Finding a maximum flow is a fundamental problem in combinatorial algorithms and has been the subject of numerous research efforts. Fundamentals of network flow algorithms can be found in [18, 92]. In a more recent work, Goldberg and Rao give a history of maximum-flow bounds and relevant references [39].

The assignment of tasks to processors can be modeled as a flow on a network $G = (T, P, E)$, where each task is represented by a vertex in T, and each processor is represented by a vertex in P. All processor–vertices are connected to the terminal t by *terminal edges*, and the source s is connected to all the task–vertices by *source edges*. Task–vertices have *assignment edges* connecting them to all the processors, which the associated task can be assigned to. The graph can be simplified by combining all vertices that have identical sets of processor neighbors. We call such sets *task groups*. An example can be found in Figure 7.1.

The capacity of an edge from the source to a task–vertex is defined by the size of the corresponding task group. We will set the capacities for assignment edges and terminal edges to be infinite. We will call this graph an *assignment* graph. Notice that assignment graphs are complete–flow graphs.

Figure 7.1 illustrates an assignment graph. There are 7 task groups and 4 processors. v_1 corresponds to a task group of 10 tasks and can be assigned to processors p_0, p_1, and/or p_2. Notice that some of the task groups can be assigned to only one processor. This situation often arises in

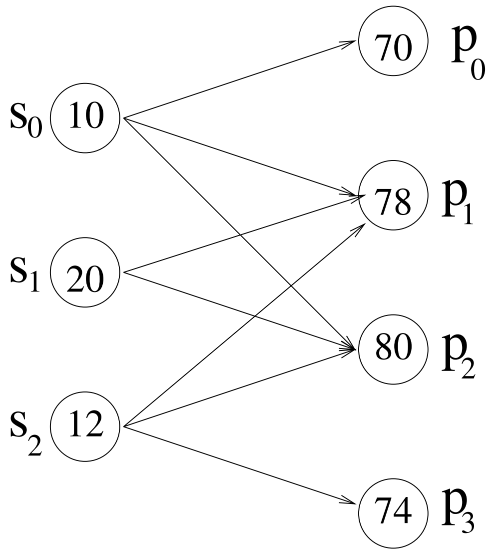


Figure 7.1: Example of an assignment graph.

practice, and these tasks correspond to work that can be assigned only to a single processor.

We can consider the assignment of tasks as a flow from the task–vertices to processor–vertices. Our objective is to find a flow that assigns all the tasks to processors while minimizing the maximum load of any processor. We need the flow to be complete (full capacity of source edges is used) to guarantee assignment of all the tasks. We can define the flow problem as follows.

Given an assignment graph $G = (T, P, E)$, find a complete flow f in G that minimizes

$$\max_{p \in P} f(p, t). \tag{7.1}$$

Although we define our flow problem for very specific flow graphs, the algorithms and analysis in the following sections are valid for any complete graph. For a general graph, we would redefine P in (7.1) to be the set of vertices that are connected to the terminal vertex (i.e., $P = \{v : (v, t) \in E\}$). Thus, the problem we are solving is equivalent to that of finding a maximum flow that minimizes the largest flow along any terminal edge.

In Figure 7.2, a solution to an assignment problem is illustrated. Numbers on the edges correspond to flow assignments for these edges. This assignment gives an optimal solution where p_1 and p_2 are the maximally loaded processors with 89 tasks. Out of 20 tasks of task group v_3, 11 will be performed on p_1 and 9 will be performed on p_2.

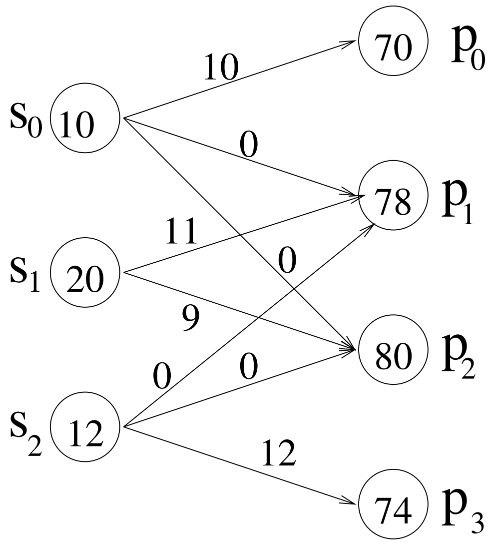


Figure 7.2: Example of a solution on an assignment graph.

7.2 Combinatorial Solutions

As outlined in §7.1, assignment of tasks to processors can be formulated as a flow in a network. In this section, we will investigate the relation between classical flow problems and the task assignment problem. First, we will discuss parametric search solutions that use standard max-flow techniques as a probe function, and describe a polynomial time algorithm for the task assignment problem. Then we will show how Ford–Fulkerson methods can be used to solve our problem. Specifically, we will revise the definition of augmenting paths and cuts, and show that any maximum-flow algorithm based on the Ford–Fulkerson method can be used for the task assignment problem.

7.2.1 Parametric Search

A parametric search algorithm has two components: a *probe* function that determines whether there is a solution with a cost less than a specified value, and a method to search on the space of candidate optimal solution values. Below, we first show how standard maximum-flow algorithms can be used as a probe function for our problem. Our cost function is the maximum work assigned to any single processor. Then we discuss two techniques to search the space of candidate values. The following lemma formalizes our claim for maximum-flow algorithms being used as a probe function.

Lemma 7.2.1 *A maximum-flow solver can be used to determine whether there is a solution to the task assignment problem with cost* $\leq B$.

Proof: Construct the assignment graph as described in §7.1 and change the capacities of all terminal edges to B. We claim that there is a solution to the task assignment problem with cost $\leq B$ if and only if the maximum flow uses the capacity of all the source edges. Proof of this claim follows.

Sufficiency. Bounds on capacities of sink edges guarantee that no processor is assigned more than B units of work, and if a flow uses all the source edge capacity then all work is assigned to processors. Moreover, the flow solution provides the corresponding task assignments.

Necessity. Assume there is a solution to the task assignment problem where no processor is assigned more than B units of work. We can use the assignments of tasks in this solution to find a corresponding flow solution. □

To solve the task assignment problem we must find the minimal value of B for which a max-flow solution uses all the source edge capacities. We present two algorithms for finding this value in the following two subsections.

Bisection Search

Bisection search is a standard technique used in parametric search algorithms. It starts with a lower and an upper bound on the optimal solution value, and discards half of the interval by probing on the midpoint of the current bounds. This gives an ϵ-approximation algorithm for real–valued solutions, but finds an exact solution when the optimal solution is an integer, as in the case of our problem.

For the task assignment problem, the total number of tasks is an upper bound on the cost of an assignment, and the number of tasks divided by $|P|$ is a lower bound. Thus, a bisection search gives the following result.

Theorem 7.2.2 *If* W_T *is the total number of tasks, bisection search solves the task assignment problem optimally with* $O(\log W_T)$ *probe calls.*

Incremental Search

An incremental search starts with a lower bound for the optimal solution value and increases it until the optimal value is found. The increments should be small to avoid missing the optimal value, but large for efficiency. The following lemma and theorem show how the lower bound can be increased after a failed probe call—that is, a max-flow problem with terminal edge capacities B in which not all the source edge capacity is utilized.

Lemma 7.2.3 *Let (u, t) be a terminal edge that is not saturated in a maximum-flow solution f for a probe value B (i.e., $f(u, t) < c(u, t) = B$). Then for any probe value $B' > B$, there is a maximum-flow solution f' in which $f'(u, t) \leq f(u, t)$.*

Proof: When the Ford-Fulkerson method is used to achieve an optimal solution f' for bound $B' > B$ by using f as an initial solution, we can get an optimal solution with $f'(u, t) \leq f(u, t)$. First, note that u is not reachable from s in G_f, and increasing terminal edge capacities does not make u reachable from s. Furthermore, u will not be reachable while the flow is being modified via augmenting paths. Consider the first augmenting path that will add a vertex to the set of vertices u is reachable from. Observe that such a path should reach a vertex that can reach to u, which contradicts u's non-reachability from s. □

Theorem 7.2.4 *For a failed probe with sink edge capacity B, let $W_r > 0$ be the total unused source edge capacity, and let K be the number of saturated terminal edges. Then there is no feasible solution to the task assignment problem with cost less than $B + W_r/K$.*

Proof: By the result of Lemma 7.2.3, additional flow cannot go to any of the unsaturated terminal edges. In the best case, additional flow will be equally distributed among the set of saturated terminal edges. □

As a result of Theorem 7.2.4, a failed probe value B can be increased to $B + \frac{W_r}{K}$. Algorithm IncSearch presented in Figure 7.3 exploits this result.

The following lemma proves that Algorithm IncSearch terminates and gives a bound on the number of probes it makes.

Lemma 7.2.5 *Algorithm IncSearch terminates after at most $|P|$ probes.*

```
Algorithm IncSearch
    B ← W_T/|P|;
    while not Probe(B)
        Let K be the number of saturated terminal edges,
        and W_r be the total unused source edge capacity.
        B ← B + W_r/K;
    return B;
```

Figure 7.3: Incremental search algorithm IncSearch

Proof: When a probe value is increased, if all previously saturated edges remain saturated then the probe call will succeed. Thus, when a probe fails, at least one new terminal edge is not saturated. That is, each failed probe decreases the number of saturated terminal edges by at least one. □

Theorem 7.2.6 *Algorithm IncSearch finds an optimal solution and makes $O(|P|)$ probes.*

Proof: We start with a lower bound. According to Theorem 7.2.4, increases on B are minimal. Thus we do not miss the optimal value. Lemma 7.2.5 ensures that the algorithm terminates after $O(|P|)$ probes with an optimal solution. □

Notice that successive probes solve max-flow problem on the same graph in an incremental manner, where only the capacities of the terminal edges increase. Thus the previous flow solution gives a feasible solution (though not optimal) for the next flow solution, which might be exploited for efficiency. Using Ford-Fulkerson method in its simpliest way will give a complexity of $O(W_T * |E|)$ for all the probes, and thus for the algorithm.

7.2.2 Ford–Fulkerson Method

The Ford–Fulkerson method has been the basis of a number of algorithms to solve the max–flow problem. It is built on three basic concepts: *residual* graphs, *augmenting* paths, and *cuts* [18]. In this section, we will discuss how it can be adopted to the task assignment problem. First, we will revise the definitions of augmenting paths and cuts for the task assignment problem, then state and prove a version of the maximum-flow/minimum-cut theorem for the task assignment problem.

The result will enable any algorithm based on the Ford–Fulkerson method to be used to solve our problem.

The generic Ford-Fulkerson method starts with a zero flow and continues to add to the flow along augmenting paths until no augmenting paths are left. In the task assignment problem, we will use augmenting paths to shift flow (tasks) from a maximally loaded terminal edge (processor) to a less loaded terminal edge. Formally, an augmenting path (pt, u, v) is a path pt in G_f that starts with the vertex u of a maximally loaded terminal edge (u, t), ends at the vertex v of a less-loaded terminal edge (v, t), and does not go through t. We define the *capacity* $c(pt, u, v)$ of an augmenting path to be the minimum of the capacities of its edges and half of the difference between the flow on the first and last terminal edges, rounded down to an integer,

$$c(pt, u, v) = \min(\lfloor \frac{f(u,t) - f(v,t)}{2} \rfloor, \min\{c_f(i,j) : (i,j) \text{ on } pt\}).$$

This implies that the capacity of a path between two processors whose loads differ by just one is zero, since such an augmentation will not yield a more balanced distribution.

We can update flow assignment in the graph for edges on the path and the two terminal edges connecting processor-vertices to the terminal to obtain a more balanced distribution, as stated by the following lemma.

Lemma 7.2.7 *Let f be the current flow, pt be an augmenting path, and $c > 0$ be the capacity of this path. Define $f^+(i,j)$ for all edges as*

$$f^+(i,j) = \begin{cases} f(u,t) - c & \text{if } i = u, j = t \\ f(v,t) + c & \text{if } i = v, j = t \\ f(i,j) + c & \text{if } (i,j) \text{ is on } pt \\ f(i,j) - c & \text{if } (j,i) \text{ is on } pt \\ f(i,j) & \text{otherwise} \end{cases} .$$

Then f^+ does not change the total flow (i.e., $|f^+| = |f|$), but decreases either the maximum load or the number of maximally loaded terminal edges.

Proof: For the total flow to change we must decrease flow from s. This is possible only if there is an edge (v, s) in pt, but this edge must be followed by another edge (s, u), and thus total flow from s does not change. With the same argument, flow conservation constraints are satisfied for all vertices. Since pt does not go through t, augmentation will affect only two terminal edges: (u, t) and (v, t). By definition of an augmenting path, (u, t) is a maximally loaded terminal edge, and we decrease its load. By definition of the capacity of an augmenting path, the load of (v, t) cannot be as high as (u, t), after we increase it. □

In traditional flow problems, a *cut* (S, T) in G is defined as a partition of vertices into S and T in which $s \in S$ and $t \in T$. The cost of a cut is defined as the sum of capacities of edges from S to T. The cost of a minimum cut and value of a maximum flow are equal. A minimum cut corresponds to a bottleneck in the flow from source to terminal. For the task assignment problem, we will define a *cut* as a bipartitioning (P_1, P_2) of the processor–vertices. The *cost* of a cut is defined to be the maximum load in P_1 when

(i) processors in P_1 are equally loaded,

(ii) all the tasks that can be assigned to a processor in P_2 are assigned to processors in P_2.

By "equally loaded" we mean that the loads of any two processors differ by at most one. Cuts will help to identify a bottleneck in the problem, just as in maximum flow problems. A bottleneck in our problem is a group of processors that have to perform a large set of tasks. Unlike the maximum flow problem, cuts provide lower bounds on the cost.

With the above definitions, we have the following maximum-flow/minimum-cut theorem for the task assignment problem.

Theorem 7.2.8 *The following statements are equivalent:*

1. *Flow f minimizes the load of the maximally loaded processor.*

2. *There is no augmenting path that decreases the maximum load in G_f.*

3. *The maximum load is equal to the cost of a cut (P_1, P_2).*

Proof:

- (1) $\implies$ (2) Assume the contrary, that there exists an augmenting path to decrease the maximum load. Then we can use this path to decrease the maximum load, so f is not an optimal flow.

```
Algorithm AugPath
    find a complete flow f in G;
    while there is an augmenting path pt
        augment flow along pt;
```

Figure 7.4: Algorithm based on augmenting paths

- (2) $\Longrightarrow$ (3) Let P_1 be the set of processors with maximum load plus processors reachable from a maximally loaded processor in G_f. The set P_2 contains the remaining processors. By construction, there are no augmenting paths from P_1 to P_2. This guarantees that all tasks between P_1 and P_2 are assigned to P_2 processors. Also, since there are no augmenting paths in G_f, the loads of all processors in P_1 are either equal to or one less than the maximum load. That is, the processors in P_1 are equally loaded.

- (3) $\Longrightarrow$ (1) Since all tasks which could be assigned to either P_1 or P_2 are assigned to processors in P_2, the work currently assigned to processors in P_1 must be performed by processors in P_1. The best we can do is to assign all work equally, which is guaranteed by the first condition in the definition of a cut, so f is an optimal solution.

$\square$

Corollary 7.2.9 *Any algorithm based on the Ford–Fulkerson method can be used to solve the task assignment problem.*

It is worth noting that although any algorithm based on the Ford–Fulkerson method might be used to solve this problem optimally, the complexity results might vary from those of the conventional max–flow problem.

In Figure 7.4, we present an algorithm AugPath, which finds an optimal solution using augmenting paths.

Theorem 7.2.10 *Algorithm AugPath in Figure 7.4 finds an optimal solution in $O(|E| * \log |P| * W_T)$-time.*

Proof: Correctness of the algorithm is implied by Corollary 7.2.9. Finding an augmenting path takes $O(E)$-time in the worst case. By Lemma 7.2.7, each augmenting path either decreases the maximal load or the number of maximally loaded processors. This gives a loose $|P| * W_T$ bound on the number of augmenting paths required. A better bound is possible however. When the maximum load is in the range $[W_T/2, W_T]$, only one processor might have the maximum value, and thus each augmenting path will decrease the maximum load by one. Generally, when the maximum is in the range $[W_T/(k+1), W_T/k]$ there can be at most k processors with the maximum load and k augmenting paths may be needed to decrease the maximum load. So the total number of augmenting paths can computed as

$$
\begin{aligned}
\sum_{1 \leq k \leq |P|-1} \left(\frac{W_T}{k} - \frac{W_T}{k+1} \right) k &= W_T \sum_{1 \leq k \leq |P|-1} \left(\frac{k+1-k}{k(k+1)} \right) k \\
&= W_T \sum_{1 \leq k \leq |P|-1} \frac{1}{k+1} \\
&= O(W_T * \log |P|).
\end{aligned}
$$

□

7.3 Numerical Solution

The flow formulations described above provide efficient, serial algorithms for optimizing task assignments. Unfortunately, flow algorithms are difficult to parallelize, particularly for large numbers of processors. In this section we describe a continuous version of the problem and show that it reduces to a well–studied numerical computation. Although this approach does not provide an integral solution, its parallelizability may make it preferable for many applications. In spirit, our approach is similar to the widely used diffusion methods to determine how much work to transfer between processors in dynamic load balancing [20, 25, 27].

In our numerical formulation, each task group generates an equation. Say the task group has m tasks in it. If the task group can be assigned to any of k processors, then there will be k unknowns associated with the task group. Each of these unknowns $x_1, \ldots, x_k$ encodes the assignment of the corresponding task group to one of the processors. In a discrete formulation we would want the x_i values to be integral, but in our continuous formulation we impose the following, weaker, set of equality and inequality constraints.

$$\sum_{i=1}^{k} x_i = m, \quad \text{and} \quad x_i \geq 0 \;\; \text{for all} \;\; 1 \leq i \leq k$$

The x values can be used to encode the work assigned to each of the $|P|$ processors. The task group with k potential processors will generate k columns of length $|P|$. The ith column is all 0's except for a single 1 in the row number that corresponds to the processor associated with x_i. These columns can be treated as a matrix, and multiplying the x vector by this matrix gives a $|P|$ length vector containing the work assigned to each of the processors.

We can continue this construction, adding variables, constraints, and work contributions from all $|T|$ tasks. Letting $|Q|$ denote the sum over all tasks of the number of processors that task could be assigned to, we obtain the following problem.

$$\min_{x} \|Ax\|_\infty \;\; \text{subject to} \;\; Bx = d \;\; \text{and} \;\; x \geq 0, \tag{7.2}$$

where A is $|P| \times |Q|$, B is $|T| \times |Q|$, and both have only a single 1 in each column. In the flow terminology from §7.1, the x vector is the assignment of a (possibly fractional) flow to the edges from source–adjacent nodes to sink–adjacent nodes (see Figure 7.1). Ax is the amount of flow into each sink–adjacent node, and Bx is the flow out of each source–adjacent node. So $Bx = d$ merely encodes the flow preservation property for each source–adjacent node in a complete flow. The ∞-norm reflects our desire to minimize the work of the maximally loaded processor. It is worth remarking that tasks that can be performed only by a single processor can be removed from the variable set, reducing the size of the problem. For the example from Figure 7.1, we get the following matrices in our numerical formulation.

$$A = \begin{pmatrix} 1 & 1 & & & & & & & & & & \\ & & 1 & & 1 & 1 & & & 1 & & & \\ & & & 1 & & & 1 & 1 & & 1 & & \\ & & & & & & & & & & 1 & 1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & & & & & & & & & & & \\ & 1 & 1 & 1 & & & & & & & & \\ & & & & 1 & & & & & & & \\ & & & & & 1 & 1 & & & & & \\ & & & & & & & 1 & & & & \\ & & & & & & & & 1 & 1 & 1 & \\ & & & & & & & & & & & 1 \end{pmatrix} \qquad d = \begin{pmatrix} 70 \\ 10 \\ 78 \\ 20 \\ 80 \\ 12 \\ 74 \end{pmatrix}$$

Unfortunately, the ∞-norm is not smooth and so can be difficult to minimize. A similar problem arises in the diffusion approach where a 1-norm is approximated by the smooth alternative of a 2-norm. We will similarly choose to replace our ∞-norm with a 2-norm as follows.

$$\min_x \|Ax\|_2 \text{ subject to } Bx = d \text{ and } x \geq 0 \tag{7.3}$$

This formulation is an instance of what is known as a *linearly-constrained least squares* problem. Unlike the diffusion methodology, in our case we lose nothing by transforming to a smoother approximation since for our problem a solution to the two norm problem also solves the ∞-norm problem. This is shown by the following theorem.

Theorem 7.3.1 *Any x that solves Problem (7.3) is also a solution for Problem (7.2).*

Proof: Begin by noting that Problem (7.2) is identical to the flow problem from §7.2, except that variables are continuous instead of discrete. The analysis from §7.2 generalizes to continuous variables in a straightforward manner, so the structure of a solution to Problem (7.2) is the same as the structure of a solution to the flow formulation. Specifically, a subset of processors (P_1) will have the same maximum load (l_{max}), and all tasks that can be assigned to processors with smaller loads (P_2) have been so assigned.

Now let x be a solution to Problem (7.3). This problem has the same constraints as Problem (7.2), so it shares the same space of feasible solutions. As with the solution to Problem (7.2), the processors in P_1 will need to be assigned at least a total of $|P_1|l_{max}$ load. It is easy to show that the contribution of these processors to the 2-norm is minimized when all their loads are equal. Any transfer of work from processors in P_2 to processors in P_1 will only increase the 2-norm. So a solution to Problem (7.3) must also be a solution to Problem (7.2). $\square$

Least squares problems are fundamental to linear algebra (see, e.g.. [11]). Constrained least squares problems have been studied by several authors. Of interest to us are iterative methods that are amenable to parallelization. Note that since we are using the continuous formulation as an approximation to a discrete problem, a low accuracy numerical solution is sufficient.

One way to deal with linear equality constraints is the method of *weighting* [11]. This method moves the equality constraints into the objective function, but severely penalizes slack in these rows by weighting these equations. So Problem (7.3) is transformed to the form

$$\min_x \left\| \begin{pmatrix} \gamma B \\ A \end{pmatrix} x - \begin{pmatrix} \gamma d \\ 0 \end{pmatrix} \right\|_2 \quad \text{subject to} \quad x \geq 0,$$

where γ is a large number used to penalize slack in the equality constraints. This transforms our problem to an instance of a nonnegative least squares problem,

$$\min_x \|Cx - b\|_2 \quad \text{subject to} \quad x \geq 0.$$

The nonnegative least squares problem is equivalent to the following optimization problem.

$$\min_x \frac{1}{2} x^T E x - c^T x \quad \text{subject to} \quad x \geq 0,$$

where $E = C^T C$ and $c = C^T b$. Cryer proposed the following SOR iteration for solving nonnegative least squares problems [19],

$$x_i^{k+1} = \max \left\{ 0, x_i^k - \frac{\omega}{e_{ii}} \left(c_i - \sum_{j<i} e_{ij} x_j^{k+1} - \sum_{j \geq i} e_{ij} x_j^k \right) \right\}.$$

This is a standard stencil operation and requires only local communication with neighbor processors. Hence, it is amenable to efficient parallelization.

7.4 Heterogeneous Systems

Discussions prior to this section were limited to homogeneous systems, where all processors have identical execution speeds. This assumption does not hold, however for many current parallel systems; processors are often heterogeneous, i.e., they have different processing powers. In particular, clusters of workstations, which are very likely to have heterogeneity, are gaining popularity, and significant research efforts are devoted to their development. Heterogeneity of the processors might lead to load imbalance even when all processors are assigned equal amounts of work. Varying processor speeds should be taken into account while distributing the work to avoid processor idle times. The objective must be to minimize the maximum completion time among all processors, as opposed to minimizing the maximum load.

In the following sections, we will discuss how our techniques can be extended to exploit flexibly assignable tasks to improve balance for heterogeneous processors. We define execution speed of a processor to be the number of unit operations it can perform in a unit time, and use e_i to denote the execution speed of the ith processor.

7.4.1 Parametric Search Algorithms

To employ a parametric search algorithm from § 7.2.1, we first need a probe function. For heterogeneous problems we can still use a max-flow solver as a probe after minor modifications. Here probe values will be the completion time of processors, i.e., $Probe(B)$ decides if there is an assignment of tasks for which all processors can complete in B units of time. By defining the capacity of a terminal edge (u, t) as $B * e_u$, where e_u is the processing speed of the respective processor, we can guarantee that all processors will complete in B units. We also need all tasks to be assigned to processors, which is guaranteed by a complete flow in the graph. Thus, a complete flow in the graph gives a feasible solution to the task assignment problem. Similarly, a feasible assignment of tasks defines a flow on the graph, and thus a max-flow solver can be used to determine the existence of a solution within a specified completion time.

For the bisection algorithm from § 7.2.1, a lower bound LB to start can be chosen as the ideal completion time, where the load is distributed perfectly, i.e.,

$$LB = \frac{W_T}{\sum_{i=1}^{P} e_i}.$$

Assigning all tasks to the fastest processor will give an upper bound

$$UB = \frac{W_T}{\max_{i=1}^{P} e_i}.$$

To use incremental search, we previously increased the bound hoping that unassigned work will be distributed evenly among active processors. We can still use the same idea, but this time we should consider the total execution power—as opposed to the number—of active processors. Let A_f denote the set of processors whose terminal edge is saturated in flow f, i.e., $A_f = \{u : f(u,t) = c(u,t)\}$, and W_r be the unused source edge capacity. Then after a failed probe value B, we can increase the bound to $B + \frac{W_r}{\sum_{u \in A_f} e_u}$.

Notice that these enhancements do not alter the complexities of the proposed algorithms.

7.4.2 Ford–Fulkerson Method

The Ford-Fulkerson method of Section 7.2.2 can be used to solve this problem for heterogeneous processors after minor modifications of the basic definitions. We define the residual graph in the same way. An augmenting path is still a path in the residual graph from the vertex of an overloaded terminal edge to the vertex of an underloaded terminal edge, and thus helps us to shift work from an overloaded processor to an underloaded processor to improve balance. The capacity of a path is a function of the edge capacities on the path and the load difference between the two processors. Edge capacities constrain how much work can be shifted from the first processor to the last processor. After shifting load from one processor to another, we want the execution times of two processors to be equal for better balance. To equalize completion times of two processors, we need to shift

$$d(u,v) = \frac{e_v f(u,t) - e_u f(v,t)}{e_u + e_v}$$

units of work from processor u to processor v, which can be achieved using simple algebra. Considering integral assignments, the capacity of a path pt can be defined as

$$c(pt, u, v) = \begin{cases} \min(\lfloor d(u,v) \rfloor, \min\{c_f(i,j) : (i,j) \text{ on } pt\}) & \text{if } e_u > e_v \\ \min(\lceil d(u,v) \rceil, \min\{c_f(i,j) : (i,j) \text{ on } pt\}) & \text{otherwise} \end{cases} .$$

The two conditions in the definition assign the extra unit of work to the faster processor.

A cut is still defined as a bipartitioning of processors (P_1, P_2). The cost of a cut is the maximum load among P_1 processors, when all tasks that can be assigned to a P_2 processor are assigned to a P_2 processor and remaining tasks are distributed so that completion times of P_1 processors are equal.

With the modified definitions, Theorem 7.2.8 still holds, so we can use any algorithm based on augmenting paths to solve this problem.

7.4.3 Numerical Formulation

In Section 7.3, we used the Ax vector to define the loads of processors. To determine the completion time for each processor, we can use EAx, where E is a diagonal matrix with the ith diagonal entry the reciprocal $1/e_i$ of the execution speed of ith processor. Minimizing $\|EAx\|_\infty$ will minimize the maximum completion time among all the processors. A 2-norm minimum solution is still sufficient for minimizing the ∞-norm, and the proof is similar to that for homogeneous systems. Our numerical formulation for heterogeneous systems still reduces to a linearly constrained least squares problem.

7.5 Conclusions

In this chapter, we studied how to exploit flexibility in assignment of some tasks to improve load balance. Distribution of the data often dictates distribution of work among processors, but there are exceptions where some tasks, which we call flexible-assignable tasks, can be assigned to one of several processors without altering the total volume of communication. We first modeled the problem as a flow problem and used combinatorial techniques for its solution. Our parametric search algorithms use maximum flow algorithms for probing on a candidate optimal solution value.

We described two algorithms to solve the assignment problem with $\log W_T$ and $|P|$ probe calls, where W_T and $|P|$, respectively, denote the total workload and number of processors. Our second combinatorial technique is based on Ford–Fulkerson method, which underlies many of the max-flow algorithms. We defined augmenting paths and cuts for this problem, and showed that any algorithm based on augmenting paths can be used to find an optimal solution for the task assignment problem. We then considered a continuous version of the problem, and showed that it reduces to the well–studied linearly constrained least squares problem. The continuous version of the problem has the advantage of being easily amenable to parallelization. Finally, we studied how these techniques can be enhanced to solve the load balancing problem for heterogeneous processors.

Chapter 8

Conclusion

This thesis covered six combinatorial problems that are relevant to various problems in scientific computing, and is naturally located at the intersection of two disciplines: computer science on the one hand and computational science and engineering on the other. The problems were motivated by different scientific computing applications, and the solution techniques used fundamental methods of computer science. We reported both theoretical and practical results for these problems. Our work included complexity analysis and design and analysis of algorithms, as well as implementation and empirical studies.

In Chapter 2, we discussed improving the performance of matrix-vector multiplication, which is arguably the most important computational kernel in scientific computing. The main idea was to exploit regular patterns within the matrix to reduce extra load operations due to memory indirections. We packed nonzeros in consecutive positions into blocks to limit load operations to one per block. We also studied how to permute nonzeros of a matrix into contiguous positions. Experiments showed runtime improvements of up to 33% and improvements of 25% on average.

One-dimensional partitioning of workload arrays was the subject of Chapter 3. We proposed run-time efficient algorithms along with implementation tips to overcome the misconception that heuristics produce good solutions and exact algorithms are difficult to implement and not run-time efficient. Experiments showed that, on average, our algorithms are 100 times faster than a matrix-vector product for 64-way partitions.

Chapter 4 was on identifying strongly connected components in parallel, motivated by the discrete ordinates methods for modeling radiation transport. We designed an easy-to-parallelize divide-and-conquer algorithm based on reachability searches, the expected running time of which is $O(|E| \log |V|)$, where $|E|$ and $|V|$ are the number of edges and vertices, respectively.

Two communication algorithms to support adaptive computation were presented in Chapter 5. The first problem was concerned with determining the communication pattern after a calculation is first defined. After subdomains are assigned, processors can determine what data they need to receive, but will not know which processors contain those data. We proposed using a distributed *directory*—a rendezvous algorithm—where each processor is responsible for maintaining directory information for a subset of the components. The second problem studied how to migrate data subject to memory constraints. While redistributing data among processors, each processor needs to have space for both its outgoing and incoming data simultaneously, which might be a problem if memory is limited. We proposed sending data in phases. After each phase, processors can free the memory of the data sent, to make it available for the next phase. We proved that scheduling messages for the minimum number of phases is NP-Complete, and showed a reduction to multi-commodity flow. We also considered a continuous approximation that yields solutions with at most two more phases than that of a discrete solution. Finally, we devised a practical 1.5-approximation algorithm.

In Chapter 6, we studied graph partitioning for complex objectives. Traditional graph partitioners depend on *a priori* information on the load of a task, which is not always available for example in balancing computation plus communication or in domain decomposition techniques where incomplete or complete factorization is used at each subdomain, or with overlapped partitions where boundary vertices are duplicated. For these problems the load is a complicated function of the partition. We proposed a framework for addressing such complex partitioning problems and applied it to achieve equal-sized overlapped subdomains and balanced computation plus communication load.

Although distribution of data implies distribution of work in many applications, there are exceptions where some tasks can be assigned to one of several processors without altering total communication volume (e.g., force computation between two particles in different processors, or node computations in finite-element simulations when element partitions are used). In Chapter 7, we examined how to exploit this flexibility in assignment of tasks to improve load balance. We proposed parametric search algorithms that use maximum flow solvers for probing. We also defined augmenting paths and cuts for this problem, and showed that any algorithm based on augmenting paths can be used to solve this problem. Then we reduced the continuous version of the problem to the well-studied linearly constrained least squares problem, which is easily amenable to parallelization.

This thesis highlighted combinatorial techniques in scientific computing as a research field full of interesting problems with real impact. From the scientific computing point of view, our results show that very significant computational savings can be achieved by employing combinatorial techniques. Moreover, the complexity of problems often requires sophisticated combinatorial techniques, and using simple, brute-force techniques may be very inefficient. From the computer science point of view, it is easy to find challenging combinatorial problems where one can apply or adapt existing theories or develop new techniques for real-world applications. Working close to the applications not only helps to find new problems, but also forces one to start with the application and then solve problems, as opposed to developing solution techniques and then looking for applications. The breadth of this thesis demonstrates the abundance of combinatorial algorithms in scientific computing as a research field. Each of the six chapters of this thesis was motivated by a different application and the solutions employ different solution techniques. Our results, together with the importance of the problems studied, show the impact of our solutions on real-world applications. To conclude, combinatorial problems in scientific computing is a rich and important field, which should attract both computer science and scientific computing researchers, and might develop into a discipline of its own, built on the heritage of computer science and computational science and engineering.

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair, *A high performance algorithm using pre-processing for sparse matrix vector multiplication*, Proc. Supercomputing '92, (1992), pp. 32–41.

[2] A. Aggarwal and R. J. Anderson, *A random NC algorithm for depth first search*, Combinatorica, 8 (1988), pp. 1–12.

[3] A. Aggarwal, R. J. Anderson, and M.-Y. Kao, *Parallel depth-first search in general directed graphs*, SIAM J. Comput., 19 (1990), pp. 397–409.

[4] R. K. Ahuja, R. L. Magnanti, and J. B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1993.

[5] S. Anily and A. Federgruen, *Structured partitioning problems*, Operations Research, (1991), pp. 130–149.

[6] C. Aykanat and F. Ozguner, *Large grain parallel conjugate gradient algorithms on a hypercube multiprocessor*, in Proc. 1987 Int. Conf. on Parallel Processing, 1987, pp. 641–645.

[7] C. Aykanat, F. Ozguner, F. Ercal, and P. Sadayappan, *Iterative algorithms for solution of large sparse systems of linear equations on hypercubes*, IEEE Trans. Computers, 37 (1988), pp. 1554–1567.

[8] D. A. Bader, *A practical parallel algorithm for cycle detection in partitioned digraphs*, Tech. Rep. Technical Report AHPCC-TR-99-013, Electrical & Computer Eng. Dept., Univ. New Mexico, Albuquerque, NM, 1999.

[9] R. S. Baker and K. R. Koch, *An Sn algorithm for the massively parallel CM-200 computer*, Nuclear Science and Engineering, 128 (1998), pp. 312–320.

[10] D. J. Becker, T. Sterling, D. Savarese, U. A. Ranawak, and C. V. Packer, *Beowulf: A parallel workstation for scientific computation*, in Proc. 1995 Int. Conf. on Parallel Processing (ICPP'95), 1995, pp. I–11–I–14.

[11] A. Bjorck, *Numerical Methods for Least Squares Problems*, SIAM, 1996.

[12] S. H. Bokhari, *Partitioning problems in parallel, pipelined, and distributed computing*, IEEE Trans. Computers, 37 (1988), pp. 48–57.

[13] D. A. Burgess and M. B. Giles, *Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines*, Tech. Rep. 95/06, Numerical Analysis Group, Oxford University Computing laboratory, 1995.

[14] Ü. V. Çatalyurek and C. Aykanat, *Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication*, IEEE Trans. on Parallel and Distributed Systems, 10 (1999), pp. 673–693.

[15] P. Chaudhuri, *Finding and updating depth-first spanning trees of acyclic digraphs in parallel*, The Computer J., 33 (1990), pp. 247–251.

[16] H. Choi and B. Narahari, *Algorithms for mapping and partitioning chain structured parallel computations*, in Proc. 1991 Int. Conf. on Parallel Processing, 1991, pp. I–625–I–628.

[17] R. Cole and U. Vishkin, *Faster optimal prefix sums and list ranking*, Information and Computation, 81 (1989), pp. 334–352.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, MIT Press and McGraw-Hill, Cambridge, MA, 1990.

[19] C. Cryer, *The solution of a quadratic programming problem using systematic overrelaxation*, SIAM J. on Control and Optimization, 9 (1971), pp. 385–392.

[20] G. Cybenko, *Dynamic load balancing for distributed memory multiprocessors*, J. Parallel Distrib. Comput., 7 (1989), pp. 279–301.

[21] R. Cypher and S. Konstantinidou, *Bounds on the efficiency of message-passing protocols for parallel computers*, SIAM J. Comput., 25 (1996), pp. 1082–1104.

[22] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, *The design and implementation of a parallel unstructured euler solver using software primitives*, AIAA J., (1994), pp. 489–496.

[23] K. D. Devine, B. Hendrickson, E. G. Boman, M. M. St.John, and C. Vaughan, *Zoltan: A dynamic load-balancing library for parallel applications – user's guide*, Tech. Rep. SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 1999.

[24] K. D. Devine, B. A. Hendrickson, E. G. Boman, M. M. St.John, and C. Vaughan, *Zoltan: A dynamic load–balancing library for parallel applications — user's guide*, Tech. Rep. SAND99-1377, Sandia National Laboratories, Albuquerque, NM, 1999.

[25] R. Diekmann, A. Frommer, and B. Monien, *Efficient schemes for nearest neighbor load balancing*, Parallel Computing, (1999), pp. 789–812.

[26] M. R. Dorr and C. H. Still, *Concurrent source iteration in the solution of 3-dimensional, multigroup discrete ordinates neutron-transport equations*, Nuclear Science and Engineering, 122 (1996), pp. 287–308.

[27] R. Elsässer, B. Monien, and R. Preis, *Diffusive load balancing schemes on heterogeneous networks*, in Proc. 12th ACM Symp. Parallel Alg. Arch. (SPAA), 2000, pp. 30–38.

[28] C. Farhat and F. X. Roux, *An unconventional domain decomposition method for an efficient parallel solution of large-scale finite element systems*, SIAM J. Sci. Stat. Comp., 13 (1992), pp. 379–396.

[29] C. M. Fiduccia and R. M. Mattheyses, *A linear-time heuristic for improving network partitions*, in Proc. the 19th ACM/IEEE Design Automation Conf., 1982, pp. 175–181.

[30] L. Fleischer, B. Hendrickson, and A. Pinar, *On identifying strongly connected components in parallel*, Solving Irregularly Structured Problems in Parallel: 7th Int. Symp., Irregular '00, Lecture Notes in Computer Science, 1800 (2000), pp. 505–512.

[31] L. K. Fleischer, B. Hendrickson, and A. Pinar, *A divide-and-conquer algorithm for identifying strongly connected components*, SIAM J. Discrete Mathematics, (2001).

[32] G. N. Frederickson, *Optimal algorithms for partitioning trees and locating p-centers in trees*, Tech. Rep. CSD-TR-1029, Purdue University, 1990.

[33] ——, *Optimal algorithms for partitioning trees*, in Proc. Second ACM-SIAM Symp. Discrete Algorithms, 1991, pp. 168–177.

[34] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Fransisco, U.S.A., 1979.

[35] M. R. Garey, D. S. Johnson, and L. Stockmeyer, *Some simplified np-complete graph problems*, Theoretical Computer Science, 1 (1976), pp. 237–267.

[36] D. M. Gay, *Electronic mail distribution of linear programming test problems*, Mathematical Programming Society COAL Newsletter, (1985).

[37] H. Gazit and G. L. Miller, *An improved parallel algorithm that computes the BFS numbering of a directed graph*, Inform. Process. Lett., 28 (1988), pp. 61–65.

[38] A. George and J.-H. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[39] A. V. Goldberg and S. Rao, *Beyond the flow decomposition barrier*, J. ACM, 45 (1998), pp. 783–797.

[40] T. Hagerup, *Planar depth-first search in $O(\log n)$ parallel time*, SIAM J. Comput., 19 (1990), pp. 678–704.

[41] Y. Han, B. Narahari, and H.-A. Choi, *Mapping a chain task to chained processors*, Information Processing Letters, 44 (1992), pp. 141–148.

[42] P. Hansen and K. W. Lih, *Improved algorithms for partitioning problems in parallel, pipelined and distributed computing*, IEEE Trans. Computers, 41 (1992), pp. 769–771.

[43] M. T. Heath, *SCIENTIFIC COMPUTING: An Introductory Survey*, Mc Graw-Hill, 1997.

[44] B. Hendrickson and K. Devine, *Dynamic load balancing in computational mechanics*, Comp. Meth. Appl. Mech. Eng., (2000). Invited paper. To appear.

[45] B. Hendrickson and T. Kolda, *Graph partitioning models for parallel computing*, Parallel Comput., 26 (2000), pp. 1519–1534.

[46] B. Hendrickson and T. G. Kolda, *Partitioning rectangular and structurally nonsymmetric sparse matrices for parallel processing*, SIAM J. Scientific Computing, 21 (2000), pp. 2048–2072.

[47] B. Hendrickson and R. Leland, *The Chaco user's guide: Version 2.0*, Tech. Rep. SAND94–2692, Sandia National Labs, Albuquerque, NM, June 1995.

[48] M. Heroux. Personal Communication, June 2000.

[49] *Linear programming problems.* ftp://col.biz.uiowa.edu:pub/testprob/lp/gondzio.

[50] M. A. Iqbal, *Approximate algorithms for partitioning and assignment problems*, Tech. Rep. 86-40, NASA, June 1986.

[51] ——, *Efficient algorithms for partitioning problems*, in Proc. 1990 Int. Conf. on Parallel Processing (ICPP'90), 1990, pp. III–123–III–127.

[52] M. A. Iqbal, *Approximate algorithms for partitioning and assignment problems*, Int. J. Parallel Programming, 20 (1991).

[53] M. A. Iqbal and S. H. Bokhari, *Efficient algorithms for a class of partitioning problems*, Tech. Rep. 90-49, ICASE, July 1990.

[54] M. A. Iqbal and S. H. Bokhari, *Efficient algorithms for a class of partitioning problems*, IEEE Trans. Parallel and Distributed Systems, 6 (1995), pp. 170–175.

[55] M. A. Iqbal, J. H. Saltz, and S. H. Bokhari, *A comparative analysis of static and dynamic load balancing strategies*, in Proc. 1986 Int. Conf. on Parallel Processing, 1986, pp. 1040–1047.

[56] L. Kale, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, *Namd2: Greater scalability for parallel molecular dynamics*, J. Computational Physics, 15 (1999), pp. 283–312.

[57] L. V. Kalé, M. Bhandarkar, and R. Brunner, *Load balancing in parallel molecular dynamics*, in Fifth Int. Symp. on Solving Irregularly Structured Problems in Parallel, vol. 1457 of Lecture Notes in Computer Science, 1998, pp. 251–262.

[58] M.-Y. Kao, *Linear-processor NC algorithms for planar directed graphs I: Strongly connected components*, SIAM J. Comput., 22 (1993), pp. 431–459.

[59] R. M. Karp, *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, NY, 1972, pp. 85–103.

[60] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, Tech. Rep. TR 95-035, Department of Computer Science, University of Minnesota, 1995.

[61] B. W. Kernighan and S. Lin, *An efficient heuristic procedure for partitioning graphs*, The Bell System Technical J., 49 (1970), pp. 291–307.

[62] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing*, Benjamin/Cummings, Redwood City, California, 1994.

[63] H. Kutluca, T. M. Kurç, and C. Aykanat, *Image-space decomposition algorithms for sort-first parallel volume rendering of unstructured grids*, J. Supercomputing, 15 (2000), pp. 51–93.

[64] T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, Wiley, Chichester, U.K., 1990.

[65] F. Manne and T. Sørevik, *Optimal partitioning of sequences*, J. Algorithms, 19 (1995), pp. 235–249.

[66] W. McLendon III, B. Hendrickson, S. Plimpton, and L. Rauchwerger, *Identifying strongly connected components in parallel*, in Proc. 10th SIAM Conf. Parallel Processing for Scientific Computing, March 2001.

[67] S. Miguet and J. M. Pierson, *Heuristics for 1d rectilinear partitioning as a low cost and high quality answer to dynamic load balancing*, Lecture Notes in Computer Science, 1225 (1997), pp. 550–564.

[68] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, *A sorting classification of parallel rendering*, IEEE Computer Graphics and Applications, 14 (1994), pp. 23–32.

[69] C. Mueller, *The sort-first rendering architecture for high-performance graphics*, in Proc. 1995 Symp. Interactive 3D Graphics, 1995, pp. 75–84.

[70] D. M. Nicol, *Rectilinear partitioning of irregular data parallel computations*, J. Parallel and Disributed Computing, 23 (1994), pp. 119–134.

[71] D. M. Nicol and D. R. O'Hallaron, *Improved algorithms for mapping pipelined and parallel computations*, Tech. Rep. 88-2, ICASE, April 1988.

[72] ——, *Improved algorithms for mapping pipelined and parallel computation*, IEEE Trans. Computers, 40 (1991), pp. 295–306.

[73] B. Olstad and F. Manne, *Efficient partitioning of sequences*, IEEE Trans. Computers, 44 (1995), pp. 1322–1326.

[74] S. Pautz. Personal Communication, October 1999.

[75] J. Perttunen, *The traveling salesman problem: the significance of initial solutions in composite heuristics*, Tech. Rep. 0358-9080, Vaasan Korkeakoulun julkaisuja, 1989.

[76] J. R. Pilkington and S. B. Baden, *Dynamic partitioning of non-uniform structured workloads with spacefilling curves*, IEEE Trans. Parallel and Distributed Systems, 7 (1996), pp. 288–299.

[77] A. Pinar and C. Aykanat, *Sparse matrix decomposition with optimal load balancing*, in Proc. Int. Conf. High Performance Computing, 1997, pp. 224–229.

[78] ——, *Fast optimal load balancing algorithms for 1d partitioning*, J. Parallel and Distributed Computing, (2000).

[79] A. Pinar and M. Heath, *Improving performance of sparse matrix-vector multiplication*, in Proc. SC99, November 1999.

[80] A. Pinar and B. Hendrickson, *Interprocessor communication with memory constraints*, in Proc. ACM Symp. Parallel Algorithms and Applications, 2000, pp. 39–45.

[81] ——, *Communication support for adaptive computation*, in Proc. SIAM Conf. Parallel Processing for Scientific Computing, 2001.

[82] ——, *Partitioning for complex objectives*, in Proc. IPDPS 01, 2001.

[83] S. Plimpton, *Fast parallel algorithms for short-range molecular dynamics*, J. Computational Physics, 117 (1995), pp. 1–19.

[84] ——. Personal Communication, May 1999.

[85] S. Plimpton, S. Attaway, B. Hendrickson, J. Swegle, C. Vaughan, and D. Gardner, *Transient dynamics simulations: Parallel algorithms for contact detection and smoothed particle hydrodynamics*, J. Parallel Distrib. Comput., 50 (1998), pp. 104–122. Previous version appeared in Proc. Supercomputing'96.

[86] S. Plimpton, B. Hendrickson, S. Burns, and W. M. III, *Parallel algorithms for radiation transport on unstructured grids*, in Proc. SC'00, 2000.

[87] J. H. Reif, *Depth-first search is inherently sequential*, Inform. Process. Lett., 20 (1985), pp. 229–234.

[88] G. Reinelt, *The traveling salesman: computational solutions for TSP applications*, Springer-Verlag, 1994. Lecture Notes in Computer Science, Vol: 840.

[89] Y. Saad, *Practical use of polynomial preconditionings for the conjuagte gradient method*, SIAM J. Scientific and Statistical Computing, 6 (1985), pp. 865–881.

[90] J. Scott, October 1999. Personal communication.

[91] B. Smith, P. Bjørstad, and W. Gropp, *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, Cambridge University Press, Cambridge, UK, 1996.

[92] R. E. Tarjan, *Depth first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

[93] O. Temam and W. Jalby, *Characterizing the behavior of sparse algorithms on caches*, Proc. Supercomputing'92, (1992).

[94] S. Toledo, *Improving memory-system performance of sparse matrix-vector multiplication*, in Proc. the 8th SIAM Conf. Parallel Processing for Scientific Computing, 1997.

[95] R. S. Tuminaro, M. Heroux, S. A. Hutchinson, and J. N. Shadid, *Official aztec user's guide: Version 2.1*, Tech. Rep. SAND99-8801, Sandia National Laboratories, 1999.

[96] M. U. Ujaldon, E. L. Zapata, B. M. Chapman, and H. P. Zima, *Vienna-fortran/hpf extensions for sparse and irregular problems and their compilation*, IEEE Trans. Parallel and Distributed Systems, 8 (1997), pp. 1068–1083.

[97] M. U. Ujaldon, E. L. Zapata, S. D. Sharma, and J. Saltz, *Experimental evaluation of efficient sparse matrix distributions*, in Proc. ACM Int. Conf. of Supercomputing, 1996, pp. 78–86.

[98] ——, *Parallelization techniques for sparse matrix applications*, J. Parallel and Distributed Computing, 38 (1996), pp. 256–266.

Vita

Ali Pınar was born on November 9, 1971 in Kütahya, Turkey. He received his B.S. and M.S. degrees in Computer Engineering and Information Science from Bilkent University, Ankara, Turkey in 1994 and 1996, respectively. He joined the Department of Computer Science at University of Illinois at Urbana-Champaign in 1997 as a graduate student. He worked as a visiting researcher during summers of 1999 and 2000 at Sandia National Laboratories in Albuquerque, New Mexico, which also supported his Ph.D. studies. He completed his doctorate in 2001, and will join Lawrence Berkeley National Laboratories as a postdoctoral researcher. His research interests include graph theory, design and analysis of combinatorial algorithms, parallel computing, and scientific computing.