UNIVERSITY OF CALIFORNIA
Santa Barbara

Linear Algebraic Primitives for Parallel Computing on Large Graphs

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Aydın Buluç

Committee in Charge:

Professor John R. Gilbert, Chair

Professor Shivkumar Chandrasekaran

Professor Frederic T. Chong

Professor Ömer N. Eğecioğlu

March 2010

The Dissertation of
Aydın Buluç is approved:

Professor Shivkumar Chandrasekaran

Professor Frederic T. Chong

Professor Ömer N. Eğecioğlu

Professor John R. Gilbert, Committee Chairperson

March 2010

Linear Algebraic Primitives for Parallel Computing on Large Graphs

by

Aydın Buluç

To my friend of misery.

Acknowledgements

It is a great pleasure to thank everybody who made this thesis possible.

It is downright impossible to overstate how thankful I am to my advisor, John R. Gilbert. He has been a magnificant advisor and a great role model in all aspects: as a scientist, collaborator, teacher, and manager. He provided me a great work environment, perfectly balanced between my thesis work and the side projects that I wanted to pursue. This allowed me to enhance my breadth of knowledge of the general areas of parallel computing, sparse linear algebra and graph algorithms, without comprimising my depth of knowledge on my thesis work. Especially during my junior years as his Ph.D. student, John was as patient as a human being can ever be, pointing me to the right direction over and over again, secretly knowing that I would keep on asking the same type of questions for another three months until I would get mature enough to comprehend what he had been trying to tell me from the very beginning. I will always remember his amazing ability to motivate me without micromanaging.

I am grateful to all my committee members, for their time and valuable feedback. I am especially indebted to Ömer N. Eğecioğlu for being my mentor from the very first day I came to UCSB. He gave me more of his time than I probably deserved. I want to thank Alan Edelman for officially hosting me during my visit

to MIT, and giving me the opportunity to meet extremely smart people from all fields.

Special thanks to Charles E. Leiserson for his lead in reshaping our casual research discussion into a real collaboration. I am truly amazed by his outstanding scholarship and strong work ethics. The chapter on parallel sparse matrix-vector multiplication is a reprint of the material resulted from that collaboration with Jeremy T. Fineman and Matteo Frigo. I learned a lot from all three of them.

I am grateful to Erik Boman for having me at Sandia National Labs as a summer intern in 2008. I thank all the folks in our group at Sandia/Albuquerque for their company. Special thanks to Bruce Hendrickson for his support and encouragement. As a great scientist, he has a unique ability to stay righteous while speaking his opinions without reservations.

I would also like to thank my lab mates over the years: Viral Shah, Vikram Aggarwal, Imran Patel, Stefan Karpinski and Adam Lugowski. Many times, our conversations and meetings helped me think out of the box or sharpen my arguments.

When it comes to who gets the most credits about who I become, my family can simply not face competition. Thanks to "Annem ve Babam" who has supported me and my decisions without reservations. I know they would make God their enemy in this cause, if they had to.

I would like to thank all my friends who made my journey enjoyable. Special thanks to Murat Altay and Lütfiye Bulut for generously opening up their apartment to me while I was visiting MIT. Many thanks to my off-county friends, Simla Ceyhan and Osman Ertörer, for making North California my third home. I am especially grateful to Ahmet Bulut, Petko Bogdanov, Ali Irturk, Arda Atalı, Ceren Budak, Bahar Köymen, Emre Sargın, Başak Alper, Pegah Kamousi, Mike Wittie and Lara Deek for their company at UCSB. Vlasia Anagnostopoulou gets the most credit for reasons that will stay with me longer.

Curriculum Vitæ

Aydın Buluç

Education

2009	Master of Science in Computer Science, UC Santa Barbara.
2005	Bachelor of Science in Computer Science, Sabanci University

Experience

2007 – Present	Graduate Research Assistant, UC Santa Barbara.
2008	Summer Intern, Sandia National Labs
2006	Summer Intern, Citrix Online
2005 – 2007	Teaching Assistant, UC Santa Barbara.

Selected Publications

Aydın Buluç and John R. Gilbert. "On the Representation and Multiplication of Hypersparse Matrices," In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2008.

Aydın Buluç and John R. Gilbert. "Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication," In *Proceedings of the 37th International Conference on Parallel Processing (ICPP)*, September, 2008.

Aydın Buluç , Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication using

Compressed Sparse Blocks" In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, August 2009.

Aydın Buluç , Ceren Budak, and John R. Gilbert. "Solving path problems on the GPU", *Parallel Computing (2009)*, in press.

Abstract

Linear Algebraic Primitives for Parallel Computing on Large Graphs

Aydın Buluç

This dissertation presents a scalable high-performance software library to be used for graph analysis and data mining. Large combinatorial graphs appear in many applications of high-performance computing, including computational biology, informatics, analytics, web search, dynamical systems, and sparse matrix methods.

Graph computations are difficult to parallelize using traditional approaches due to their irregular nature and low operational intensity. Many graph computations, however, contain sufficient coarse grained parallelism for thousands of processors that can be uncovered by using the right primitives. We will describe the Parallel Combinatorial BLAS, which consists of a small but powerful set of linear algebra primitives specifically targeting graph and data mining applications.

Given a set of sparse matrix primitives, our approach to developing a library consists of three steps. We (1) design scalable parallel algorithms for the key primitives, analyze their performance, and implement them on distributed memory machines, (2) develop reusable software and evaluate its performance, and finally (3) perform pilot studies on emerging architectures.

The technical heart of this thesis is the development of a scalable sparse (generalized) matrix-matrix multiplication algorithm, which we use extensively as a primitive operation for many graph algorithms such as betweenness centrality, graph clustering, graph contraction, and subgraph extraction. We show that 2D algorithms scale better than 1D algorithms for sparse matrix-matrix multiplication. Our 2D algorithms perform well in theory and in practice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Background

Come, Come, Whoever You Are
Wonderer, worshipper, lover of leaving. It doesn't matter.
Ours is not a caravan of despair.
Come, even if you have broken your vow a thousand times
Come, yet again, come, come.

Mevlana Celaleddin Rumi

This thesis provides a scalable high-performance software library, the Combinatorial BLAS, to be used for graph analysis and data mining. It targets parallel computers and its main toolkit is composed of sparse linear algebraic operations.

In this chapter, I will try to give the motivation behind this work. Most of these motivations are accompanied by historical background and references to recent trends. The first section is a personal interpretation of the parallel computing world as of early 2010. Section 1.2 reviews the state of the art of parallel graph computations in practice. The following two sections, Sections 1.3 and 1.4, summarize the justifications of the two main ideas behind this thesis:

the use of primitives and of sparse matrices. The final section summarizes the contributions and provides an outline of this thesis.

1.1 The Landscape of Parallel Computing

It is no news that the economy is driving the past, present, and future of computer systems. It was the economy that drove the "killer micro" [45] and stalled innovative supercomputer design in the early 1990s. It is the economy that is driving GPUs to get faster and forcing a unification of GPU/CPU architectures today. It will be the economy that will drive energy efficient computing and massive parallelism. This is partially due to a number of fundamental physical limitations on sequential processing, such as the speed of light and the dissipation of heat [156].

Although the literature contains several taxonomies of parallelism [85, 120, 176], one can talk about two fundamental types of parallelism available for exploitation in software: data parallelism and task parallelism. The former executes the same instruction on a relatively large data set. For example, elementwise addition of two vectors has lots of data parallelism. Task parallelism, on the other hand, is achieved by decomposing the application into multiple independent tasks that can be executed as separate procedures. A multi-threaded web server pro-

vides a good example for task parallelism, where multiple requests of different kinds are handled in parallel. In reality, most applications use a combination of data and task parallelism, and therefore, fall somewhere in the middle of the spectrum. Although it is possible to rewrite some applications that were previously written in the data parallel fashion in a task parallel fashion, and vice versa, this is not always possible.

In general, one can speak about a relationship between the current parallel architectures and types of available parallelism. For example, massively multithreaded architectures [91, 129] are better than others when dealing with large amounts of task parallelism. On the other side, GPUs [1, 2] excel in data parallel computations. However, as most computations cannot be hard-classified as having solely task parallelism or solely data parallelism, an ultimate direct mapping of applications to architectures is unlikely to emerge.

From the late 1990s to the late 2000s, the supercomputing market was mostly dominated by clusters made from commercial off-the-shelf processors. After this decade of relative stability in parallel computing architectures, we are now experiencing disruptions and divergences. Different application have different resource requirements, leading to diversity and heterogeneity in parallel computing architectures. On the other hand, the economic of scale dictate that a handful general-purpose architectures that can be manufactured with low cost will domi-

nate the HPC market. While building custom architectures that perfectly match the underlying problem might be tempting for the HPC community, commodity architectures have the advantage of achieving a much lower cost per unit. Special purpose supercomputers, such as Anton [182], which is used to simulate molecular dynamics of biological systems, will still find applications where the reward exceeds the cost. For broader range of applicability, however, supercomputers that feature a balanced mixture of commodity and custom parts are likely to prevail.

Examples of currently available high-performance computing systems include the following:

1. Distributed memory multiprocessors (such as the Cray XT4 and XT5) and beowulf clusters that are primarily programmed using MPI. This class of machines also include the RISC-based distributed-memory multi-processors such as the IBM BlueGene [193],

2. ccNUMA and multicore architectures that are programmed either explicitly through pthreads or through concurrency platforms like Cilk/Cilk++ [138], OpenMP and Intel Building Blocks [167],

3. Massively multithreaded shared memory machines such as the Cray XMT [91] and the Sun Niagara [129], and

4. GPU or IBM Cell accelerated clusters. The largest scale example of the latter (as of March 2010) is the Roadrunner system deployed at Los Alamos National Laboratory [22].

Regardless of which architure(s) will prevail in the end, the economic trends favor more parallelism in computing because building a parallel computer using large number of simple processors has proved to be more efficient, both financially and in terms of power, than using a small number of complex processors [181]. The software world has to deal with this revolutionary change in computing. It is safe to say that the software industry has been caught off-guard by this challenge. Most programmers are not fundamentally trained to think "in parallel". Many tools, such as debuggers and profilers, that are taken for granted when writing sequential programs, were (and still are) lacking for parallel software development. In the last few years, there have been improvements towards making parallel programming easier, including parallel debuggers [9, 199], concurrency platforms [138, 167], and various domain specific libraries. Part of this thesis strives to be a significant addition to the latter group of parallel libraries.

One of the most promising approaches for tackling the software challenge in parallel computing is the top-down, application-driven, approach where common algorithmic kernels in various important application domains are identified. In the inspiring Berkeley Report [13], these kernels are called "dwarfs"(or "motifs").

This thesis is mostly concerned about the close interaction between two of those dwarfs: graph traversal and sparse linear algebra.

1.2 Parallel Graph Computations

This section surveys working implementations of graph computations, rather than describing research on parallel graph algorithms. We also focus on frameworks and libraries instead of parallelization of stand-alone applications. The current landscape of software for graph computations is summarized in Table 1.1.

Table 1.1: High-performance libraries and toolkits for parallel graph analysis

Library/Toolkit	Parallelism	Abstraction	Offering	Scalability
PBGL [108]	Distributed	Visitor	Algorithms	Limited
GAPDT [105]	Distributed	Sparse Matrix	Both	Limited
MTGL [30]	Shared	Visitor	Algorithms	Unknown
SNAP [145]	Shared	Various	Both	Good
Combinatorial BLAS	Distributed	Sparse Matrix	Kernels	Good

The Parallel Boost Graph Library (PBGL) [108] is a parallel library for distributed memory computing on graphs. It is a significant step towards facilitating rapid development of high performance applications that use distributed graphs as their main data structure. Like sequential Boost Graph Library [184], it has a dual focus on efficiency and flexibility. It heavily relies on generic program-

ming through C++ templates. To the user, it offers complete algorithms instead of tools to implement the algorithms. Therefore, its applicability is limited for users who need to experiment with new algorithms or instrument the existing ones. Lumsdaine et al.[143] observed poor scaling of PBGL for some large graph problems.

We believe that the scalability of PBGL is limited due to two main reasons. The graph is distributed by vertices instead of edges, which corresponds to a one-dimensional partitioning in the sparse matrix world. In Chapter 3, we show that this approach is unscalable. We also believe that the *visitor* concept is too low-level for providing scalability in distributed memory because it makes the computation data driven and obstructs opportunies for optimization.

The Graph Algorithms and Pattern Discovery Toolbox (GAPDT) [105] provides several tools to manipulate large graphs interactively. It is designed to run sequentially on MATLAB [104] or in parallel on STAR-P [179], a parallel dialect of MATLAB. Although its focus is on algorithms, the underlying sparse matrix infrastructures of MATLAB and STAR-P also exposes necessary kernels (linear algebraic building blocks, in this case). It targets the same platform as PBGL, namely distributed-memory machines. Differently from PBGL, it uses operations on distributed sparse matrices for parallelism. It provides an interactive environment instead of compiled code, which makes it unique among all the other

approaches we survey here. Similar to PBGL, GAPDT's main weakness is its limited scalability due to the one-dimensional distribution of its sparse matrices.

A number of approaches have been tried in order to mitigate the poor scalability. One architectural approach is to tolerate latency by using massive multithreading. This idea, known as ***interleaved multithreading*** [135, 201], relies on CPUs that can switch thread contexts on every cycle. Currently, a limited number of architectures are capable of performing true hardware multithreading. Cray XMT(formerly MTA) [91], IBM Cyclops64 [10], and Sun Niagara [129] based servers are among the important examples. The first two exclusively target the niche supercomputing market, therefore limiting their large scale deployment prospects. In contrast, Sun Niagara processors are used in Sun's business servers that run commercial multithreaded applications. With its impressive performace per watt for high throughput applications [134], Niagara may make massive hardware multithreading affordable and widespread as long as it maintains its status as a competitive server platform for commercial applications.

All three massively multithreaded architectures, namely XMT, Cyclops64, and Niagara, tolerate the data access latencies by keeping lots of threads on the fly. Cyclops64 is slightly different than others in the way it manages thread contexts. In Cyclops64, each thread context has its own execution hardware , whereas in MTA/XMT the whole execution pipeline is shared among threads. Niagara is

somewhere in between in the sense that a group of threads (composed of four threads) shares a processing pipeline but each group has a different pipeline from other groups. Niagara differs from the other two also by having large on-die caches, which are managed by a simple cache coherence protocol. Interleaved multithreading, although very promising, has at least one more obstacle in addition to finding a big enough market. The large number of threads that are kept on the fly puts too much pressure on the bandwidth requirements of the interconnect. In the case of MTA-2, this was solved by using a modified Cayley graph whose bisection bandwidth scales linearly with the number of processors. The custom interconnect later proved to be too expensive, and for the next generation XMT, Cray decided to use a 3D torus interconnect instead. This move made the XMT system more economically accessible, but it also sacrificed scalability for applications with high bandwidth requirements [144].

The *MultiThreaded Graph Library* (MTGL) [30] was originally designed to facilitate the development of graph applications on massively multithreaded machines of Cray, namely MTA-2 and XMT. Later, it was extended to run on the mainstream shared-memory and multicore architectures as well [23]. The MTGL is a significant step towards an extendible and generic parallel graph library. It will certainly be interesting to quantify the abstraction penalty paid due to its

generality. As of now, only preliminary performance results are published for MTGL.

The Small-world Network Analysis and Partitioning (SNAP) [145] framework contains algorithms and kernels for exploring large-scale graphs. It is a collection of different algorithms and building blocks that are optimized for small-world networks. It combines shared-memory thread level parallelism with state-of-the-art algorithm engineering for high performance. The graph data can be represented in a variety of different formats depending on the characteristics of the algorithm that operates on it. Its performance and scalability is high for the reported algorithms, but a head-to-head performance comparison with PBGL and GAPDT is not available.

Both MTGL and SNAP are great toolboxes for graph computations on multithreaded architectures. For future extensions, MTGL relies on the visitor concept it inherits from the PBGL, while SNAP relies on its own kernel implementations. Both software architectures are maintainable as long as the target architectures remain the same.

Algorithms on massive graphs with billions of vertices and edges require hundreds of gigabytes of memory. For a special purpose supercomputer such as XMT, memory might not be a problem; but commodity shared-memory architectures have limited memory. Thus, MTGL or SNAP will likely to find limited use in com-

modity architectures without either distributed memory or out-of-core support. Experimental studies show that an out-of-core approach [8] is two orders of magnitude slower than an MTA-2 implementation for parallel breadth-first search [20]. Given that many graph algorithms, such as clustering and betweenness centrality, are computationally intensive, out-of-core approaches are infeasible. Therefore, distributed memory support for running graph applications of general purpose computers is essential. Neither MTGL or SNAP seem easily extendible to distributed memory.

1.3 The Case for Primitives

Large scale software development is a formidable task that requires an enormous amount of human expertise, especially when it comes to writing software for parallel computers. Writing every application from scratch is an unscalable approach given the complexity of the computations and the diversity of the computing environments involved. Raising the level of abstraction of parallel computing by identifying the algorithmic commonalities across applications is becoming a widely accepted path to solution for the parallel software challenge [13, 44]. Primitives both allow algorithm designers to think on a higher level of abstraction, and help to avoid duplication of implementation efforts.

Achieving good performance on modern architectures requires substantial programmer effort and expertise. Primitives save programmers from implementing the common low-level operations. This often leads to better understanding of the mechanics of the computation in hand because carefully designed primitives can usually handle seemingly different but algorithmically similar operations. Productivity of the application-level programmer is also dramatically increased as he or she can now concentrate on the higher-level structure of the algorithm without worrying about the low level details. Finally, well-implemented primitives often outperform hand-coded versions. In fact, after a package of primitives proves to have widespread use, it is usually developed and tuned by the manifacturers for their own architectures.

1.3.1 A Short Survey of Primitives

Primitives have been successfully used in the past to enable many computing applications. *The Basic Linear Algebra Subroutines* (BLAS) for numerical linear algebra are probably the canonical example [136] of a successful primitives package. The BLAS became widely popular following the success of LAPACK [12]. The BLAS was originally designed for increasing modularity of scientific software, and LINPACK used it to increase the code sharing among projects [78]. LINPACK's use of the BLAS encouraged experts (preferably the vendors themselves)

implement its vector operations for optimal performance. Other than the efficiency benefits, it offered portability by providing a common interface for these subroutines. It also indirectly encouraged structured programming.

Later, as computers started to have deeper memory hierarchies and advances in microprocessors made memory access more costly than performing floating-point operations, BLAS Level 2 [81] and Level 3 [80] specifications were developed, in late 1980s. They emphasize blocked linear algebra to increase the ratio of floating-point operations to slow memory accesses. Although BLAS 2 and 3 had different tactics for achieving high performance, both followed the same strategy of packaging the commonly used operations and having experts provide the best implementations through performing algorithmic transformations and machine-specific optimizations. Most of the reasons for developing the BLAS package about four decades ago are valid for the general case for primitives today.

Google's MapReduce programming model [75], which is used to process massive data on clusters, is also of similar spirit. The programming model allows the user to customize two primitives: `map` and `reduce`. Although two different customized `map` operations are likely to perform different computations semantically, they perform similar tasks algorithmically as they both apply a (user-defined) function to every element of the input set. A similar reasoning applies for the `reduce` operation.

Guy Blelloch advocates the use of prefix sums (scan primitives) for implementing a wide range of classical algorithms in parallel [32]. The data-parallel language NESL is primarily based on these scan primitives [33]. Scan primitives have also been ported to manycore processors [117] and found widespread use.

1.3.2 Graph Primitives

In contrast to numerical computing, a scalable software stack that eases the application programmer's job does not exist for computations on graphs. Some of the primitives we surveyed can be used to implement a number of graph algorithms. Scan primitives are used for solving the maximum flow, minimum spanning tree, maximal independent set, and (bi)connected components problems efficiently. On the other hand, it is possible to implement some clustering and connected components algorithms using the MapReduce model, but the approaches are quite unintuitive and the performance is unknown [64]. Our thesis fills a crucial gap by providing primitives that can be used for traversing graphs. By doing so, the Combinatorial BLAS can be used to perform tightly-coupled, such as shortest paths based and diffusion based, computations on graphs.

We consider the shortest paths problem on dense graphs in Chapter 7. By using an unorthodox blocked recursive elimination strategy together with a highly optimized matrix-matrix multiplication, we achieve up to 480 times speedup over

a standard code running on a single CPU. The conclusion of that pilot study is that carefully chosen and optimized primitives, such as the ones found in the combinatorial BLAS, are the key to achieve high performance.

1.4 The Case for Sparse Matrices

The connection between graphs and sparse matrices was first exploited for computation five decades ago in the context of Gaussian elimination [160]. Graph algorithms have always been a key component in sparse matrix computations [65, 101]. In this thesis, we turn this relationship around and use sparse matrix methods to efficiently implement graph algorithms [103, 105]. Sparse matrices seemlessly raise the level of abstraction in graph computations by replacing the irregular data access patterns with more structured matrix operations.

The sparse matrix infrastructures of the MATLAB, STAR-P, Octave and R programming languages [94] allow for work-efficient implementations of graph algorithms. STAR-P is a parallel dialect of MATLAB that includes distributed sparse matrices, which are distributed across processors by blocks of rows. The efficiency of graph operations results from the efficiency of sparse matrix operations. For example, both MATLAB and STAR-P follow the design principle that the storage of a sparse matrix should be proportional to the number of nonzero elements

and the running time for a sparse matrix algorithm should be proportional to the number of floating point operations required to obtain the result. The first principle ensures storage efficiency for graphs while the second principle ensures work efficiency.

Graph traversals, such as breadth-first search and depth-first search, are the natural tools for designing graph algorithms. Traversal-based algorithms visit vertices following the connections (edges) between them. When translated into actual implementation, this traditional way of expressing graph algorithms poses performance problems in practice. Here, we summarize these challenges, which are examined in detail by Lumsdaine et al. [143], and provide a sparse matrix perspective for tackling these challenges.

Traditional graph computations suffer from poor locality of reference due to their irregular access patterns. Graphs computations in the language of linear algebra, on the other hand, involve operations on matrix blocks. Matrix operations give opportunities for the implementer to restructure the computation in a way that would exploit the deep memory hierarchies of modern processors.

Implementations for parallel computers also suffer from unpredictable communication patterns because they are mostly data driven. Consider an implementation of parallel breadth-first search in which the vertices are assigned to processors. The owner processor finds the adjacency of each vertex in the current

frontier, in order to form the next frontier. The adjacent vertices are likely to be owned by different processors, resulting in communication. Since the next frontier is not known in advance, the schedule and timing of this communication is also not known in advance. On the other hand, sparse linear algebra operations have fixed communication schedules that are built into the algorithm. Although sparse matrices are no panacea for irregular data dependencies, the operations on them can be restructured to provide more opportunities for optimizing the communication schedule such as overlapping communication with computation and pipelining.

Both in serial and parallel settings, the computation time is dominated by the latency of fetching the data (from slow memory in serial case and from remote processor's memory in parallel case) to local registers, due to fine grained data accesses of graph computations. Massively multithreaded architectures tolerate this latency by keeping lots of outstanding memory requests on the fly. Sparse matrix operations have coarse-grained parallelism, which is much less affected by latency costs.

1.5 Definitions and Conventions

Let $\mathbf{A} \in \mathbb{S}^{m \times n}$ be a sparse rectangular matrix of elements from an arbitrary semiring $\mathbb{S}$. We use $nnz(\mathbf{A})$ to denote the number of nonzero elements in $\mathbf{A}$. When the matrix is clear from context, we drop the parenthesis and simply use nnz. For sparse matrix indexing, we use the convenient MATLAB® colon notation, where $\mathbf{A}(:, i)$ denotes the ith column, $\mathbf{A}(i, :)$ denotes the ith row, and $\mathbf{A}(i, j)$ denotes the element at the (i, j)th position of matrix $\mathbf{A}$. For one-dimensional arrays, $\mathbf{a}(i)$ denotes the ith component of the array. Sometimes, we abbreviate and use $nnz(j)$ to denote the number of nonzeros elements in the jth column of the matrix in context. Array indices are 1-based throughout this thesis, except where stated otherwise. We use flops($\mathbf{A}\, op\, \mathbf{B}$), pronounced "flops", to denote the number of nonzero arithmetic operations required by the operation $\mathbf{A}\, op\, \mathbf{B}$. Again, when the operation and the operands are clear from context, we simply use flops. To reduce notational overhead, we take each operation's complexity to be at least one, i.e. we say $O(\cdot)$ instead of $O(\max(\cdot, 1))$.

For testing and analysis, we have extensively used three main models: the R-MAT model, the Erdős-Rényi random graph model, and the regular 3D grid model. We have frequently used other matrices for testing, which we will explain in detail in their corresponding chapters.

1.5.1 Synthetic R-MAT Graphs

The R-MAT matrices represent the adjacency structure of scale-free graphs, generated using repeated Knonecker products [56, 140]. R-MAT models the behavior of several real-world graphs such as the WWW graph, small world graphs, and citation graphs. We have used an implementation based on Kepner's vectorized code [16], which generates directed graphs. Unless otherwise stated, R-MAT matrices used in our experiments have an average of degree of 8, meaning that there will be approximately $8\,n$ nonzeros in the adjacency matrix. The parameters for the generator matrix are $a = 0.6$, and $b = c = d = 0.13$. As the generator matrix is 2-by-2, R-MAT matrices have dimensions that are powers of two. An R-MAT graph with ***scale*** l has $n = 2^l$ vertices.

1.5.2 Erdős-Rényi Random Graphs

An Erdős-Rényi random graph $G(n, p)$ has n vertices, each of the possible n^2 edges in the graph exists with fixed probability p, independent of the other edges [88]. In other words, each edge has an equally likely chance to exist. A matrix modeling the Erdős-Rényi graph $G(n, p)$ is expected to have with n^2/p nonzeros, independently and identically distributed (i.i.d.) across the matrix. Erdős-Rényi random graphs can be generated using the `sprand` function of MATLAB.

1.5.3 Regular 3D Grids

As the representative of regular grid graphs, we have used matrices arising from graphs representing the 3D 7-point finite difference mesh (`grid3d`). These input matrices, which are generated using the MATLAB Mesh Partitioning and Graph Separator Toolbox [103], are highly structured block diagonal matrices.

1.6 Contributions

This thesis presents the combinatorial BLAS, a parallel software library that consists of a set of sparse matrix primitives. The combinatorial BLAS enables rapid parallel implementation of graph algorithms through composition of primitives. The development of this work has four main contributions.

The first contribution is the analysis of important combinatorial algorithms to identify the linear-algebraic primitives that serve as the workhorses of these algorithms. Early work on identifying primitives was explored in the relevant chapters [92, 168] of an upcoming book on Graph Algorithms in the Language of Linear Algebra [127]. In short, the majority of traditional and modern graph algorithms can be efficiently written in the language of linear algebra, except for algorithms whose complexity depends on a priority queue data structure. Although we will not be duplicating those efforts, non-exclusive list of graph algorithms that

are represented in the language of matrices, along with detailed pseudocodes, can be found in various chapters of this thesis.

The second contribution is the design, analysis, and implementation of key sparse matrix primitives. Here we take both theory and practice into account by providing practically useful algorithms with rigorous theoretical analysis. Chapter 2 provides details on implementing key primitives using sparse matrices. It surveys a variety of sequential sparse matrix storage formats, pinpointing their advantages and disadvantages for the primitives at hand. Chapter 3 presents novel algorithms for the least studied and the most important primitive in the combinatorial BLAS: Generalized sparse matrix-matrix multiplication (SpGEMM). The work in this chapter mainly targets scalability on distributed memory architectures.

The third contribution is software development and performance evaluation. The implementation details and performance enhancing optimizations for the SpGEMM primitive is separately analyzed in the last two sections of Chapter 3. These sections also report on the performance of the SpGEMM primitive on various test matrices. Chapter 4 explains the interface design for the combinatorial BLAS in detail. The whole combinatorial BLAS library is evaluated using two important graph algorithms, in terms of both performance and ease-of-use, in Chapter 5. Chapter 6 (like Chapter 3) provide another detailed example of op-

timizing primitives, this time sparse matrix-vector and sparse matrix-transpose-vector operations on multicore architectures.

The first three contributions are incidentally typical components of a research project in combinatorial scientific computing [119], except that the roles of problems and solutions are swapped. In other words, we are solving a combinatorial problem using matrix methods instead of solving a matrix problem using combinatorial methods.

The last contribution is our pilot studies on emerging architectures. In the context of this thesis, the contributions in Chapters 6 and 7, apart from being important in themselves, should also be seen as seed projects evaluating the feasibility of extending our work to a complete combinatorial BLAS implementation on GPU's and shared-memory systems.

Chapter 2

Implementing Sparse Matrices for Graph Algorithms

Abstract

We review and evaluate storage formats for sparse matrices in the light of the key primitives that are useful for implementing graph algorithms on them. We present complexity results of these primitives on different sparse storage formats both in the RAM model and in the I/O model. RAM complexity results were known except for the analysis of sparse matrix indexing. On the other hand, most of the I/O complexity results presented are new. The paper focuses on different variations of the triples (coordinates) format and the widely used compressed sparse formats (CSR/CSC). For most primitives, we provide detailed pseudocodes for implementing them on triples and CSR/CSC.

2.1 Introduction

The choice of data structure is one of the most important steps in algorithm design and implementation. Sparse matrix algorithms are no exception. The representation of a sparse matrix not only determines the efficiency of the algorithm that operates on it, but also influences the algorithm design process.

Given this bidirectional relationship, this chapter reviews and evaluates existing sparse matrix data structures with key primitives in mind. In the case of array-based graph algorithms, these primitives are sparse matrix-vector multiplication (SpMV), sparse matrix-matrix multiplication (SpGEMM), sparse matrix indexing/assignment (SpRef/SpAsgn), and sparse matrix addition (SpAdd). The administrative overheads of different sparse matrix data structures, both in terms of storage and processing, are also important and are exposed throughout the chapter.

One of the traditional ways to analyze the computational complexity of a sparse matrix operation is by counting the number of floating point operations performed. This is similar to analyzing algorithms according to their RAM complexities [7]. As memory hierarchies became dominant in computer architectures, the I/O complexity (also called the cache complexity) of a given algorithm became as important as its RAM complexity. Aggarwal and Vitter [6] roughly defines the I/O complexity of an algorithm as the number of block memory transfers it makes between the fast and slow memories. Cache performance is especially important for sparse matrix computations due to their irregular nature and low ratio of flops to memory access. Another approach to hiding the memory-processor speed gap is to use massively multithreaded architectures such as Cray's XMT [91]. However, these architectures have limited availability and high costs at present.

In many popular I/O models, only two levels of memory are considered for simplicity: a fast memory, and a slow memory. The fast memory is called cache and the slow memory is called disk, but the analysis is valid at different levels of memory hierarchy with appropriate parameter values. Both levels of memories are partitioned into blocks of size L, usually called the cache line size. The size of the fast memory is denoted by Z. If data needed by the CPU is not found in the fast memory, a *cache miss* occurs, and the memory block containing the needed data is fetched from the slow memory. One exception to these two-level I/O models is the uniform memory hierarchy of Alpern et al. [11], which views the computer's memory as a hiearchy of increasingly large memory modules. Figure 2.1 shows a simple memory hierarchy with some typical latency values as of 2006. Meyer et al. provide a contemporary treatment of algorithmic implications of memory hiearchies [150].

We present the computational complexity of algorithms in the RAM model as well as the I/O model. However, instead of trying to come up with the most I/O efficient implementations, we analyze the I/O complexities of the most widely used implementations, which are usually motivated by the RAM model. There are two reasons for this approach. First, I/O optimality is still an open problem for some of the key primitives presented in this chapter. Second, I/O efficient

implementations of some key primitives turn out to be suboptimal in the RAM model with respect to the amount of work they do.

We use $scan(\mathbf{A}) = \lceil nnz(\mathbf{A})/L \rceil$ as an abbreviation for the I/O complexity of examining all the nonzeros of matrix $\mathbf{A}$ in the order that they are stored.

Now, we state two crucial assumptions that are used throughout this chapter.

Assumption 1. *A sparse matrix with dimensions $m \times n$ has $nnz \geq m, n$. More formally, $nnz = \Omega(n, m)$*

Assumption 1 simplifies the asymptotic analysis of the algorithms presented in this chapter. It implies that when both the order of the matrix and its number of nonzeros are included as terms in the asymptotic complexity, only *nnz* is pronounced. While this assumption is common in numerical linear algebra (it is required for full rank), in some parallel graph computation it may not hold. In this chapter, however, we use this assumption in our analysis. In Chapter 3 (Section 3.2.1), we present an SpGEMM algorithm specifically designed for hypersparse matrices, with $nnz < n, m$.

Assumption 2. *The fast memory is not big enough to hold data structures of $O(n)$ size, where n is the matrix dimension.*

In most settings, especially for sparse matrices representing graphs, $nnz = \Theta(n)$, which means that $O(n)$ data structures are asymptotically in the same order

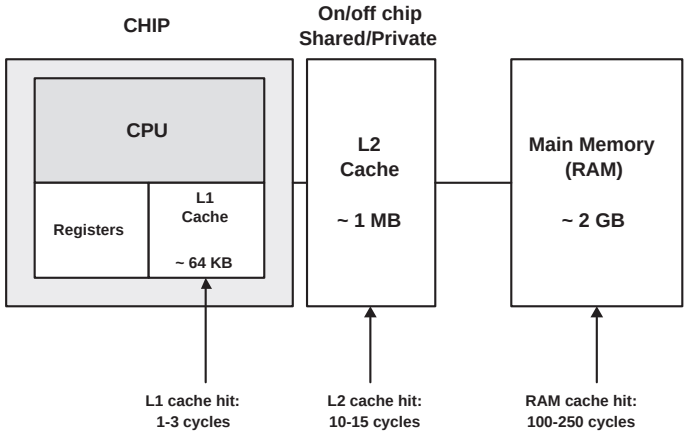


Figure 2.1: A typical memory hierarchy (approximate values as of 2006, partially adapted from Hennessy and Patterson [120], assuming a 2 Ghz processor)

as the whole problem. Assumption 2 is also justified when the fast memory under consideration is either the L1 or L2 cache. Out-of-order CPUs can generally hide memory latencies from L1 cache misses, but not L2 cache misses [120]. Therefore, it is more reasonable to treat the L2 cache as the fast memory and RAM (main memory) as the slow memory. The largest sparse matrix that fills the whole machine RAM (assuming the triples representation that occupies 16 bytes per nonzero, and a modern system with 1 MB L2 cache and 2 GB of RAM) has $2^{31}/16 = 2^{27}$ nonzeros. Such a square sparse matrix, with an average of 8 nonzeros per column, has dimensions $2^{24} \times 2^{24}$. A single dense vector of double-precision floating point numbers with 2^{24} elements require 128 MB of memory, which is clearly much larger than the size of the L2 cache.

The rest of this chapter is organized as follows. Section 2.2 describes the key sparse matrix primitives. Section 2.3 reviews the triples/ coordinates representation, which is natural and easy to understand. The triples representation generalizes to higher dimensions [15]. Its resemblence to database tables will help us expose some interesting connections between databases and sparse matrices. Section 2.4 reviews the most commonly used compressed storage formats for general sparse matrices, namely compressed sparse row (CSR) and compressed sparse column (CSC). Section 2.5 discusses some other sparse matrix representations proposed in the literature, followed by a conclusion. We introduce new sparse matrix data structures in Chapters 3 and 6. These data structures, DCSC and CSB, are both motivated by parallelism.

We explain sparse matrix data structures progressively, starting from the least structured and most simple format (unordered triples) and ending with the most structured formats (CSR and CSC). This way, we provide motivation on why experts prefer to use CSR/CSC formats by comparing and contrasting them with simpler formats. For example, CSR, a dense collection of sparse row arrays, can also be viewed as an extension of the triples format enhanced with row indexing capabilities. Furthermore, many ideas and intermediate data structures that are used to implement key primitives on triples are also widely used with implementations on CSR/CSC formats.

2.2 Key Primitives

Most of the sparse matrix operations have been motivated by numerical linear algebra. Some of them are also useful for graph algorithms:

1. Sparse matrix indexing and assignment (SpRef/SpAsgn)
2. Sparse matrix-dense vector multiplication (SpMV)
3. Sparse matrix addition and other pointwise operations (SpAdd)
4. Sparse matrix-sparse matrix multiplication (SpGEMM)

SpRef is the operation of storing a submatrix of a sparse matrix in another sparse matrix ($\mathbf{B} \leftarrow \mathbf{A}(\mathbf{p}, \mathbf{q})$), and SpAsgn is the operation of assigning a sparse matrix to a submatrix of another sparse matrix ($\mathbf{B}(\mathbf{p}, \mathbf{q}) \leftarrow \mathbf{A}$). It is worth noting that SpAsgn is the only key primitive that mutates its sparse matrix operand in the general case[1]. Sparse matrix indexing can be quite powerful and complex if we allow $\mathbf{p}$ and $\mathbf{q}$ to be arbitrary vectors of indices. Therefore, this chapter limits itself to row-wise ($\mathbf{A}(i, :)$), column-wise ($\mathbf{A}(:, j)$), and element-wise ($\mathbf{A}(i, j)$) indexing, as they find more widespread use in graph algorithms. SpAsgn also requires the matrix dimensions to match. For example, if $\mathbf{B}(:, i) = \mathbf{A}$ where $\mathbf{B} \in \mathbb{S}^{m \times n}$, then $\mathbf{A} \in \mathbb{S}^{1 \times n}$.

[1] While $\mathbf{A} = \mathbf{A} \oplus \mathbf{B}$ or $\mathbf{A} = \mathbf{AB}$ may also be considered as mutator operations, these are just special cases when the output is the same as one of the inputs

$\mathbf{C} : \mathbb{R}^{S(m \times n)} = \text{InnerProduct-SpGEMM}(\mathbf{A} : \mathbb{R}^{S(m \times k)}, \mathbf{B} : \mathbb{R}^{S(k \times n)})$

```
1  for i ← 1 to m
2        do for j ← 1 to n
3              do C(i, j) ← A(i, :) · B(:, j)
```

Figure 2.2: Inner product formulation of matrix multiplication

$\mathbf{C} : \mathbb{R}^{S(m \times n)} = \text{OuterProduct-SpGEMM}(\mathbf{A} : \mathbb{R}^{S(m \times k)}, \mathbf{B} : \mathbb{R}^{S(k \times n)})$

```
1  C ← 0
2  for l ← 1 to k
3        do C ← C + A(:, l) · B(l, :)
```

Figure 2.3: Outer-product formulation of matrix multiplication

SpMV is the most widely used sparse matrix kernel since it is the workhorse of iterative linear equation solvers and eigenvalue computations. A sparse matrix can be multiplied by a dense vector either on the right ($\mathbf{y} \leftarrow \mathbf{Ax}$) or on the left ($\mathbf{y}^\mathsf{T} \leftarrow \mathbf{x}^\mathsf{T}\mathbf{A}$). This chapter concentrates on the multiplication on the right. It is generally straightforward to reformulate algorithms that use multiplication on the left so that they use multiplication on the right. Some representative graph computations that use SpMV are page ranking (an eigenvalue computation), breadth-first search, the Bellman-Ford shortest paths algorithm, and Prim's minimum spanning tree algorithm.

SpAdd, $\mathbf{C} \leftarrow \mathbf{A} \oplus \mathbf{B}$, computes the sum of two sparse matrices of dimensions $m \times n$. SpAdd is an abstraction that is not limited to a particular summation

operator. In general, any pointwise binary scalar operation between two sparse matrices falls into this primitive. Examples include the MIN operator that returns the minimum of its operands, logical AND, logical OR, ordinary addition, and subtraction.

SpGEMM computes the sparse product $\mathbf{C} \leftarrow \mathbf{AB}$, where the input matrices $\mathbf{A} \in \mathbb{S}^{m \times k}$ and $\mathbf{B} \in \mathbb{S}^{k \times n}$ are both sparse. It is a common operation for operating on large graphs, used in graph contraction, peer pressure clustering, recursive formulations of all-pairs-shortest-path algorithms, and breadth-first search from multiple source vertices. Chapter 3 presents novel ideas for computing SpGEMM.

The computation for matrix multiplication can be organized in several ways. One common formulation uses inner products, shown in Figure 2.2. Every element of the product $\mathbf{C}(i, j)$ is computed as the dot product of a row i of $\mathbf{A}$ and a column j of $\mathbf{B}$. Another formulation of matrix multiplication uses outer products (Figure 2.3). The product is computed as a sum of k rank-one matrices. Each rank-one matrix is the outer product of a column of $\mathbf{A}$ with the corresponding row of $\mathbf{B}$.

SpGEMM can also be organized so that $\mathbf{A}$ and $\mathbf{B}$ are accessed by row or columns, computing one row/column of the product $\mathbf{C}$ at a time. Figure 2.5 shows the column-wise formulation where column j of $\mathbf{C}$ is computed as a linear combination of the columns of $\mathbf{A}$ as specified by the nonzeros in column j of $\mathbf{B}$.

$\mathbf{C} : \mathbb{R}^{S(m \times n)} = \text{RowWise-SpGEMM}(\mathbf{A} : \mathbb{R}^{S(m \times k)}, \mathbf{B} : \mathbb{R}^{S(k \times n)})$

```
1   for i ← 1 to m
2         do for l where A(i, l) ≠ 0
3               do C(i, :) ← C(i, :) + A(i, l) · B(l, :)
```

Figure 2.4: Row-wise formulation of matrix multiplication

$\mathbf{C} : \mathbb{R}^{S(m \times n)} = \text{ColumnWise-SpGEMM}(\mathbf{A} : \mathbb{R}^{S(m \times k)}, \mathbf{B} : \mathbb{R}^{S(k \times n)})$

```
1   for j ← 1 to n
2         do for l where B(l, j) ≠ 0
3               do C(:, j) ← C(:, j) + A(:, l) · B(l, j)
```

Figure 2.5: Column-wise formulation of matrix multiplication

Figure 2.6 gives a diagram. Similarly, for the row-wise formulation, each row i of $\mathbf{C}$ is computed as a linear combination of the rows of $\mathbf{B}$ specified by nonzeros in row i of $\mathbf{A}$ as shown in Figure 2.4.

2.3 Triples

The simplest way to represent a sparse matrix is the *triples* (or *coordinates*) format. For each $\mathbf{A}(i, j) \neq 0$, the triple $(i, j, \mathbf{A}(i, j))$ is stored in memory. Each entry in the triple is usually stored in a different array and the whole matrix $\mathbf{A}$ is represented as three arrays $\mathbf{A}.\mathsf{I}$ (row indices), $\mathbf{A}.\mathsf{J}$ (column indices) and $\mathbf{A}.\mathsf{V}$ (numerical values), as illustrated in Figure 2.7. These separate arrays are

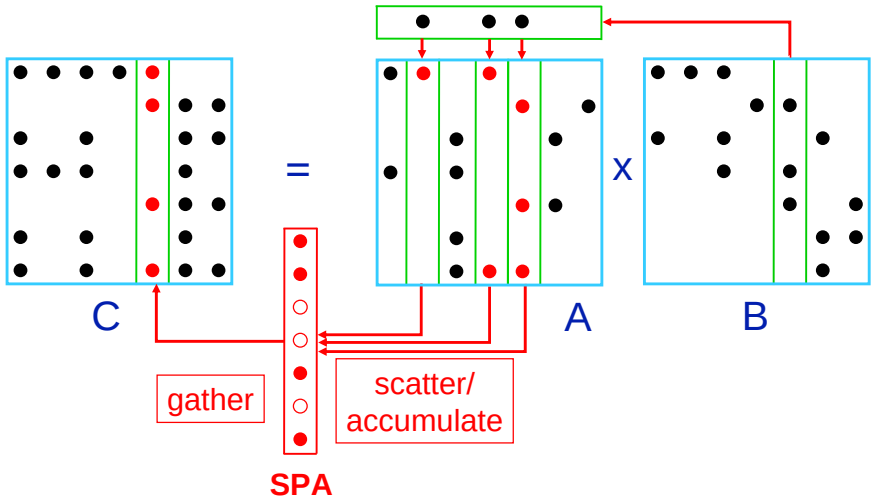


Figure 2.6: Multiplication of sparse matrices stored by columns. Columns of **A** are accumulated as specified by the non-zero entries in a column of **B** using a sparse accumulator or SPA. The contents of the SPA are stored in a column of **C** once all required columns are accumulated.

$$\mathrm{A} = \begin{pmatrix} 19 & 0 & 11 & 0 \\ 0 & 43 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 27 & 0 & 35 \end{pmatrix}$$

A.I	**A**.J	**A**.V
1	1	19
4	2	27
2	2	43
4	4	35
1	3	11

Figure 2.7: Matrix **A** (left) and an unordered triples representation (right)

called "parallel arrays" by Duff and Reid [84] but we reserve "parallel" for parallel algorithms. Using 8-byte integers for row and column indices, storage cost is $8 + 8 + 8 = 24$ bytes per nonzero.

Modern programming languages offer easier ways of representing an array of tuples than using three separate arrays. An alternative implementation might choose to represent the set of triples as an array of records (or structs). Such an implementation might improve cache performance, especially if the algorithm accesses elements of same index from different arrays. This cache optimization is known as array merging [132]. Some programming languages, such as Python and Haskell, even support tuples as built-in types, and C++ is about to add tuples support to its standard library [27]. In this section, we use the established notation of three separate arrays (**A**.I, **A**.J, **A**.V) for simplicity, but an implementer should keep in mind the other options.

This section evaluates triples format under different levels of ordering. The unordered triples representation imposes no ordering constraints on the triples.

Table 2.1: RAM Complexities of Key Primitives on Unordered and Row-Ordered Triples

	Unordered	Row Ordered
SpRef	$O(nnz(\mathbf{A}))$	$O(\lg nnz(\mathbf{A}) + nnz(\mathbf{A}(i,:)))\begin{cases}\mathbf{A}(i,j)\\ \mathbf{A}(i,:)\end{cases}$ $O(nnz(\mathbf{A}))\{\ \mathbf{A}(:,j)$
SpAsgn	$O(nnz(\mathbf{A})) + O(nnz(\mathbf{B}))$	$O(nnz(\mathbf{A})) + O(nnz(\mathbf{B}))$
SpMV	$O(nnz(\mathbf{A}))$	$O(nnz(\mathbf{A}))$
SpAdd	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$
SpGEMM	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + \text{flops})$	$O(nnz(\mathbf{A}) + \text{flops})$

Row-ordered triples keep nonzeros ordered with respect to their row indices only. Nonzeros within the same row are stored arbitrarily, irrespective of their column indices. Finally, row-major order keeps nonzeros ordered lexicographically first according to their row indices and then according to their column indices to break ties. Column-ordered and column-major ordered triples are similar; we analyze the row based versions. RAM and I/O complexities of key primitives for unordered and row-ordered triples are listed in Tables 2.1 and 2.2.

A theoretically attractive fourth option is to use hashing and store triples in a hash table. In the case of SpGEMM and SpAdd, dynamically managing the output matrix is computationally expensive since dynamic perfect hashing does not yield high performance in practice [149], and requires $35n$ space [76]. A

Table 2.2: I/O Complexities of Key Primitives on Unordered and Row-Ordered Triples

	Unordered	Row Ordered
SpRef	$O(scan(\mathbf{A}))$	$O(\lg nnz(\mathbf{A}) + scan(\mathbf{A}(i,:))\begin{cases}\mathbf{A}(i,j)\\ \mathbf{A}(i,:)\end{cases}$ $O(scan(\mathbf{A}))\{\ \mathbf{A}(:,j)$
SpAsgn	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$
SpMV	$O(nnz(\mathbf{A}))$	$O(nnz(\mathbf{A}))$
SpAdd	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$
SpGEMM	$O(nnz(\mathbf{A})+nnz(\mathbf{B})+\text{flops})$	$O(\min\{nnz(\mathbf{A}) + \text{flops},$ $scan(\mathbf{A})\ \lg(nnz(\mathbf{B}))+\text{flops}\})$

recently proposed dynamic hashing method called Cuckoo hashing is promising. It supports queries in worst-case constant time, and updates in amortized expected constant time, while using only $2n$ space [157]. Experiments show that it is substantially faster than existing hashing schemes on modern architectures like Pentium 4 and IBM Cell [169]. Although hash based schemes seem attractive, especially for SpAsgn and SpRef primitives [14], further research is required to test their efficiency for sparse matrix storage.

2.3.1 Unordered Triples

The administrative overhead of the triples representation is low, especially if the triples are not sorted in any order. With unsorted triples, however, there is

no spatial locality[2] when accessing nonzeros of a given row or column. In the worst case, all indexing operations might require a complete scan of the data structure. Therefore, SpRef has $O(nnz(\mathbf{A}))$ RAM complexity and $O(scan(\mathbf{A}))$ I/O complexity.

SpAsgn is no faster either, even though insertions take only constant time per element. In addition to accessing all the elements of the right hand side matrix $\mathbf{A}$, SpAsgn also invalidates the existing nonzeros that need to be changed in the left hand side matrix $\mathbf{B}$. Just finding those triples takes time proportional to the number of nonzeros in $\mathbf{B}$ with unordered triples. Thus, the RAM complexity of SpAsgn is $O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$ and its I/O complexity is $O(scan(\mathbf{A}) + scan(\mathbf{B}))$. A simple implementation achieving these bounds performs a single scan of $\mathbf{B}$, outputs only the non-assigned triples (e.g. for $\mathbf{B}(:, l) = \mathbf{A}$, those are the triples $(i, j, \mathbf{B}(i, j))$ where $j \neq l$), and finally concatenates the nonzeros in $\mathbf{A}$ to the output.

SpMV has full spatial locality when accessing the elements of $\mathbf{A}$, because the algorithm scans all the nonzeros of $\mathbf{A}$ in the exact order that they are stored. Therefore, $O(scan(\mathbf{A}))$ cache misses are taken for granted as compulsory misses[3]. Although SpMV is optimal in the RAM model without any ordering constraints,

[2] A procedure exploits spatial locality if data that are stored in nearby memory locations are likely to be referenced close in time

[3] Assuming that no explicit data prefetching mechanism is used

```
y : ℝ^m = TRIPLES-SPMV(A : ℝ^{S(m×n)}, x : ℝ^n)
1  y ← 0
2  for k ← 1 to nnz(A)
3        do y(A.I(k)) ← y(A.I(k)) + A.V(k) · x(A.J(k))
```

Figure 2.8: Operation $\mathbf{y} \leftarrow \mathbf{Ax}$ using triples

its cache performance suffers, as the algorithm cannot exploit any spatial locality when accessing vectors $\mathbf{x}$ and $\mathbf{y}$.

Considering the cache misses involved, for each triple $(i, j, \mathbf{A}(i, j))$, a random access to the jth component of $\mathbf{x}$ is required and the result of the elementwise multiplication $\mathbf{A}(i, j) \cdot \mathbf{x}(j)$ must be written to the random location $\mathbf{y}(i)$. Assumption 2 implies that the fast memory is not big enough to hold the dense arrays $\mathbf{x}$ and $\mathbf{y}$. Thus, we make up to two extra cache misses per flop. These indirect memory accesses can be clearly seen in the Triples-SpMV code shown in Figure 2.8, where the values of $\mathbf{A}.\mathsf{I}(k)$ and $\mathbf{A}.\mathsf{J}(k)$ may change in every iteration. Consequently, I/O complexity of SpMV on unordered triples is:

$$nnz(\mathbf{A})/L + 2 \cdot nnz(\mathbf{A}) = O(nnz(\mathbf{A})) \tag{2.1}$$

The SpAdd algorithm needs to identify all (i, j) pairs such that $\mathbf{A}(i, j) \neq 0$ and $\mathbf{B}(i, j) \neq 0$, and add their values to create a single entry in the resulting matrix. This can be accomplished by first sorting the nonzeros of the input matrices

and then performing a simultaneous scan of sorted nonzeros to sum matching triples. Using a linear time counting sort, SpAdd is fast in the RAM model with $O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$ complexity.

Counting sort, in its naïve form, has poor cache utilization because the total size of the counting array is likely to be bigger than the size of the fast memory. While sorting the nonzeros of a sparse matrix, this translates into one cache miss per nonzero in the worst case. Therefore, the complexity of SpAdd in the I/O model becomes $O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$. The number of cache misses can be decreased by using cache optimal sorting algorithms [6] but such algorithms are comparison based. They do $O(n \lg n)$ work as opposed to linear work. Rahman and Raman [165] give a counting sort algorithm that has better cache utilization in practice than the naïve algorithm, and still does linear work.

SpGEMM needs fast access to columns, rows, or a given particular element, depending on the algorithm. One can also think $\mathbf{A}$ as a table of i's and l's and $\mathbf{B}$ as a table of l's and j's; then $\mathbf{C}$ is their join [130] on l. This database analogy may lead to alternative SpGEMM implementations based on ideas from databases. An outer-product formulation of SpGEMM on unordered triples has three basic steps (a similar algorithm for general sparse tensor multiplication is given by Bader and Kolda [15]):

1. For each $l \in \{1, .., k\}$, identify the set of triples that belong to the lth column of $\mathbf{A}$, and the lth row of $\mathbf{B}$. Formally, find $\mathbf{A}(:, l)$ and $\mathbf{B}(l, :)$.

2. For each $l \in \{1, .., k\}$, compute the cartesian product of the row indices of $\mathbf{A}(:, l)$ and the column indices of $\mathbf{B}(l, :)$.

   Formally, compute the sets $\mathbf{C}_l = \{\mathbf{A}(:, l).\mathsf{I}\} \times \{\mathbf{B}(l, :).\mathsf{J}\}$

3. Find the union of all cartesian products, summing up duplicates during set union: $\mathbf{C} = \bigcup_{l \in \{1,..,k\}} \mathbf{C}_l$

Step 1 of the algorithm can be efficiently implemented by sorting the triples of $\mathbf{A}$ according to their column indices and the triples of $\mathbf{B}$ according to their row indices. Computing the cartesian products in Step 2 takes time

$$\sum_{l=1}^{k} nnz(\mathbf{A}(:, l)) \cdot nnz(\mathbf{B}(l, :)) = \text{flops}. \tag{2.2}$$

Finally, summing up duplicates can be done by lexicographically sorting the elements from sets $\mathbf{C}_l$. Since there are a total of flops such intermediate triples in sets $\mathbf{C}_l$ for all l, it makes up a total running time of

$$O(sort(nnz(\mathbf{A})) + sort(nnz(\mathbf{B})) + \text{flops} + sort(\text{flops})). \tag{2.3}$$

As long as the number of nonzeros is more than the dimensions of the matrices (Assumption 1), it is advantageous to use a linear time sorting algorithm instead

of a comparison based sort. Since a lexicographic sort is not required for finding $\mathbf{A}(:, l)$ or $\mathbf{B}(l, :)$ in Step 1, a single pass of linear time counting sort [68] suffices for each input matrix. However, two passes of linear time counting sort are required in Step 3 to produce a lexicographically sorted output. The RAM complexity of this implementation turns out to be

$$nnz(\mathbf{A}) + nnz(\mathbf{B}) + 3 \cdot \text{flops} = O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + \text{flops}). \tag{2.4}$$

However, due to the cache inefficient nature of counting sort, this algorithm makes $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + \text{flops})$ cache misses in the worst case.

Another way to implement SpGEMM on unordered triples is to iterate through the triples of $\mathbf{A}$. For each $(i, j, \mathbf{A}(i, j))$, we find $\mathbf{B}(:, j)$ and multiply $\mathbf{A}(i, j)$ by each nonzero in $\mathbf{B}(:, j)$. The duplicate summation step is left intact. The time this implementation takes is

$$nnz(\mathbf{A}) \cdot nnz(\mathbf{B}) + 3 \cdot \text{flops} = O(nnz(\mathbf{A}) \cdot nnz(\mathbf{B})). \tag{2.5}$$

The term flops is dominated by the term $nnz(\mathbf{A}) \cdot nnz(\mathbf{B})$ according to Theorem 1. Therefore, the performance is worse than the previous implementation that sorts the input matrices first.

Theorem 1. *For all matrices* $\mathbf{A}$ *and* $\mathbf{B}$, $\text{flops}(\mathbf{AB}) \leq nnz(\mathbf{A}) \cdot nnz(\mathbf{B})$

Proof. Let the vector of column counts of $\mathbf{A}$ be

$$\mathbf{a} = (a_1, a_2, ..., a_k) = (nnz(\mathbf{A}(:, 1), nnz(\mathbf{A}(:, 2)), ..., nnz(\mathbf{A}(:, k))$$

and the vector of row counts of $\mathbf{B}$

$$\mathbf{b} = (b_1, b_2, ..., b_k) = (nnz(\mathbf{B}(1,:), nnz(\mathbf{B}(2,:)), ..., nnz(\mathbf{B}(k,:)).$$

Note that flops $= \mathbf{a}^\mathsf{T}\mathbf{b} = \sum_{i=j} a_i b_j$, and $\sum_{i \neq j} a_i b_j \geq 0$ as $\mathbf{a}$ and $\mathbf{b}$ are nonnegative. Consequently,

$$nnz(\mathbf{A}) \cdot nnz(\mathbf{B}) = (\sum_{l=1}^{k} a_l) \cdot (\sum_{l=1}^{k} b_l) = (\sum_{i=j} a_i b_j) + (\sum_{i \neq j} a_i b_j)$$
$$\geq \sum_{i=j} a_i b_j = \mathbf{a}^\mathsf{T}\mathbf{b} = \text{flops}$$

□

It is worth noting that both implementions of SpGEMM using unordered triples have $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + \text{flops})$ space complexity, due to the intermediate triples that are all present in the memory after Step 2. Ideally, the space complexity of SpGEMM should be $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + nnz(\mathbf{C}))$, which is independent of flops.

2.3.2 Row-Ordered Triples

The second option is to keep the triplets sorted according to their rows or columns only. We analyze the row-ordered version; column order is symmetric. This section is divided into three subsections. The first one is on indexing and SpMV. The second one is on a fundamental abstract data type that is used

frequently in sparse matrix algorithms, namely the sparse accumulator (SPA). The SPA is used for implementing some of the SpAdd and SpGEMM algorithms throughout the rest of this chapter. Finally, the last subsection is on SpAdd and SpGEMM algorithms.

Indexing and SpMV with Row-Ordered Triples

Using row-ordered triples, indexing still turns out to be inefficient. In practice, even a fast row access cannot be accomplished, since there is no efficient way of spotting the beginning of the ith row without using a pointer[4]. Row-wise referencing can be done by performing binary search on the whole matrix to identify a nonzero belonging to the referenced row, and then by scanning in both directions to find the rest of the nonzeros belonging to that row. Therefore, SpRef for $\mathbf{A}(i,:)$ has $O(\lg nnz(\mathbf{A}) + nnz(\mathbf{A}(i,:))$ RAM complexity and $O(\lg nnz(\mathbf{A}) + scan(\mathbf{A}(i,:)))$ I/O complexity. Element-wise referencing also has the same cost, in both models. Column-wise referencing, on the other hand, is as slow as it was with unordered triples, requiring a complete scan of the triples.

SpAsgn might incur excessive data movement, as the number of nonzeros in the left hand side matrix $\mathbf{B}$ might change during the operation. For a concrete example, consider the operation $\mathbf{B}(i,:) = \mathbf{A}$ where $nnz(\mathbf{B}(i,:)) \neq nnz(\mathbf{A})$ before

[4]That is a drawback of the triples representation in general. The compressed sparse storage formats described in the Section 2.4 provide efficient indexing mechanisms for either rows or columns.

the operation. Since the data structure needs to keep nonzeros with increasing row indices, all triples with row indices bigger than i need to be shifted by distance $|\, nnz(\mathbf{A}) - nnz(\mathbf{B}(i,:))|$.

SpAsgn has RAM complexity $O(nnz(\mathbf{A}))+nnz(\mathbf{B}))$ and I/O complexity $O(1+ scan(\mathbf{A}) + scan(\mathbf{B}))$, where $\mathbf{B}$ is the left hand side matrix before the operation. While implementations of row-wise and element-wise referencing are straightforward, column-wise referencing ($\mathbf{B}(:,i) \leftarrow \mathbf{A}$) seems harder as it reduces to a restricted case of SpAdd, the restriction being that $\mathbf{B} \in \mathbb{S}^{1\times n}$ has at most one nonzero in a given row. Therefore, a similar scanning implementation suffices.

Row-ordered triples format allows an SpMV implementation that makes at most one extra cache miss per flop. The reason is that references to vector $\mathbf{y}$ show good spatial locality: they are ordered with monotonically increasing values of $\mathbf{A}.\mathsf{I}(k)$, avoiding scattered memory referencing on vector $\mathbf{y}$. However, accesses to vector $\mathbf{x}$ are still irregular as the memory stride when accessing $\mathbf{x}$ ($|\mathbf{A}.\mathsf{J}(k+1)$ - $\mathbf{A}.\mathsf{J}(k)|$) might be as big as the matrix dimension n. Memory strides can be reduced by clustering the nonzeros in every row. More formally, this corresponds to reducing the bandwidth of the matrix, which it is defined as: $\beta(\mathbf{A}) = max\{|i-j| : \mathbf{A}(i,j) \neq 0\}$. Toledo [197] experimentally studied different methods of reordering the matrix to reduce its bandwidth, along with other optimizations like blocking and prefetching, to improve the memory performance

of SpMV. Overall, row ordering does not improve the asymptotic I/O complexity of SpMV over unordered triples, although it cuts the cache misses by nearly half. Its I/O complexity becomes

$$nnz(\mathbf{A})/L + n/L + nnz(\mathbf{A}) = O(nnz(\mathbf{A})). \tag{2.6}$$

The Sparse Accumulator

Most operations that output a sparse matrix generate it one row (or column) at a time. The current active row is stored temporarily in a special structure called the sparse accumulator (SPA) [104] (or expanded real accumulator [163]). The SPA helps merging unordered lists in linear time.

There are different ways of implementing the SPA as it is an abstract data type, not a concrete data structure. In our SPA implementation, $\mathbf{w}$ is the dense vector of values, $\mathbf{b}$ is the boolean dense vector that contains "occupied" flags, and LS is the list that keeps an unordered list of indices, as Gilbert et al. described [104].

`Scatter-SPA` function, given in Figure 2.9, adds a scalar (*value*) to a specific position (*pos*) of the SPA. Scattering is a constant time operation. Gathering the SPA's nonzeros to the output matrix $\mathbf{C}$ takes $O(nnz(\mathsf{SPA}))$ time. The pseudocode for the `Gather-SPA` is given in Figure 2.10. It is crucial to initialize the SPA only once at the beginning, as this takes $O(n)$ time. Resetting it later for the next

```
SCATTER-SPA(SPA, value, pos)
1  if (SPA.b(pos) = 0)
2     then SPA.w(pos) ← value
3          SPA.b(pos) ← 1
4          INSERT(SPA.LS, pos)
5     else SPA.w(pos) ← SPA.w(pos) + value
```

Figure 2.9: Scatters/Accumulates the nonzeros in the SPA

```
nzi = GATHER-SPA(SPA, val, col, nzcur)
1  cptr ← head(SPA.LS)
2  nzi ← 0                          ▷ number of nonzeros in the ith row of C
3  while cptr ≠ NIL
4        do
5           col(nzcur + nzi) ← element(cptr)          ▷ Set column index
6           val(nzcur + nzi) ← SPA.w(element(cptr))   ▷ Set value
7           nzi ← nzi +1
8           ADVANCE(cptr)
```

Figure 2.10: Gathers/Outputs the nonzeros in the SPA

active row takes only $O(nnz(\mathsf{SPA}))$ time by using LS to reach all the nonzero elements and resetting only those indices of **w** and **b**.

The cost of resetting the SPA can be completely avoided by using the *multiple switch* technique (also called the *phase counter* technique) described by Gustavson [111, 112]. Here, **b** becomes a dense *switch vector* of integers instead of a dense boolean vector. For computing each row, we use a different switch value. Everytime a nonzero is introduced to position *pos* of the SPA, we set the switch

```
C : ℝ^{S(m×n)} = RowTriples-SpAdd(A : ℝ^{S(m×n)}, B : ℝ^{S(m×n)})
 1  Set-SPA(SPA)                          ▷ Set w = 0, b = 0 and create empty list LS
 2  ka ← kb ← kc ← 1                      ▷ Initialize current indices to one
 3  for i ← 1 to m
 4        do while (ka ≤ nnz(A) and A.I(ka) = i)
 5              do Scatter-SPA(SPA, A.V(ka), A.J(ka))
 6                 ka ← ka + 1
 7           while (kb ≤ nnz(B) and B.I(kb) = i)
 8              do Scatter-SPA(SPA, B.V(kb), B.J(kb))
 9                 kb ← kb + 1
10           nznew ← Gather-SPA(SPA, C.V, C.J, kc)
11           for j ← 0 to nznew −1
12              do C.I(kc + j) ← i                    ▷ Set row index
13           kc ← kc + nznew
14           Reset-SPA(SPA)                ▷ Reset w = 0, b = 0 and empty LS
```

Figure 2.11: Operation $\mathbf{C} \leftarrow \mathbf{A} \oplus \mathbf{B}$ using row-ordered triples

to the current active row index ($\mathsf{SPA}.\mathbf{b}(pos) \leftarrow i$). During the computation of subsequent rows $j = \{i+1, ..., m\}$, the switch value being less than the current active row index ($\mathsf{SPA}.\mathbf{b}(pos) \leq j$) means that the position pos of the SPA is "free". Therefore, the need to reset $\mathbf{b}$ for each row is avoided.

SpAdd and SpGEMM with Row-Ordered Triples

Using the SPA, we can implement SpAdd with $O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$ RAM complexity. The full procedure is given in Figure 2.11. The I/O complexity of this SpAdd implementation is also $O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$. This is because for each nonzero scanned from inputs, the algorithm checks and updates an arbitrary

position of the SPA. From Assumption 2, these arbitrary accesses are likely to incur cache misses every time.

It is possible to implement SpGEMM using the same outer-product formulation described in Section 2.3.1, with a slightly better asymptotic RAM complexity of $O(nnz(\mathbf{A}) + \text{flops})$, as the triples of $\mathbf{B}$ are already sorted according to their row indices. Instead, we describe a row-wise implementation, similar to the CSR based algorithm described in Section 2.4. Due to inefficient row-wise indexing support of row-ordered triples, however, the operation count is higher than the CSR version. A SPA of size n is used to accumulate the nonzero structure of the current active row of $\mathbf{C}$. A direct scan of the nonzeros of $\mathbf{A}$ allows enumeration of nonzeros in $\mathbf{A}(i,:)$ for increasing values of $i \in \{1,..,m\}$. Then, for each triple $(i, l, \mathbf{A}(i,l))$ in the ith row of $\mathbf{A}$, the matching triples $(l, j, \mathbf{B}(l,j))$ of the lth row of $\mathbf{B}$ need to be found using the SpRef primitive. This way, the nonzeros in $\mathbf{C}(i,:)$ are accumulated. The whole procedure is given in Figure 2.12. Its RAM complexity is

$$\sum_{\mathbf{A}(i,l)\neq 0} \big(nnz(\mathbf{B}(i,:))\ \lg(nnz(\mathbf{B}))\big) + \text{flops} = O(nnz(\mathbf{A})\ \lg(nnz(\mathbf{B})) + \text{flops}) \quad (2.7)$$

where the $\lg nnz(B)$ factor per each nonzero in $\mathbf{A}$ comes from the row-wise SpRef operation in line 5. Its I/O complexity is

$$O(scan(\mathbf{A})\ \lg(nnz(\mathbf{B})) + \text{flops}). \quad (2.8)$$

```
C : ℝ^{S(m×n)} = RowTriples-SpGEMM(A : ℝ^{S(m×k)}, B : ℝ^{S(k×n)})
 1  Set-SPA(SPA)                    ▷ Set w = 0, b = 0 and create empty list LS
 2  ka ← kc ← 1                                  ▷ Initialize current indices to one
 3  for i ← 1 to m
 4        do while (ka ≤ nnz(A) and A.I(ka) = i)
 5              do BR ← B(A.J(ka), :)                              ▷ Using SpRef
 6                 for kb ← 1 to nnz(BR)
 7                       do value ← A.NUM(ka) · BR.NUM(kb)
 8                          Scatter-SPA(SPA, value, BR.J(kb))
 9                 ka ← ka + 1
10           znew ← Gather-SPA(SPA, C.V, C.J, kc)
11           for j ← 0 to znew −1
12                 do C.I(kc + j) ← i                              ▷ Set row index
13           kc ← kc + znew
14           Reset-SPA(SPA)              ▷ Reset w = 0, b = 0 and empty LS
```

Figure 2.12: Operation $\mathbf{C} \leftarrow \mathbf{AB}$ using row-ordered triples

While the complexity of row-wise implementation is asymptotically worse than the outer-product implementation in the RAM model, it has the advantage of using only $O(nnz(\mathbf{C}))$ space as opposed to the $O(\text{flops})$ space used by the outer-product implementation. On the other hand, the I/O complexities of the outer-product version and the row-wise version are not directly comparable. Which one is faster depends on the cache line size and the number of nonzeros in $\mathbf{B}$.

2.3.3 Row-Major Ordered Triples

We now consider the third option of storing triples in lexicographic order, either in column-major or row-major order. Once again, we focus on the row oriented scheme.

In order to reference a whole row, binary search on the whole matrix, followed by a scan on both directions is used, as with row-ordered triples. As the nonzeros in a row are ordered by column indices, it seems there should be a faster way to access a single element than the method used on row-ordered triples. A faster way indeed exists but ordinary binary search would not do it, because the beginning and the end of the ith row is not known in advance. The algorithm has three steps:

1. Spot a triple $(i, j, \mathbf{A}(i, j))$ that belongs to the ith row by doing binary search on the whole matrix

2. From that triple, perform an unbounded binary search [147] in both directions. In an unbounded search, the step length is doubled at each iteration. The search terminates at a given direction when it hits a triple that does not belong to the ith row. Those two triples (one from each direction) becomes the boundary triples.

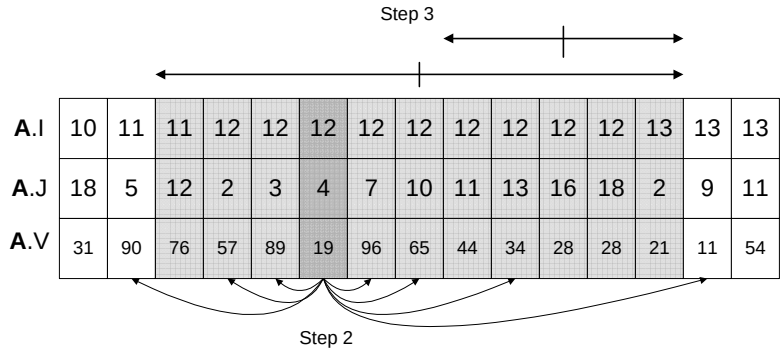


Figure 2.13: Element-wise indexing of $\mathbf{A}(12, 16)$ on row-major ordered triples

3. Perform ordinary binary search within the exclusive range defined by the boundary vertices.

The number of total operations is $O(\lg nnz(\mathbf{A}) + \lg nnz(\mathbf{A}(i,:)) = O(\lg nnz(\mathbf{A}))$. An example is given in Figure 2.13.

While unbounded binary search is the preferred method in the RAM model, simple scanning might be faster in the I/O model. Searching an element in an ordered set of n elements can be achieved with $\Theta(\log_L n)$ cost in the I/O model, using B-trees [26]. However, using an ordinary array, search incurs $\lg n$ cache misses. This may or may not be less than $scan(n)$. Therefore, we define the cost of searching within an ordered row as follows:

$$search(\mathbf{A}(i,:)) = \min\{\lg nnz(\mathbf{A}(i,:)), scan(\mathbf{A}(i,:))\} \tag{2.9}$$

For column-wise referencing as well as for SpAsgn operations, row-major ordered triples format does not provide any improvement over row-ordered triples.

In SpMV, the only array that does not show excellent spatial locality is $\mathbf{x}$, since $\mathbf{A}.\mathsf{I}, \mathbf{A}.\mathsf{J}, \mathbf{A}.\mathsf{V}$, and $\mathbf{y}$ are accessed with mostly consecutive, increasing index values. Accesses to $\mathbf{x}$ are also with increasing indices, which is an improvement over row-ordered triples. However, memory strides when accessing $\mathbf{x}$ can still be high, depending on the number of nonzeros in each row and the bandwidth of the matrix. In the worst case, each access to $\mathbf{x}$ might incur a cache miss.

Bender et al. [28] came up with cache-optimal algorithms for SpMV using the column-major layout. From a high-level view, their method first generates all the intermediate triples of $\mathbf{y}$, possibly with repeating indices. Then, the algorithm sorts those intermediate triples with respect to their row indices, performing additions on the triples with same row index on the fly. I/O optimality of their SpMV algorithm relies on the existence of an I/O optimal sorting algorithm. Their complexity measure assumes a fixed k number of nonzeros per column, leading to I/O complexity of

$$O\left(scan(\mathbf{A}) \log_{Z/L} \frac{n}{max\{Z, k\}}\right). \tag{2.10}$$

SpAdd is now more efficient even without using any auxiliary data structure. A scan-based array merging algorithm is sufficient as long as we sum duplicates while

Table 2.3: RAM Complexities of Key Primitives on Row-Major Ordered Triples and CSR

	Row-Major Ordered Triples	CSR
SpRef	$O(\lg nnz(\mathbf{A}))\{\ \mathbf{A}(i,j)$ $O(\lg nnz(\mathbf{A}) + nnz(\mathbf{A}(i,:))\{\ \mathbf{A}(i,:)$ $O(nnz(\mathbf{A}))\{\ \mathbf{A}(:,j)$	$O(\lg nnz(\mathbf{A}(i,:))\{\ \mathbf{A}(i,j)$ $O(nnz(\mathbf{A}(i,:))\{\ \mathbf{A}(i,:)$ $O(nnz(\mathbf{A}))\{\ \mathbf{A}(:,j)$
SpAsgn	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$
SpMV	$O(nnz(\mathbf{A}))$	$O(nnz(\mathbf{A}))$
SpAdd	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$	$O(nnz(\mathbf{A}) + nnz(\mathbf{B}))$
SpGEMM	$O(nnz(\mathbf{A})+\text{flops})$	$O(nnz(\mathbf{A})+\text{flops})$

merging. Such an implementation has $O(nnz(\mathbf{A})+nnz(\mathbf{B}))$ RAM complexity and $O(scan(\mathbf{A}) + scan(\mathbf{B}))$ I/O complexity[5].

Row-major ordered triples allow outer-product and row-wise SpGEMM implementations at least as efficiently as row-ordered triples. Indeed, some finer improvements are possible by exploiting the more specialized structure. In the case of row-wise SpGEMM, a technique called *finger search* [43] can be used to improve the RAM complexity. While enumerating all triples $(i, l, \mathbf{A}) \in \mathbf{A}(i,:)$, they are naturally sorted with increasing l values. Therefore, accesses to $\mathbf{B}(l,:)$ are also with increasing l values. Instead of restarting the binary search from the beginning of $\mathbf{B}$, one can use fingers and only search the yet unexplored subse-

[5]These bounds are optimal only if $nnz(\mathbf{A}) = \Theta(nnz(\mathbf{B}))$ [46]

Table 2.4: I/O Complexities of Key Primitives on Row-Major Ordered Triples and CSR

	Row-Major Ordered Triples	CSR
SpRef	$O(\lg nnz(\mathbf{A}) + search(\mathbf{A}(i,:))\{\ \mathbf{A}(i,j)$ $O(\lg nnz(\mathbf{A}) + scan(\mathbf{A}(i,:))\{\ \mathbf{A}(i,:)$ $O(scan(\mathbf{A}))\{\ \mathbf{A}(:,j)$	$O(search(\mathbf{A}(i,:)))\{\ \mathbf{A}(i,j)$ $O(scan(\mathbf{A}(i,:)))\{\ \mathbf{A}(i,:)$ $O(1+scan(\mathbf{A}))\{\ \mathbf{A}(:,j)$
SpAsgn	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$
SpMV	$O(nnz(\mathbf{A}))$	$O(nnz(\mathbf{A}))$
SpAdd	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$	$O(scan(\mathbf{A}) + scan(\mathbf{B}))$
SpGEMM	$O(\min\{nnz(\mathbf{A}) + \text{flops},$ $scan(\mathbf{A})\ \lg(nnz(\mathbf{B}))+\text{flops}\})$	$O(scan(\mathbf{A}) + \text{flops})$

quence. Finger search uses the unbounded binary search as a subroutine when searching the unexplored subsequence. Row-wise SpGEMM using finger search has a RAM complexity of

$$O(\text{flops}) + \sum_{i=1}^{n} O\big(nnz(\mathbf{A}(i,:))\ \lg \frac{nnz(\mathbf{B})}{nnz(\mathbf{A}(i,:))}\big), \tag{2.11}$$

which is asymptotically faster than the $O(nnz(\mathbf{A})\ \lg(nnz(\mathbf{B})) + \text{flops})$ cost of the same algorithm on row-ordered triples.

Outer-product SpGEMM can be modified to use only $O(nnz(\mathbf{C}))$ space during execution by using multiway merging [49]. However, this comes at the price of an extra $\lg ni$ factor in the asymptotic RAM complexity, where ni is the number of indices i for which $\mathbf{A}(:,i) \neq \emptyset$ and $\mathbf{B}(i,:) \neq \emptyset$.

Although both of these refined algorithms are asymptotically slower than the naive outer-product method, they might be faster in practice because of the cache effects and difference in constants in the asymptotic complexities. Further research is required in algorithm engineering of SpGEMM to find the best performing algorithm in real life. Chapter 3 includes extensive experiments, for two new SpGEMM algorithms, under various settings.

2.4 Compressed Sparse Row/Column

The most widely used storage schemes for sparse matrices are compressed sparse column (CSC) and compressed sparse row (CSR). For example, MATLAB uses CSC format to store its sparse matrices [104]. Both are dense collections of sparse arrays. We examine CSR, which is introduced by Gustavson under the name of sparse row-wise representation [110]; CSC is symmetric.

CSR can be seen as a concatenation of sparse row arrays. On the other hand, it is also very close to row-ordered triples, with an auxiliary index of size $\Theta(n)$. In this section, we assume that nonzeros within each sparse row array are ordered with increasing row indices. This is not a general requirement though. Davis's CSparse package [74], for example, does not impose any ordering within the sparse arrays.

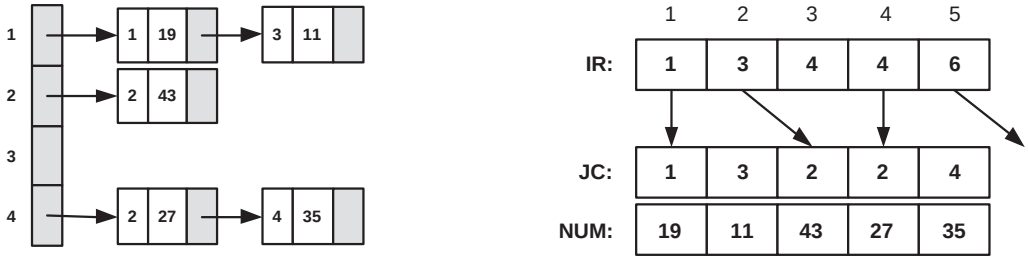


Figure 2.14: Adjacency list (left) and CSR (right) representations of matrix **A** from Figure 2.7

2.4.1 CSR and Adjacency Lists

In principle, CSR is almost identical to the adjacency list representation of a directed graph [191]. In practice, however, it has much less overhead and much better cache efficiency. Instead of storing an array of linked lists as in the adjacency list representation, CSR is composed of three arrays that store whole rows contiguously. The first array (IR) of size $m+1$ stores the row pointers as explicit integer values, the second array (JC) of size nnz stores the column indices, and the last array (NUM) of size nnz stores the actual numerical values. Observe that column indices stored in the JC array indeed come from concatenating the edge indices of the adjacency lists. Following the sparse matrix / graph duality, it is also meaningful to call the first array the vertex array and the second array the edge array. The vertex array holds the offsets to the edge array, meaning that the nonzeros in the ith row are stored from NUM(IR(i)) to NUM(IR($i+1$)-1) and their

respective positions within that row are stored from $\mathsf{JC}(\mathsf{IR}(i))$ to $\mathsf{JC}(\mathsf{IR}(i+1)-1)$. Also note that $\mathsf{JC}(i) = \mathsf{JC}(i+1)$ means there are no nonzeros in the ith row.

Figure 2.4.1 shows the adjacency list and CSR representations of matrix $\mathbf{A}$ from Figure 2.7 . While the arrows in the adjacency based representation are actual pointers to memory locations, the arrows in CSR are not. The edge array offsets are actually (unsigned) integers.

The efficiency advantage of the CSR data structure as compared with the adjacency list can be explained by the memory architecture of modern computers. In order to access all the nonzeros in a given row i, which is equivalent to traversing all the outgoing edges of a given vertex v_i, CSR makes at most $\lceil nnz(\mathbf{A}(i,:))/L \rceil$ cache misses. A similar access to the adjacency list representation incurs $nnz(\mathbf{A}(i,:))$ cache misses in the worst case, worsening as the memory becomes more and more fragmented. In an experiment published in 1998, Black et al. [31] found out that an array based representation was 10 times faster to traverse than a linked-list based representation. This performance gap is due to the high cost of pointer chasing that happens frequently in linked data structures. The efficiency of CSR comes at a price though: introducing new nonzero elements or deleting a nonzero element is computationally inefficient [104]. Therefore, CSR is best suited for representing static graphs. The only one of our key primitives that changes the graph is SpAsgn.

```
y : ℝ^m = CSR-SpMV(A : ℝ^{S(m)×n}, x : ℝ^n)
1  y ← 0
2  for i ← 1 to m
3        do for k ← A.IR(i) to A.IR(i + 1) − 1
4              do y(i) ← y(i) + A.NUM(k) · x(A.JC(k))
```

Figure 2.15: Operation $\mathbf{y} \leftarrow \mathbf{Ax}$ using CSR

2.4.2 CSR on Key Primitives

Unlike triples storage formats, CSR allows constant time random access to any row of the matrix. Its ability to enumerate all the elements in the ith row with $O(nnz(\mathbf{A}(i,:))$ RAM complexity and $O(scan(\mathbf{A}(i,:))$ I/O complexity makes it an excellent data structure for row-wise SpRef. Element-wise referencing takes at most $O(\lg nnz(\mathbf{A}(i,:))$ time in the RAM model as well as the I/O model, using a binary search. Considering column-wise referencing, however, CSR does not provide any improvement over the triples format.

On the other hand, even row-wise SpAsgn operations are inefficient if the number of elements in the assigned row changes. In that general case, $O(nnz(\mathbf{B}))$ elements might need to be moved. This is also true for column-wise and element-wise SpAsgn as long as not just existing nonzeros are reassigned to new values.

The code in Figure 2.15 shows how to perform SpMV when matrix $\mathbf{A}$ is represented in CSR format. This code and SpMV with row-major ordered triples has

similar performance characteristics except for a few subtleties. When some rows of $\mathbf{A}$ are all zeros, those rows are effectively skipped in row-major ordered triples, but still need to be examined in CSR. On the other hand, when $m << nnz$, CSR has a clear advantage since it needs to examine only one index ($\mathbf{A}.\mathsf{JC}(k)$) per inner loop iteration while row-major ordered triples needs to examine two ($\mathbf{A}.\mathsf{I}(k)$ and $\mathbf{A}.\mathsf{J}(k)$). This may make up to a factor of two difference in the number of cache misses. CSR also has some advantages over CSC when the SpMV primitive is considered (expecially in the case of $\mathbf{y} \leftarrow \mathbf{y} + \mathbf{A}\mathbf{x}$), as experimentally shown by Vuduc [206].

Blocked versions of CSR and CSC try to take advantage of clustered nonzeros in the sparse matrix. While blocked CSR (BCSR) achieves superior performance for SpMV on matrices resulting from finite element meshes [206] mostly by using loop unrolling and register blocking, it is of little use when the matrix itself does not have its nonzeros clustered. Pinar and Heath proposed a reordering mechanism to cluster those nonzeros to get dense subblocks [162]. However, it is not clear whether such mechanisms are successful for highly irregular matrices from sparse real world graphs.

Except for the additional bookkeeping required for getting the row pointers right, SpAdd can be implemented in the same way as with row-major ordered

triples. The extra bookkeeping of row pointers does not affect the asymptotic complexity.

One subtlety overlooked in the SpAdd implementations throughout this chapter is management of the memory required by the resulting matrix $\mathbf{C}$. We implicitly assumed that the data structure holding $\mathbf{C}$ has enough space to accommodate all of its elements. Repeated doubling of memory whenever necessary is one way of addressing this issue. Another conservative way is to reserve $nnz(\mathbf{A}) + nnz(\mathbf{B})$ space for $\mathbf{C}$ at the beginning of the procedure and shrink any unused portion after the computation, right before the procedure returns.

The efficiency of accessing and enumerating rows in CSR makes the row-wise SpGEMM formulation, described in Figure 2.4, the preferred matrix multiplication formulation. An efficient implementation of the row-wise SPGEMM using CSR was first given by Gustavson [113]. It had a RAM complexity of

$$O(m + n + nnz(\mathbf{A}) + \text{flops}) = O(nnz(\mathbf{A}) + \text{flops}), \tag{2.12}$$

where the equality follows from Assumption 1. Recent column-wise implementations with similar RAM complexities are provided by Davis in his CSparse software [74] and by MATLAB [104]. The algorithm, presented in Figure 2.16 uses the sparse accumulator (SPA) described in Section 2.3.2. Once again, the multiple switch technique can be used to avoid the cost of resetting the SPA for every

```
C : ℝ^{S(m)×n} = CSR-SpGEMM(A : ℝ^{S(m)×k}, B : ℝ^{S(k)×n})
 1  SET-SPA(SPA)                    ▷ Set w = 0, b = 0 and create empty list LS
 2  C.IR(1) ← 0
 3  for i ← 1 to m
 4        do for k ← A.IR(i) to A.IR(i + 1)
 5              do for j ← B.IR(A.JC(k)) to B.IR(A.JC(k) + 1)
 6                    do
 7                       value ← A.NUM(k) · B.NUM(j)
 8                       SCATTER-SPA(SPA, value, B.JC(j))
 9           nznew ← GATHER-SPA(SPA, C.NUM, C.JC, C.IR(i))
10           C.IR(i + 1) ← C.IR(i) + nznew
11           RESET-SPA(SPA)          ▷ Reset w = 0, b = 0 and empty LS
```

Figure 2.16: Operation $\mathbf{C} \leftarrow \mathbf{AB}$ using CSR

iteration of the outermost loop. As in the case of SpAdd, generally the space required to store $\mathbf{C}$ cannot be determined quickly. Repeated doubling or more sophisticated methods such as Cohen's algorithm [63] may be used. Cohen's algorithm is a randomized iterative algorithm that does $\Theta(1)$ SpMV operations over a semiring to estimate the row and column counts. It can be efficiently implemented even on unordered triples.

The row-wise SpGEMM implementation does $O(scan(\mathbf{A}) + \text{flops})$ cache misses in the worst case. Due to the size of the SPA and Assumption 2, the algorithm makes a cache miss for every flop. As long as no cache interference occurs between the nonzeros of $\mathbf{A}$ and the nonzeros of $\mathbf{C}(i, :)$, only $scan(\mathbf{A})$ additional cache misses are made instead of $nnz(\mathbf{A})$.

2.5 Other Related Work and Conclusion

In this chapter, we gave a brief survey of sparse matrix infrastructure for doing graph algorithms. We focused on implementation and analysis of key primitives using various standard sparse matrix data structures. We tried to complement the existing literature in two directions. First, we analyzed sparse matrix indexing and assignment operations. Second, we gave I/O complexity bounds for all operations. Taking I/O complexies into account is key to achieving high performance on modern architectures with multiple levels of cache.

A vast literature exists on sparse matrix storage schemes. We tried to cover the most general ones in this chapter. There are specialized data structures that perform certain computations more efficiently. For example:

- Blocked compressed stripe formats (BCSR and BCSC) [122] uses less bandwidth to accelerate bandwidth limited computations such as SpMV.

- Knuth storage [128] allows fast access to both rows and columns at the same time, and it makes dynamic changes to the matrix possible. Therefore, it is very suitable for all kinds of SpRef and SpAsgn operations. Its drawback is its excessive memory usage ($5\ nnz + 2n$) and high cache miss ratio.

- Hierarchical storage schemes such as quadtrees [173, 209] are theoretically attractive, but achieving good performance in practice requires careful al-

gorithm engineering to avoid high cache miss ratios that would result from straightforward pointer based implementations.

- Parallel data structures that are designed for multithreaded executions are becoming attractive with the multicore revolution. Our ***Compressed Sparse Blocks (CSB)*** format, introduced in Section 6 guarantees plenty of parallelism for multithreaded SpMV and SpMV_T (sparse matrix-transpose-dense vector multiplication) computations.

Chapter 3

New Ideas in Sparse Matrix-Matrix Multiplication

Part of the material in this chapter previously appeared in preliminary form in papers by Buluç and Gilbert that appeared in the proceedings of IPDPS'08 [49] and ICPP'08 [48].

3.1 Introduction

Development and implementation of large-scale parallel graph algorithms poses numerous challenges in terms of scalability and productivity [143, 210]. Linear algebra formulations of many graph algorithms already exist in the literature [7, 146, 192]. By exploiting the duality between matrices and graphs, linear algebraic formululations aim to apply the existing knowledge on parallel matrix algorithms to parallel graph algorithms. One of the key primitives in array-based

graph algorithms is computing the product of two sparse matrices (SpGEMM) over a semiring. Sparse matrix-matrix multiplication is a building block for many algorithms including graph contraction [105], breadth-first search from multiple source vertices, peer pressure clustering [180], recursive formulations of all-pairs shortest-paths algorithms [71], matching algorithms [164], and cycle detection [211], as well as for some other applications such as multigrid interpolation/restriction [42], and parsing context-free languages [161].

Most large graphs in applications, such as the WWW graph, finite element meshes, planar graphs, and trees, are sparse. In this work, we consider a graph to be sparse if $nnz = O(n)$, where nnz is the number of edges and n is the number of vertices. Dense matrix multiplication algorithms are inefficient for SpGEMM since they require $O(n^3)$ space and the current fastest dense matrix multiplication algorithm runs in $O(n^{2.38})$ [67, 177] time. Furthermore, fast dense matrix multiplication algorithms operate on a ring instead of a semiring, which makes them unsuitable for many algorithms on general graphs. For example, it is possible to embed the semiring into the ring of integers for the all-pairs shortest-paths problem on unweighted and undirected graphs [177], but the same embedding does not work for weighted or directed graphs [213].

The previous chapter explains CSC/CSR, the most widely used data structures for sparse matrices, in detail. It also gives concise descriptions of common

SpGEMM algorithms operating both on CSC/CSR and triples. The SpGEMM problem was recently reconsidered by Yuster and Zwick [212] over a ring, where the authors use a fast dense matrix multiplication such as arithmetic progression [67] as a subroutine. Their algorithm uses $O(nnz^{0.7}\, n^{1.2} + n^{2+o(1)})$ arithmetic operations, which is theoretically close to optimal only if we assume that the number of nonzeros in the resulting matrix $\mathbf{C}$ is $\Theta(n^2)$. This assumption rarely holds in reality. Instead, we provide a work sensitive analysis by expressing the computation complexity of our SpGEMM algorithms in terms of flops.

Practical sparse algorithms have been proposed by different researchers over the years [159, 189] using various data structures. Although they achieve reasonable performance on some classes of matrices, none of these algorithms outperforms the classical sparse matrix-matrix multiplication algorithm, which was first described by Gustavson [113] and was used in Matlab [104] and CSparse [74]. The classical algorithm runs in $O(\text{flops} + nnz + n)$ time.

In Section 3.2, we present two novel algorithms for sequential SpGEMM. The first one is geared towards computing the product of two ***hypersparse*** matrices. A matrix is hypersparse if the ratio of nonzeros to its dimension is asymptotically 0. It is used as the sequential building block of our parallel 2D algorithms described in Section 3.3. Our HYPERSPARSE_GEMM algorithm uses a new $O(nnz)$ data structure, called ***DCSC*** for ***doubly compressed sparse columns***, which

is explained in Section 3.2.1. The HYPERSPARSE_GEMM is based on the outer-product formulation and has time complexity $O(nzc(\mathbf{A}) + nzr(\mathbf{B}) + \text{flops} \cdot \lg ni)$, where $nzc(\mathbf{A})$ is the number of columns of $\mathbf{A}$ that contain at least one nonzero, $nzr(\mathbf{B})$ is the number of rows of $\mathbf{B}$ that contain at least one nonzero, and ni is the number of indices i for which $\mathbf{A}(:, i) \neq \emptyset$ and $\mathbf{B}(i, :) \neq \emptyset$. The overall space complexity of our algorithm is only $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + nnz(\mathbf{C}))$. Notice that the time complexity of our algorithm does not depend on n, and the space complexity does not depend on flops.

The second sequential algorithm is an ordered variant of the column-by-column formulation, and has better expected time complexity for random matrices, but worse worst-case time complexity in general. It is specifically geared towards matrices with dimensions large enough to force a single dense column not to fit in cache. It works on any sparse matrix data structure that can enumerate nonzeros in the jth column in $O(nnz(j))$ time. We include a preliminary experimental evaluation of the column-by-column algorithm in this section, comparing it with the classical algorithm.

Section 3.3 presents parallel algorithms for SpGEMM. We propose novel algorithms based on 2D block decomposition of data in addition to giving the complete description of an existing 1D algorithm. To the best of our knowledge, parallel

algorithms using a 2D block decomposition have not earlier been developed for sparse matrix-matrix multiplication.

Toledo et al. [123] proved that 2D dense matrix multiplication algorithms are optimal with respect to the communication volume, making 2D sparse algorithms likely to be more scalable than their 1D counterparts. In Section 3.4, we show that this intuition is indeed correct by providing a theoretical analysis of the parallel performance of 1D and 2D algorithms.

In Section 3.5, we model the speedup of parallel SpGEMM algorithms using realistic simulations and projections. Our results show that existing 1D algorithms are not scalable to thousands of processors. By contrast, 2D algorithms have the potential for scaling up indefinitely, albeit with decreasing parallel efficiency, which is defined as the ratio of speedup to the number of processors.

Section 3.6 describes the experimental setup we used for evaluating our Sparse SUMMA implementation, and presents the final results. We describe other techniques we have used for implementating our parallel algorithms, and their effects on performance in Section 3.7. Section 3.8 offers some future directions.

3.2 Sequential Sparse Matrix Multiply

In this section, we first analyze different formulations of sparse matrix-matrix multiplication using the layered graph model. We present our HYPERSPARSE_GEMM algorithm in Section 3.2.1. Finally, we present our cache-efficient column-by-column algorithm in Section 3.2.2 together with its experimental evaluation.

Matrix multiplication can be organized in many different ways. The inner-product formulation that usually serves as the definition of matrix multiplication is well-known. Given two matrices $\mathbf{A} \in \mathbb{R}^{m \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times n}$, each element in the product $\mathbf{C} \in \mathbb{R}^{m \times n}$ is computed by the following formula:

$$\mathbf{C}(i,j) = \sum_{l=1}^{k} \mathbf{A}(i,l)\mathbf{B}(l,j). \tag{3.1}$$

This formulation is rarely useful for multiplying sparse matrices since it requires $\Omega(mn)$ operations regardless of the sparsity of the operands.

We represent the multiplication of two matrices $\mathbf{A}$ and $\mathbf{B}$ as a three layered graph, following Cohen [63]. The layers have m, k and n vertices, in that order. The first layer of vertices (U) represent the rows of $\mathbf{A}$ and the third layer of vertices (V) represent the columns of $\mathbf{B}$. The second layer of vertices (W) represent the dimension shared between matrices. Every nonzero $\mathbf{A}(i,l) \neq 0$ in the ith row of $\mathbf{A}$ forms an edge (u_i, w_l) between the first and second layers and every nonzero in

$\mathbf{B}(l, j) \neq 0$ in the jth column of $\mathbf{B}$ forms an edge (w_l, v_j) between the second and third layers.

We perform different operations on the layered graph depending on the way we formulate the multiplication. In all cases though, the goal is to find pairs of vertices (u_i, v_j) sharing an adjacent vertex $w_k \in W$, and if any pair shares multiple adjacent vertices, to merge their contributions.

Using inner products, we analyze each pair (u_i, v_j) to find the set of vertices in $\widetilde{W_{ij}} \subseteq W = \{w_1, w_2, ..., w_k\}$ that are connected to both u_i amd v_j in the graph shown in Figure 3.1. The algorithm then accumulates contributions $a_{il} \cdot b_{lj}$ for all $w_l \in \widetilde{W_{ij}}$. The result becomes the value of $\mathbf{C}(i, j)$ in the output. In general this inner-product subgraph is sparse, and a contribution from w_l happens only when both edges a_{il} and b_{lj} exist. However, this sparsity is not exploited using inner products as it needs to examine each (u_i, v_j) pair, even when the set $\widetilde{W_{ij}}$ is empty.

In the outer-product formulation, the product is written as the summation of k rank one matrices:

$$\mathbf{C} = \sum_{l=1}^{k} \mathbf{A}(:, l)\mathbf{B}(l, :). \tag{3.2}$$

A different subgraph results from this formulation as it is the set of vertices W that represent the shared dimension that play the central role. Note that the edges are traversed in the outward direction from a node $w_i \in W$, as shown in Figure 3.2. For sufficiently sparse matrices, this formulation may run faster

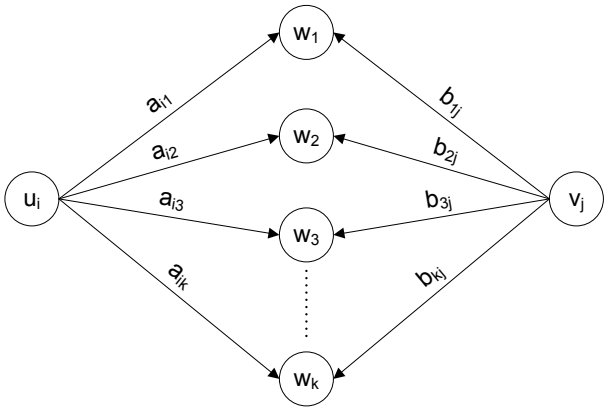


Figure 3.1: Graph representation of the inner product $\mathbf{A}(i,:) \cdot \mathbf{B}(:,j)$

because this traversal is performed only for the vertices in W (size k) instead of the inner product traversal that had to be performed for every pair (size mn). The problem with outer-product traversal is that it is hard to accumulate the intermediate results into the final matrix.

A row-by-row formulation of matrix multiplication performs a traversal starting from each of the vertices in U towards V, as shown in Figure 3.3 for u_i. Each traversal is independent from each other because they generate different rows of $\mathbf{C}$. Finally, a column-by-column formulation creates an isomorphic traversal, in the reverse direction (from V to U).

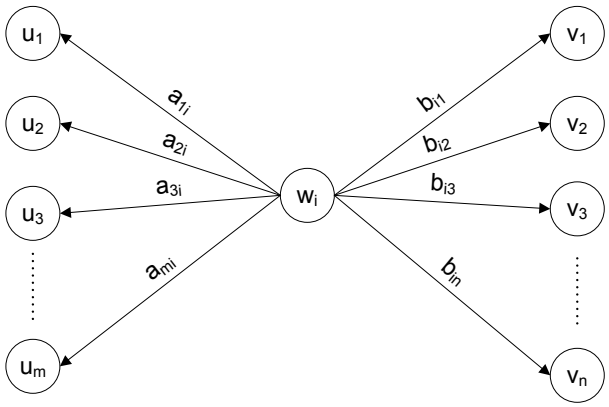


Figure 3.2: Graph representation of the outer product $\mathbf{A}(:, i) \cdot \mathbf{B}(i, :)$

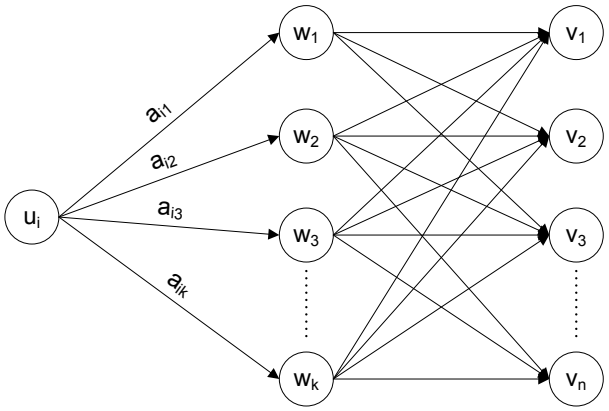


Figure 3.3: Graph representation of the sparse row times matrix product $\mathbf{A}(i, :) \cdot \mathbf{B}$

3.2.1 Hypersparse Matrices

Recall that a matrix is hypersparse if $nnz < n$. Although CSR/CSC is a fairly efficient storage scheme for general sparse matrices having $nnz = \Omega(n)$, it is asymptotically suboptimal for hypersparse matrices. Hypersparse matrices are fairly rare in numerical linear algebra (indeed, a nonsingular square matrix must have $nnz \geq n$), but they occur frequently in computations on graphs, particularly in parallel.

Our main motivation for hypersparse matrices comes from parallel processing. Hypersparse matrices arise after the 2-dimensional block data decomposition of ordinary sparse matrices for parallel processing. Consider a sparse matrix with c nonzero elements in each column. After the 2D decomposition of the matrix, each processor locally owns a submatrix with dimensions $(n/\sqrt{p}) \times (n/\sqrt{p})$. Storing each of those submatrices in CSC format takes $O(n\sqrt{p} + nnz)$ space, whereas the amount of space needed to store the whole matrix in CSC format on a single processor is only $O(n + nnz)$. As the number of processors increases, the $n\sqrt{p}$ term dominates the nnz term.

Figure 3.4 shows that the average number of nonzeros in a single column of a submatrix, $nnz(j)$, goes to zero as p increases. Storing a graph using CSC is similar to using adjacency lists. The column-pointers array represents the vertices, and the row-indices array represents their adjacencies. In that sense, CSC is a

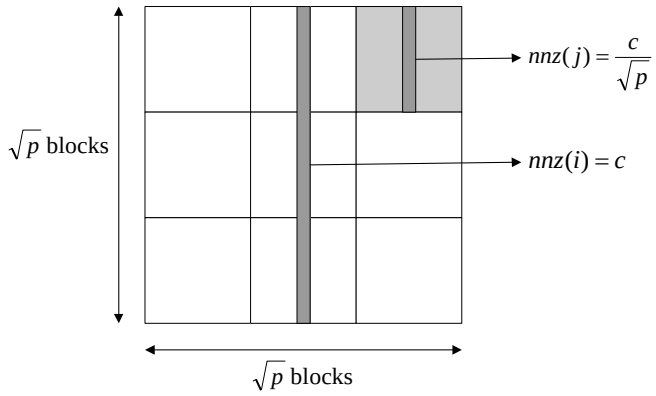


Figure 3.4: 2D Sparse Matrix Decomposition

vertex based data structure, making it suitable for 1D (vertex) partitioning of the graph. 2D partitioning, on the other hand, is based on edges. Therefore, using CSC with 2D distributed data is forcing a vertex based representation on edge distributed data. The result is unnecessary replication of column pointers (vertices) on each processor along the processor column.

The inefficiency of CSC leads to a more fundamental problem: any algorithm that uses CSC and scans all the columns is not scalable for hypersparse matrices. Even without any communication at all, such an algorithm cannot scale for $n\sqrt{p} \geq max\{\text{flops}, nnz\}$. SpMV and SpGEMM are algorithms that scan column indices. For these operations, any data structure that depends on the matrix dimension (such as CSR or CSC) is asymptotically too wasteful for submatrices.

JC	=	1	3	3	3	3	3	3	4	5	5
		↓						↓	↓		
IR	=	6	8					4	2		
NUM	=	0.1	0.2					0.3	0.4		

Figure 3.5: Matrix **A** in CSC format

DCSC Data Structure

We use a new data structure for our sequential hypersparse matrix-matrix multiplication. This structure, called ***DCSC*** for doubly compressed sparse columns, has the following properties:

1. It uses $O(nnz)$ storage.

2. It lets the hypersparse algorithm scale with increasing sparsity.

3. It supports fast access to columns of the matrix.

For an example, consider the 9-by-9 matrix with 4 non-zeros whose triples representation is given in Figure 3.6. Figure 3.5 showns its CSC storage, which includes repetitions and redundancies in the column pointers array (JC). Our new data structure compresses the JC array to avoid repetitions, giving the CP(column pointers) array of DCSC as shown in Figure 3.7. DCSC is essentially a sparse array of sparse columns, whereas CSC is a dense array of sparse columns.

A.I	**A**.J	**A**.V
6	1	0.1
8	1	0.2
4	7	0.3
2	8	0.4

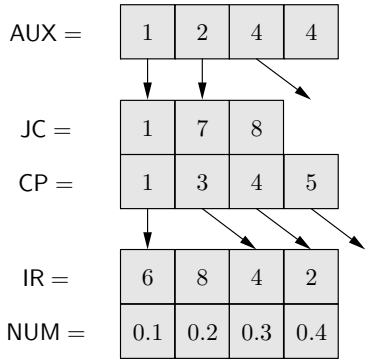


Figure 3.6: Matrix **A** in Triples format Figure 3.7: Matrix **A** in DCSC format

After removing repetitions, CP$[i]$ does no longer refer to the ith column. A new JC array, which is parallel to CP, gives us the column numbers. Although our HYPERSPARSE_GEMM algorithm does not need column indexing, DCSC supports fast column indexing for completeness. Whenever column indexing is needed, we construct an AUX array that contains pointers to nonzero columns (columns that have at least one nonzero element). Each entry in AUX refers to a $\lceil n/\, nzc \rceil$-sized chunk of columns, pointing to the first nonzero column in that chunk (there might be none). The storage requirement of DCSC is $O(nnz)$ since $|\mathsf{NUM}| = |\mathsf{IR}| = nnz$, $|\mathsf{JC}| = nzc$, $|\mathsf{CP}| = nzc + 1$, and $|\mathsf{AUX}| \approx nzc$.

In our implementation, the AUX array is a temporary work array that is constructed on demand, only when an operation requires repetitive use of it. This

keeps the storage and copying costs low. The time to construct AUX is only $O(nzc)$, which is subsumed by the cost of multiplication.

The careful reader will see that DCSC (without the AUX array) resembles two other data structures. One is the column-major ordered triples representation from Chapter 2, except we pack all the indices with the same column index into a single value JC[i], at the cost of introducing a integer pointer CP[i]. DCSC is also similar in spirit to 2D search trees [99], because we can perform column indexing using retrieval by primary key, and nonzero indexing within a column using retrieval by secondary key. This similarity suggests different implementations of DCSC, including lexicographical splay trees [185] and k-d trees [29]. Our particular representation is more cache-friendly than conventional 2D search trees because it uses arrays instead of pointers.

A Sequential Algorithm to Multiply Hypersparse Matrices

The sequential hypersparse algorithm (HYPERSPARSE_GEMM) is based on outer product multiplication. Therefore, it requires fast access to rows of matrix **B**. This could be accomplished by having each input matrix represented in DCSC and also in DCSR (doubly compressed sparse rows), which is the same as the transpose in DCSC. This method, which we described in an early version of this work [49], doubles the storage but does not change the asymptotic space and time

$$\mathbf{A} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} \times & & & \times & & \\ \times & \times & & & & \\ & & \times & \times & & \times \\ \times & & & \times & & \\ & \times & & & & \\ & & \times & & & \end{pmatrix} \end{array}, \mathbf{B} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \begin{pmatrix} \times & & & \times & & \\ & & & & & \\ & & \times & & \times & \\ \times & & & \times & & \\ & \times & & & \times & \times \\ & & \times & & \times & \times \end{pmatrix} \end{array}$$

Figure 3.8: Nonzero structures of operands $\mathbf{A}$ and $\mathbf{B}$

complexities. Here, we describe a more practical version where $\mathbf{B}$ is transposed as a preprocessing step, at a cost of trans($\mathbf{B}$). The actual cost of transposition is either $O(n + nnz(\mathbf{B}))$ or $O(nnz(\mathbf{B}) \lg nnz(\mathbf{B}))$, depending on the implementation.

The idea behind the Hypersparse_GEMM algorithm is to use the outer product formulation of matrix multiplication efficiently. The first observation about DCSC is that the JC array is already sorted. Therefore, $\mathbf{A}$.JC is the sorted indices of the columns that contain at least one nonzero and similarly $\mathbf{B^T}$.JC is the sorted indices of the rows that contain at least one nonzero. In this formulation, the ith column of $\mathbf{A}$ and the ith row of $\mathbf{B}$ are multiplied to form a rank-1 matrix. The naive algorithm does the same procedure for all values of i and gets n different rank-1 matrices, adding them to the resulting matrix $\mathbf{C}$ as they become available. Our algorithm has a preprocessing step that finds intersection Isect = $\mathbf{A}$.JC $\cap$ $\mathbf{B^T}$.JC, which is the set of indices that participate nontrivially in the outer product.

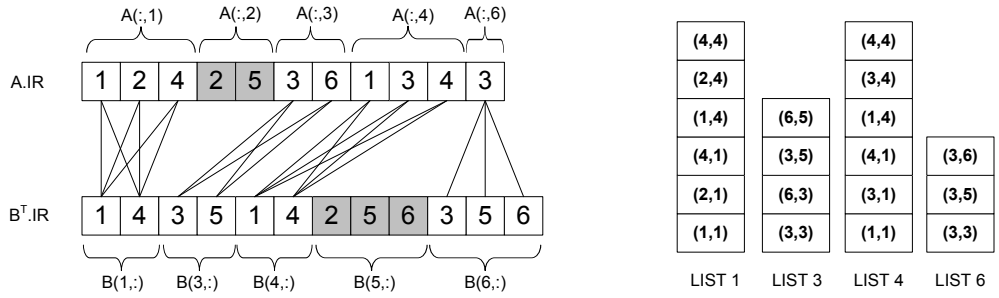


Figure 3.9: Cartesian product and the multiway merging analogy

The preprocessing takes $O(nzc(\mathbf{A}) + nzr(\mathbf{B}))$ time as $|\mathbf{A}.\mathsf{JC}| = nzc(\mathbf{A})$ and $|\mathbf{B^T}.\mathsf{JC}| = nzr(\mathbf{B})$. The next phase of our algorithm performs $|\mathsf{Isect}|$ cartesian products, each of which generates a fictitious list of size $nnz(\mathbf{A}(:,i)) \cdot nnz(\mathbf{B}(i,:))$. The lists can be generated sorted, because all the elements within a given column are sorted according to their row indices (i.e. $\mathsf{IR}(\mathsf{JC}(i))...\mathsf{IR}(\mathsf{JC}(i)+1)$ is a sorted range). The algorithm merges those sorted lists, summing up the intermediate entries having the same (row_id, col_id) index pair, to form the resulting matrix $\mathbf{C}$. Therefore, the second phase of HYPERSPARSE_GEMM is similar to multiway merging [128]. The only difference is that we never explicitly construct the lists; we compute their elements one-by-one on demand.

Figure 3.9 shows the setup for the matrices from Figure 3.8. As $\mathbf{A}.\mathsf{JC} = \{1,2,3,4,6\}$ and $\mathbf{B^T}.\mathsf{JC} = \{1,3,4,5,6\}$, $\mathsf{Isect} = \{1,3,4,6\}$ for this product. The algorithm does not touch the shaded elements, since they do not contribute to the output.

The merge uses a priority queue (represented as a heap) of size ni, which is the size of Isect, the number of indices i for which $\mathbf{A}(:, i) \neq \emptyset$ and $\mathbf{B}(i, :) \neq \emptyset$. The value in a heap entry is its NUM value and the key is a pair of indices (i, j) in column-major order. The idea is to repeatedly extract the entry with minimum key from the heap and insert another element from the list that the extracted element originally came from. If there are multiple elements in the lists with the same key, then their values are added on the fly. If we were to explicitly create ni lists instead of doing the computation on the fly, we would get the lists shown in the right side of Figure 3.9, which are sorted from bottom to top. For further details of multiway merging, consult Knuth [128].

The time complexity of this phase is $O(\text{flops} \cdot \lg ni)$, and the space complexity is $O(nnz(\mathbf{C}) + ni)$. The output is a stack of NUM values in column-major order. The $nnz(\mathbf{C})$ term in the space complexity comes from the output, and the flops term in the time complexity comes from the observation that

$$\sum_{i \in \mathsf{Isect}} nnz(\mathbf{A}(:, i)) \cdot nnz(\mathbf{B}(i, :)) = \text{flops}.$$

The final phase of the algorithm constructs the DCSC structure from this column-major-ordered stack. This requires $O(nnz(\mathbf{C}))$ time and space.

The overall time complexity of our algorithm is $O(nzc(\mathbf{A}) + nzr(\mathbf{B}) + \text{flops} \cdot \lg ni)$, plus the preprocessing time to transpose matrix $\mathbf{B}$. Note that $nnz(\mathbf{C})$ does not appear in this bound, since $nnz(\mathbf{C}) \leq \text{flops}$. We opt to keep the cost

```
C : ℝ^{S(m×n)} = HYPERSPARSE_GEMM(A : ℝ^{S(n×k)}, B^T : ℝ^{S(n×k)})
 1  lsect ← INTERSECTION(A.JC, B^T.JC)
 2  for j ← 1 to |lsect|
 3        do CARTMULT-INSERT(A, B^T, PQ, lsect, j)
 4           INCREMENT-LIST(lsect, j)
 5  while ISNOTFINISHED(lsect)
 6        do (key, value) ← EXTRACT-MIN(PQ)
 7           (product, i) ← UNPAIR(value)
 8           if key ≠ TOP(Q)
 9              then ENQUEUE(Q, key, product)
10              else  UPDATETOP(Q, product)
11           if ISNOTEMPTY(lsect(i))
12              then CARTMULT-INSERT(A, B^T, PQ, lists, lsect, i)
13                   INCREMENT-LIST(lsect, i)
14  CONSTRUCT-DCSC(Q)
```

Figure 3.10: Pseudocode for hypersparse matrix-matrix multiplication algorithm

of transposition separate, because our parallel 2D block SpGEMM will amortize this transposition of each block over $\sqrt{p}$ uses of that block. Therefore, the cost of transposition will be negligible in practice. The space complexity is $O(nnz(\mathbf{A}) + nnz(\mathbf{B}) + nnz(\mathbf{C}))$. The time complexity does not depend on n, and the space complexity does not depend on flops.

Figure 3.10 gives the pseudocode for the whole algorithm. It uses two subprocedures: CARTMULT-INSERT generates the next element from the ith fictitious list and inserts it to the heap PQ, and INCREMENT-LIST increments the pointers of the ith fictitious list or deletes the list from the intersection set if it is empty.

To justify the extra logarithmic factor in the flops term, we briefly analyze the complexity of each submatrix multiplication in the parallel 2D block SpGEMM. Our parallel 2D block SpGEMM performs $p\sqrt{p}$ submatrix multiplications, since each submatrix of the output is computed using $\mathbf{C}_{ij} = \sum_{k=1}^{\sqrt{p}} \mathbf{A}_{ik}\,\mathbf{B}_{kj}$. Therefore, with increasing number of processors and under perfect load balance, flops scale with $1/p\sqrt{p}$, nnz scale with $1/p$, and n scales with $1/\sqrt{p}$. Figure 3.11 shows the trends of these three complexity measures as p increases. The graph shows that the n term becomes the bottleneck after around 50 processors and flops becomes the lower-order term. In contrast to the classical algorithm, our Hypersparse_GEMM algorithm becomes independent of n, by putting the burden on the flops instead.

3.2.2 Sparse Matrices with Large Dimension

Even for ordinary large sparse matrices ($nnz = O(n)$), the use of SPA in the classical algorithm can hurt the performance. As shown in the previous chapter, the classical algorithm has an I/O complexity of $O(scan(\mathbf{A}) + \text{flops})$, because the fast memory (cache) is often not big enough to hold a $O(n)$ data structure like SPA. In this section, we give a cache-friendly formulation of the column-by-column algorithm that uses a heap instead of a SPA.

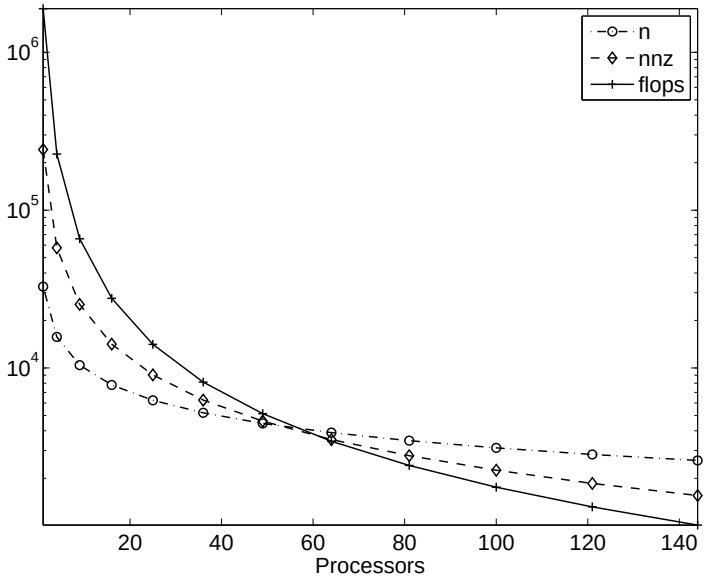


Figure 3.11: Trends of different complexity measures for submatrix multiplications as p increases. The inputs are randomly permuted RMAT matrices (scale 15 with an average of 8 nonzeros per column) that are successively divided into $(n/\sqrt{p}) \times (n/\sqrt{p})$. The counts are averaged over all submatrix multiplications.

A Cache Efficient Sequential Algorithm

Our second algorithm is based on the column-by-column formulation the classical algorithm. Any data structure that can enumerate nonzeros in the jth column in $O(nnz(j))$ time is suitable for this algorithm, so we assume that the matrices are in CSC format for simplicity. Equivalently, we could have used DCSC to avoid any format conversion in case this subroutine is used as part of a polyalgorithm. Similar to the classical algorithm, we will be computing a whole column of $\mathbf{C}$ in one step by examining the same column of $\mathbf{B}$. In other words, $\mathbf{C}(:, j)$ is a linear combination of the columns $\mathbf{A}(:, i)$ for which $\mathbf{B}(i, j) \neq 0$. Time complexity, however, is independent of m since we do not use a SPA.

For the construction of $\mathbf{C}(:, j)$, we use a heap of size $nnz(\mathbf{B}(:, j))$. As in the case of the hypersparse algorithm, we require the row indices within each column to be sorted. The idea of merging columns using a heap has been employed before, within the Ordered-SPA data structure [124]. However, the Ordered-SPA was never used to suppress the m factor in the algorithm because it is an $\Theta(m)$ data structure. Furthermore, the Ordered-SPA uses a heap of size $nnz(\mathbf{C}(:, j))$, which can be much bigger than $nnz(\mathbf{B}(:, j))$.

Figure 3.12 illustrates the algorithm where column j of $\mathbf{C}$ is computed as a linear combination of the columns of $\mathbf{A}$ as specified by the nonzeros in column j

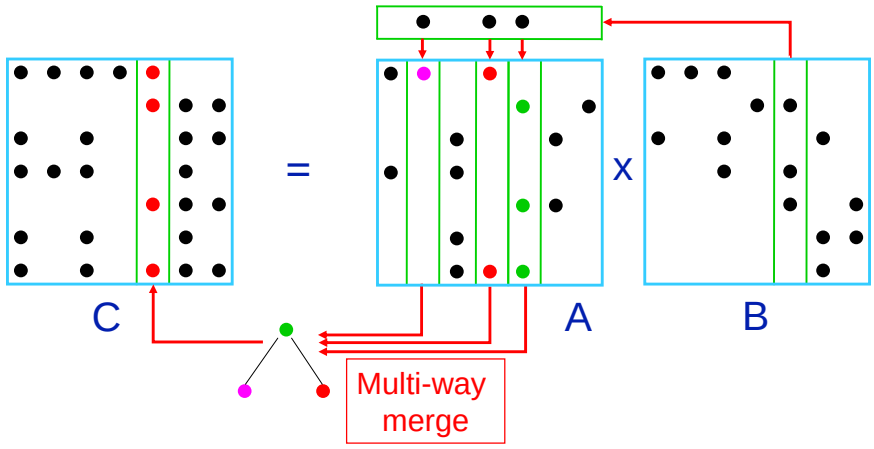


Figure 3.12: Cache efficient multiplication of sparse matrices stored by columns. Columns of **A** are merged as specified by the non-zero entries in a column of **B** using a heap that is ordered by row indices of nonzero elements. The contents of the heap are stored in a column of **C** once all required columns are merged.

of **B**. Let us illustrate the execution of the algorithm through the example inputs **A** and **B** shown in 3.3.

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 3 & 0 & 4 \\ 6 & 0 & 0 & 0 \\ 0 & 5 & 5 & 5 \end{pmatrix}, \mathbf{B} = \begin{pmatrix} 7 & 0 & 2 & 0 \\ 3 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 0 & 1 \end{pmatrix} \tag{3.3}$$

The first column of **B** gives the set of column indices of **A** that is required during the construction of the first column of **C** (in this case they are the 1st and 2nd columns of **A**). We can now do a multiway merge, with a heap of size $nnz(\mathbf{B}(:,1)) = 2$ as follows: Initially, the heap contains $(key, value) = (2, 7 \cdot 6)$

and $(key, value) = (1, 3 \cdot 3)$. We repeatedly extract the entry with minimum key and insert the next element from the column that the extracted element originally came from. The row index alone is sufficient as the key because we construct one column at a time and all the elements in that column has the same column index. The high level pseudocode for the algorithm is given in Figure 3.14, which utilizes a subroutine given in Figure 3.13.

```
MULT-INSERT(PQ, lists, i, bval)
1  product = lists(i). value · bval
2  value ← PAIR(product, i)
3  key ← lists(i). index
4  INSERT(PQ, key, value)
5  ADVANCE(lists(i))
```

Figure 3.13: Subroutine to multiply *bval* with the next element from the ith list and insert it to the priority queue

The time complexity of the algorithm is

$$\sum_{j=1}^{n} \text{flops}(\mathbf{C}(:, j)) \lg nnz(\mathbf{B}(:, j)),$$

where $\text{flops}(\mathbf{C}(:, j))$ is the number of nonzero multiplications required to generate the jth column of $\mathbf{C}$.

When the inputs are matrices from Erdős-Rényi graphs, the average number of nonzeros in any column is constant. Furthermore, permutation matrices and matrices representing regular grids, have a fixed number of nonzeros in each column.

```
C : R^{S(m×n)} = CscHeap_SpGEMM(A : R^{S(m×k)}, B : R^{S(k×n)})
 1  for j ← 1 to n
 2       do for k where B(k, j) ≠ 0
 3            do lists(k) ← SparseList(A(:, k))
 4               Mult-Insert(PQ, lists, k, B(k, j))
 5          while IsNotFinished(lists)
 6            do (key, value) ← Extract-Min(PQ)
 7               (product, i) ← UnPair(value)
 8               if key ≠ Top(Q)
 9                  then Enqueue(Q, key, product)
10                  else UpdateTop(Q, product)
11               if IsNotEmpty(lists(k))
12                  then Mult-Insert(PQ, lists, k, B(k, j))
13                  else Delete(lists, k)
14          C(:, k) ← Output(Q)
15          Reset(Q)
```

Figure 3.14: Pseudocode for heap assisted column-by-column algorithm

Let the average number of nonzeros in any column of $\mathbf{B}$ be c. Then, for those families of matrices, the expected cost of the CscHeap_SpGEMM algorithm is

$$\sum_{j=1}^{n} \text{flops}(\mathbf{C}(:, j)) \cdot \lg c = O(n + \text{flops} \cdot \lg c) = O(n + \text{flops}).$$

The CscHeap_SpGEMM algorithm differs from the classical algorithm only in its choice of the data structure that is used to construct columns of $\mathbf{C}$. The right choice depends on the inputs and is often an algorithm engineering decision. The classical algorithm is more suitable for fairly dense matrices, or matrices having dimensions that are small enough so that SPA fits into the cache. When the matrix dimensions are bigger than the cache size and the matrices are sufficiently sparse

so that the added $\lg c$ factor is negligible, we expect CscHeap_SpGEMM to outperform the classical algorithm. We verify our intuition through experiments in Section 3.2.3.

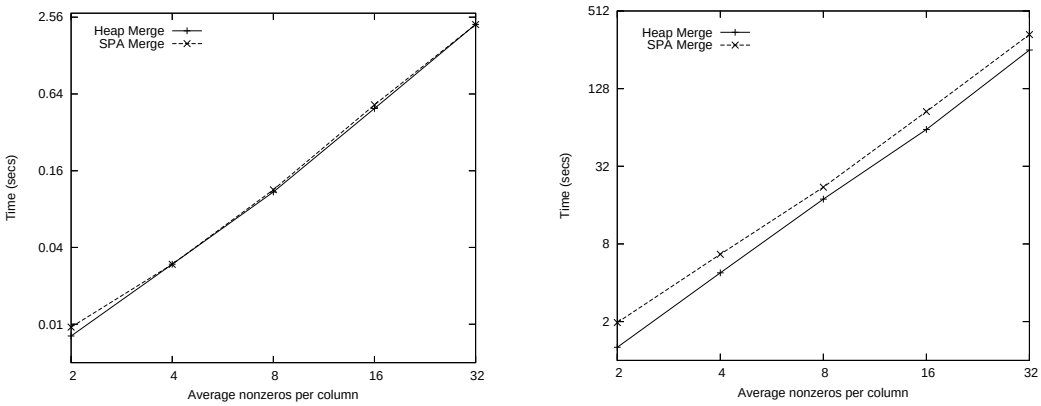
3.2.3 Performance of the Cache Efficient Algorithm

In order to unveil the effects of cache misses on the column-by-column SpGEMM algorithms, we ran experiments with matrix dimensions varying from 10^3 to 10^6. For matrix dimension, we timed the multiplication of two sparse matrices from Erdős-Rényi graphs with average nonzeros per column varying from 2 to 32.

Our test platform is a 2.2 Ghz Opteron that has 512 KB L2 cache per core. Therefore, SPA starts to fall out of L2 cache around $n = 10^5$. We compared two column-wise algorithms: the classical algorithm that uses a SPA and the cache-friendly algorithm explained in Section 3.2.2 that uses a heap. We implemented both algorithms using C++ and compiled them with the Intel C++ compiler version 9.1, as it is the best performing compiler on our system, with the optimization flag `-fast` .

The results for $n = 10^4$ show the case where SPA completely fits into the cache. In this case, as shown in Figure 3.15(a), both algorithms are comparable in terms of performance, with only a marginal difference of at most $\pm 5\%$. On the other hand, Figure 3.15(a) shows the results for 10^6 where SPA falls out of cache.

The advantage of using a heap is more pronounced in this case, with relative performance increases varying from 24% to 55% over the SPA based version.



(a) $n = 10^4$ (b) $n = 10^6$

Figure 3.15: Performance of two column-wise algorithms for multiplying two $n \times n$ sparse matrices from Erdős-Rényi random graphs

3.3 Parallel Algorithms for Sparse GEMM

This section describes parallel algorithms for multiplying two sparse matrices in parallel on p processors, which we call PSpGEMM. The design of our algorithms is motivated by distributed memory systems, but expect them to perform well in shared memory too, as they avoid hot spots and load imbalances by ensuring proper work distribution among processors. Like most message passing algorithms, they can be implemented in the partitioned global address space (PGAS) model as well.

3.3.1 1D Decomposition

We assume the data is distributed to processors in block rows, where each processor receives m/p consecutive rows. We write $\mathbf{A}_i = \mathbf{A}(ip : (i+1)p - 1, :)$ to denote the block row owned by the ith processor. To simplify the algorithm description, we use $\mathbf{A}_{ij}$ to denote $\mathbf{A}_i(:, jp : (j+1)p - 1)$, the jth block column of $\mathbf{A}_i$, although block rows are not physically partitioned.

$$\mathbf{A} = \left(\begin{array}{c} \mathbf{A}_1 \\ \hline \vdots \\ \hline \mathbf{A}_p \end{array}\right) = \left(\begin{array}{c|c|c} \mathbf{A}_{11} & \dots & \mathbf{A}_{1p} \\ \hline \vdots & \ddots & \vdots \\ \hline \mathbf{A}_{p1} & \dots & \mathbf{A}_{pp} \end{array}\right), \mathbf{B} = \left(\begin{array}{c} \mathbf{B}_1 \\ \hline \vdots \\ \hline \mathbf{B}_p \end{array}\right) \tag{3.4}$$

For each processor $P(i)$, the computation is:

$$\mathbf{C}_i = \mathbf{C}_i + \mathbf{A}_i \mathbf{B} = \mathbf{C}_i = \mathbf{C}_i + \sum_{j=1}^{p} \mathbf{A}_{ij} \mathbf{B}_j$$

3.3.2 2D Decomposition

Our 2D parallel algorithms, Sparse Cannon and Sparse SUMMA, use the hypersparse algorithm, which has complexity $O(nzc(\mathbf{A}) + nzr(\mathbf{B}) + \text{flops} \cdot \lg ni)$, as shown in Section 3.2.1, for multiplying submatrices. Processors are logically organized on a square $\sqrt{p} \times \sqrt{p}$ mesh, indexed by their row and column indices so that the (i, j)th processor is denoted by $P(i, j)$. Matrices are assigned to processors according to a 2D block decomposition. Each node gets a submatrix of dimensions $(n/\sqrt{p}) \times (n/\sqrt{p})$ in its local memory. For example, $\mathbf{A}$ is partitioned as shown below and $\mathbf{A}_{ij}$ is assigned to processor $P(i, j)$.

$$\mathbf{A} = \left(\begin{array}{c|c|c} \mathbf{A}_{11} & \cdots & \mathbf{A}_{1\sqrt{p}} \\ \hline \vdots & \ddots & \vdots \\ \hline \mathbf{A}_{\sqrt{p}1} & \cdots & \mathbf{A}_{\sqrt{p}\sqrt{p}} \end{array}\right) \tag{3.5}$$

For each processor $P(i)$, the computation is:

$$\mathbf{C}_{ij} = \sum_{k=1}^{\sqrt{p}} \mathbf{A}_{ik}\, \mathbf{B}_{kj}$$

3.3.3 Sparse 1D Algorithm

The row-wise SpGEMM forms one row of $\mathbf{C}$ at a time, and each processor may potentially need to access all of $\mathbf{B}$ to form a single row of $\mathbf{C}$. However, only a portion of $\mathbf{B}$ is locally available at any time in parallel algorithms. The algorithm, thus, performs multiple iterations to fully form one row of $\mathbf{C}$. We use

```
C : ℝ^{P(S(n)×n)} = Block1D_PSpGEMM(A : ℝ^{P(S(n)×n)}, B : ℝ^{P(S(n)×n)})
1  for all processors P(i) in parallel
2       do Initialize(SPA)
3          for j ← 1 to p
4               do Broadcast(B_j)
5                  for k ← 1 to n/p
6                       do Load(SPA, C_i(k,:))
7                          SPA ← SPA + A_ij(k,:) B_j
8                          Unload(SPA, C_i(k,:))
```

Figure 3.16: Operation $\mathbf{C} \leftarrow \mathbf{AB}$ using block row Sparse 1D algorithm

a SPA to accumulate the nonzeros of the current active row of $\mathbf{C}$. Figure 3.16 shows the pseudocode of the algorithm. Loads and unloads of SPA, which is not amortized by the number of nonzero arithmetic operations in general, dominate the computational time.

3.3.4 Sparse Cannon

Our first 2D algorithm is based on Cannon's algorithm for dense matrices [52]. The pseudocode of the algorithm is given in Figure 3.19. Sparse Cannon, although elegant, is not our choice of algorithm for the final implementation, as it is hard to generalize to non-square grids, non-square matrices, and matrices whose dimensions are not perfectly divisible by grid dimensions.

```
LEFT-CIRCULAR-SHIFT(Local : ℝ^{S(n×n)}, s)
1  SEND(Local, P(i, (j − s) mod √p))          ▷ This is processor P(i, j)
2  RECEIVE(Temp, P(i, (j + s) mod √p))
3  Local ← Temp
```

Figure 3.17: Circularly shift left by s along the processor row

```
UP-CIRCULAR-SHIFT(Local : ℝ^{S(n×n)}, s)
1  SEND(Local, P((i − s) mod √p, j))          ▷ This is processor P(i, j)
2  RECEIVE(Temp, P((i + s) mod √p, j))
3  Local ← Temp
```

Figure 3.18: Circularly shift up by s along the processor column

```
C : ℝ^{P(S(n×n))} = CANNON_PSPGEMM(A : ℝ^{P(S(n×n))}, B : ℝ^{P(S(n×n))})
1  for all processors P(i, j) in parallel
2       do LEFT-CIRCULAR-SHIFT(A_ij, i − 1)
3          UP-CIRCULAR-SHIFT(B_ij, j − 1)
4  for all processors P(i, j) in parallel
5       do for k ← 1 to √p
6             do C_ij ← C_ij + A_ij B_ij
7                LEFT-CIRCULAR-SHIFT(A_ij, 1)
8                UP-CIRCULAR-SHIFT(B_ij, 1)
```

Figure 3.19: Operation $\mathbf{C} \leftarrow \mathbf{AB}$ using Sparse Cannon

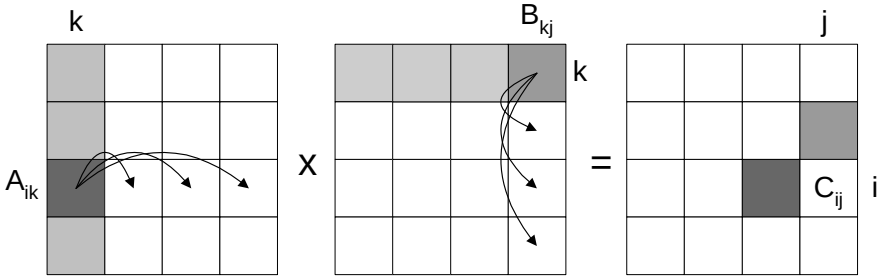


Figure 3.20: Sparse SUMMA Execution ($b = n/\sqrt{p}$)

3.3.5 Sparse SUMMA

SUMMA [100] is a memory efficient, easy to generalize algorithm for parallel dense matrix multiplication. It is the algorithm used in parallel BLAS [61]. As opposed to Cannon's algorithm, it allows a tradeoff to be made between latency cost and memory by varying the degree of blocking. The algorithm, illustrated in Figure 3.20, proceeds in k/b stages. At each stage, $\sqrt{p}$ active row processors broadcast b columns of $\mathbf{A}$ simultaneously along their rows and $\sqrt{p}$ active column processors broadcast b rows of $\mathbf{B}$ simultaneously along their columns.

Sparse SUMMA is our algorithm of choice for our final implementation, because it is easy to generalize to non-square matrices, matrices whose dimensions are not perfectly divisible by grid dimensions.

3.4 Analysis of Parallel Algorithms

In this section, we analyze the parallel performance of our algorithms, and show that they scale better than existing 1D algorithms in theory. We begin by introducing our parameters and model of computation. Then, we present a theoretical analysis showing that 1D decomposition, at least with the current algorithm, is not sufficient for PSpGEMM to scale. Finally, we analyze our 2D algorithms in depth.

In our analysis, the cost of one floating-point operation, along with the cost of cache misses and memory indirections associated with the operation, is denoted by γ, measured in nanoseconds. The latency of sending a message over the communication interconnect is α, and the inverse bandwidth is β, measured in nanoseconds and nanoseconds per word transfered, respectively. The running time of a parallel algorithm on p processors is given by

$$T_p = T_{comm} + T_{comp},$$

where T_{comm} denotes the time spent in communication and T_{comp} is the time spent during local computation phases. T_{comm} includes both the latency (delay) costs and the actual time it takes to transfer the data words over the network. Hence, the cost of transmitting h data words in a communication phase is

$$T_{comm} = \alpha + h\beta.$$

The sequential work of SpGEMM, unlike dense GEMM, depends on many parameters. This makes parallel scalability analysis a tough process. Therefore, we restrict our analysis to sparse matrices following the Erdős-Rényi graph model explained in Section 1.5.2. Consequently, the analysis is probabilistic, exploiting the independent and identical distribution of nonzeros. When we talk about quantities such as nonzeros per subcolumn, we mean the expected number of nonzeros. Our analysis assumes that there are $c > 0$ nonzeros per row/column. The sparsity parameter c, albeit oversimplifying, is useful for analysis purposes, since it makes different parameters comparable to each other. For example, if $\mathbf{A}$ and $\mathbf{B}$ both have sparsity c, then $nnz(\mathbf{A}) = cn$ and flops$(\mathbf{AB}) = c^2n$. It also allows us to decouple the effects of load imbalances from the algorithm analysis because the nonzeros are assumed to be evenly distributed across processors.

The lower bound on sequential SpGEMM is $\Omega(\text{flops}) = \Omega(c^2n)$. This bound is achieved by some row-wise and column-wise implementations [104, 113], provided that $c \geq 1$. The row-wise implementation of Gustavson that uses CSR is the natural kernel to be used in the 1D algorithm where data is distributed by rows. As shown in the previous chapter, it has an asymptotic complexity of

$$O(n + nnz(\mathbf{A}) + \text{flops}) = O(n + cn + c^2n) = \Theta(c^2n).$$

Therefore, we take the sequential work (W) to be $\gamma c^2 n$ in our analysis.

3.4.1 Scalability of the 1D Algorithm

We begin with a theoretical analysis whose conclusion is that 1D decomposition is not sufficient for PSpGEMM to scale. In Block1D_PSpGEMM, each processor sends and receives $p-1$ point-to-point messages of size $nnz(\mathbf{B})/p$. Therefore,

$$T_{comm} = (p-1)\left(\alpha + \beta \frac{nnz(\mathbf{B})}{p}\right) = \Theta(p\,\alpha + \beta\, c\, n). \tag{3.6}$$

We previously showed that the Block1D_PSpGEMM algorithm is unscalable with respect to both communication and computation costs [48]. In fact, it gets slower as the number of processors grow. The current Star-P implementation [180] by-passes this problem by all-to-all broadcasting nonzeros of the $\mathbf{B}$ matrix, so that the whole $\mathbf{B}$ matrix is essentially assembled at each processor. This avoids the cost of loading and unloading SPA at every stage, but it uses $nnz(\mathbf{B})$ memory at each processor.

3.4.2 Scalability of the 2D Algorithms

In this section, we provide an in-depth theoretical analysis of our parallel 2D SpGEMM algorithms, and conclude that they scale significantly better than their 1D counterparts. Although our analysis is limited to the Erdős-Rényi model, its conclusions are strong enough to be convincing.

In CANNON_PSPGEMM, each processor sends and receives $\sqrt{p}-1$ point-to-point messages of size $nnz(\mathbf{A})/p$, and $\sqrt{p}-1$ messages of size $nnz(\mathbf{B})/p$. Therefore, the communication cost per processor is

$$T_{comm} = \sqrt{p}\left(2\,\alpha + \beta\left(\frac{nnz(\mathbf{A}) + nnz(\mathbf{B})}{p}\right)\right) = \Theta(\alpha\,\sqrt{p} + \frac{\beta\,c\,n}{\sqrt{p}}). \tag{3.7}$$

The average number of nonzeros in a column of a local submatrix $\mathbf{A}_{ij}$ is $c/\sqrt{p}$. Therefore, for a submatrix multiplication $\mathbf{A}_{ik}\mathbf{B}_{kj}$,

$$ni(\mathbf{A}_{ik}, \mathbf{B}_{kj}) = \min\{1, \frac{c^2}{p}\}\frac{n}{\sqrt{p}} = \min\{\frac{n}{\sqrt{p}}, \frac{c^2\,n}{p\sqrt{p}}\},$$

$$\text{flops}(\mathbf{A}_{ik}\mathbf{B}_{kj}) = \frac{\text{flops}(\mathbf{AB})}{p\sqrt{p}} = \frac{c^2\,n}{p\sqrt{p}},$$

$$T_{mult} = \sqrt{p}\left(2\,\min\{1, \frac{c}{\sqrt{p}}\}\frac{n}{\sqrt{p}} + \frac{c^2 n}{p\sqrt{p}}\lg\Big(\min\{\frac{n}{\sqrt{p}}, \frac{c^2 n}{p\sqrt{p}}\}\Big)\right).$$

The probability of a single column of $\mathbf{A}_{ik}$ (or a single row of $\mathbf{B}_{kj}$) having at least one nonzero is $\min\{1, c/\sqrt{p}\}$ where 1 covers the case $p \leq c^2$ and $c/\sqrt{p}$ covers the case $p > c^2$.

The overall cost of additions, using p processors, and Brown and Tarjan's $O(m\ \lg n/m)$ algorithm [46] for merging two sorted lists of size m and n (for $m < n$), is

$$T_{add} = \sum_{i=1}^{\sqrt{p}} \Big(\frac{\text{flops}}{p\sqrt{p}} \ \lg i \Big) = \frac{\text{flops}}{p\sqrt{p}} \ \lg \prod_{i=1}^{\sqrt{p}} i = \frac{\text{flops}}{p\sqrt{p}} \ \lg (\sqrt{p}\,!).$$

Note that we might be slightly overestimating, since we assume flops/ $nnz(\mathbf{C}) \approx 1$ for simplicity. From Stirling's approximation and asymptotic analysis, we know that $\lg(n\,!) = \Theta(n \lg n)$ [68]. Thus, we get:

$$T_{add} = \Theta\Big(\frac{\text{flops}}{p\sqrt{p}} \ \sqrt{p} \ \lg \sqrt{p} \Big) = \Theta\Big(\frac{c^2 n \lg \sqrt{p}}{p} \Big).$$

There are two cases to analyze: $p > c^2$ and $p \leq c^2$. Since scalability analysis is concerned with the asymptotic behavior as p increases, we just provide results for the $p > c^2$ case. The total computation cost $T_{comp} = T_{mult} + T_{add}$ is

$$T_{comp} = \gamma \Big(\frac{c\,n}{\sqrt{p}} + \frac{c^2 n}{p} \lg \big(\frac{c^2 n}{p\sqrt{p}} \big) + \frac{c^2 n \lg \sqrt{p}}{p} \Big) = \gamma \Big(\frac{c\,n}{\sqrt{p}} + \frac{c^2 n}{p} \lg \big(\frac{c^2 n}{p} \big) \Big). \quad (3.8)$$

In this case, parallel efficiency is

$$E = \frac{W}{p\,\big(T_{comp} + T_{comm}\big)} = \frac{\gamma c^2 n}{(\gamma + \beta)\,c\,n\,\sqrt{p} + \gamma c^2 n \ \lg\big(\frac{c^2 n}{p}\big) + \alpha\, p\,\sqrt{p}}. \quad (3.9)$$

Scalability is not perfect and efficiency deteriorates as p increases due to the first term. Speedup is, however, not bounded, as opposed to the 1D case. In particular, $\lg(c^2 n/p)$ becomes negligible as p increases and scalability due to latency is achieved when $\gamma c^2 n \propto \alpha\, p\sqrt{p}$, where it is sufficient for n to grow on the order of $p^{1.5}$. The biggest bottleneck for scalability is the first term in the denominator,

which scales with $\sqrt{p}$. Consequently, two different scaling regimes are likely to be present: A close to linear scaling regime until the first term starts to dominate the denominator and a $\sqrt{p}$-scaling regime afterwards.

Compared to the 1D algorithms, Sparse Cannon both lower the degree of unscalability due to bandwidth costs and mitigate the bottleneck of computation. This makes overlapping communication with computation more promising.

Sparse SUMMA, like dense SUMMA, incurs an extra cost over Cannon for using row-wise and col-wise broadcasts instead of nearest-neighbor communication, which might be modeled as an additional $O(\lg p)$ factor in communication cost. Other than that, the analysis is similar to sparse Cannon and we omit the details. Using the DCSC data structure, the expected cost of fetching b consecutive columns of a matrix $\mathbf{A}$ is b plus the size (number of nonzeros) of the output [49]. Therefore, the algorithm asymptotically has the same computation cost for all values of b.

3.5 Performance Modeling of Parallel Algorithms

In this section, we first project the estimated speedup of 1D and 2D algorithms in order to evaluate their prospects in practice. We use a quasi-analytical performance model where we first obtain realistic values for the parameters (γ,

β, α) of the algorithm performance, then use them in our projections. In the second part, we perform another modeling study where we simulate the execution of Sparse SUMMA using an actual implementation of the Hypersparse_GEMM algorithm. This modeling study concludes that Hypersparse_GEMM is scalable with increasing hypersparsity, suggesting that it is a suitable algorithm to be the sequential kernel of a 2D parallel SpGEMM.

3.5.1 Estimated Speedup of Parallel Algorithms

This study estimates the speedup of 1D and 2D algorithms by using a quasi-analytic model that projects the performance on large systems using realistic values for the performance parameters.

In order to obtain a realistic value for γ, we performed multiple runs on an AMD Opteron 8214 (Santa Rosa) processor using matrices of various dimensions and sparsity; estimating the constants using non-linear regression. One surprising result is the order of magnitude difference in the constants between sequential kernels. The classical algorithm, which is used as the 1D SpGEMM kernel, has $\gamma = 293.6$ nsec, whereas Hypersparse_GEMM, which is used as the 2D kernel, has $\gamma = 19.2$ nsec. We attribute the difference to cache friendliness of the hypersparse algorithm. The interconnect supports $\beta = 1$ GB/sec point-to-point bandwidth, and a maximum of $\alpha = 2.3$ microseconds latency, both of which are

achievable on TACC's Ranger Cluster. The communication parameters ignore network contention.

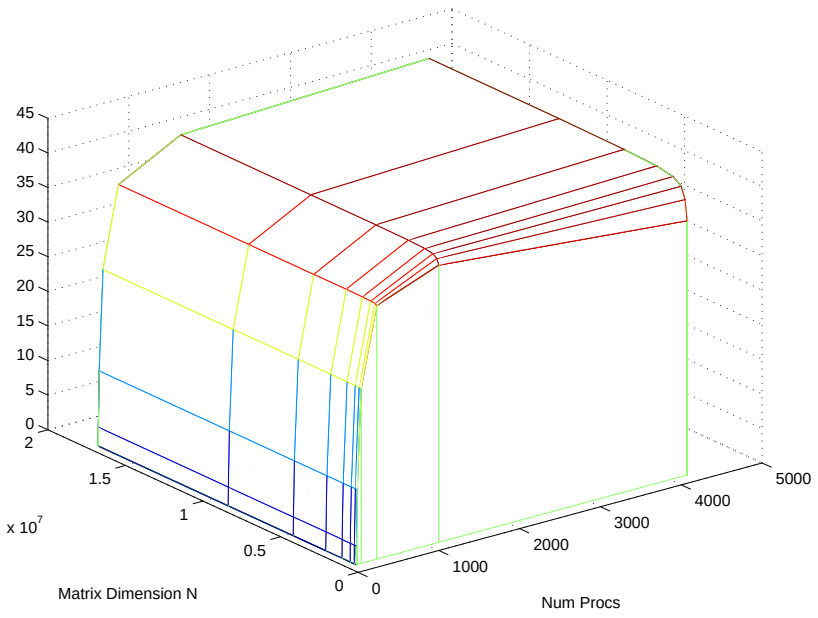


Figure 3.21: Modeled speedup of Synchronous Sparse 1D algorithm

Figures 3.21 and 3.22 show the modeled speedup of Block1D_PSpGEMM and Cannon_PSpGEMM for matrix dimensions from $n = 2^{17}$ to 2^{24} and number of processors from $p = 1$ to 4096. The inputs are Erdős-Rényi graphs.

We see that Block1D_PSpGEMM's speedup does not go beyond 50x, even on larger matrices. For relatively small matrices, having dimensions $n = 2^{17} - 2^{20}$, it starts slowing down after a thousand processors, where it achieves less than 40x speedup. On the other hand, Cannon_PSpGEMM shows increasing and almost linear speedup for up to 4096 processors, even though the slope of the curve is less

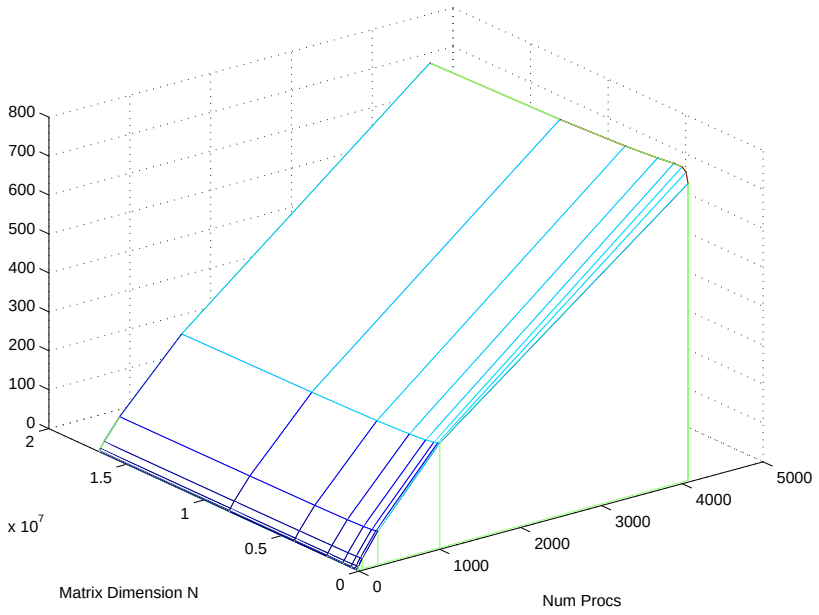


Figure 3.22: Modeled speedup of synchronous Sparse Cannon

than one. It is crucial to note that the projections for the 1D algorithm are based on the memory inefficient implementation that performs an all-to-all broadcast of **B**. This is because the original memory efficient algorithm given in Section 3.3.1 actually slows down as p increases.

It is worth explaining one peculiarity. The modeled speedup turns out to be higher for smaller matrices than for bigger matrices. Remember that communication requirements are on the same order as computational requirements for parallel SpGEMM. Intuitively, the speedup should be independent of the matrix dimension in the absence of load imbalance and network contention, but since we are estimating the speedup with respect to the optimal sequential algorithm, the

overheads associated with the hypersparse algorithm are bigger for larger matrices. The bigger the matrix dimension, the slower the hypersparse algorithm is with respect to the optimal algorithm, due to the extra logarithmic factor. Therefore, speedup is better for smaller matrices in theory. This is not the case in practice, because the peak bandwidth is usually not achieved for small sized data transfers and load imbalances are severer for smaller matrices. Section 3.7.1 addresses the load imbalance.

We also evaluate the effects of overlapping communication with computation. Following Krishnan and Nieplocha [133], we define the non-overlapped percentage of communication as:

$$w = 1 - \frac{T_{comp}}{T_{comm}} = \frac{T_{comm} - T_{comp}}{T_{comm}}$$

The speedup of the asynchronous implementation is:

$$S = \frac{W}{T_{comp} + w(T_{comm})}$$

Figure 3.23 shows the modeled speedup of asynchronous SpCannon assuming truly one-sided communication. For smaller matrices with dimensions $n = 2^{17} - 2^{20}$, speedup is about 25% more than the speedup of the synchronous implementation.

The modeled speedup plots should be interpreted as upper bounds on the speedup that can be achieved on a real system using these algorithms. Achieving

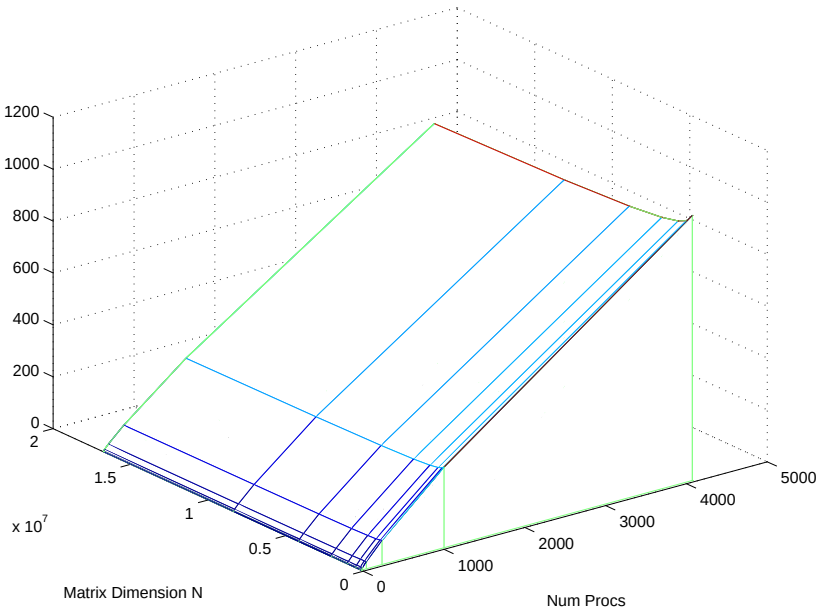


Figure 3.23: Modeled speedup of asynchronous Sparse Cannon

these speedups on real systems requires all components to be implemented and working optimally. The conclusion we derive from those plots is that no matter how hard we try, it is impossible to get good speedup with the current 1D algorithms.

3.5.2 Scalability with Hypersparsity

This modeling study reveals the scalability of our hypersparse algorithm with increasing sparsity. We have implemented our data structures and multiplication algorithms in C++. Our code is compiled using the GNU Compiler Collection (GCC) Version 4.1, with the flags `-O3`, because these are the settings

that our comparison platform, MATLAB, is compiled with. We have incorporated Peter Sander's Sequence Heaps [174] for all the priority queues used by our algorithms. Througout the experiments, the numerical values are represented as double-precision floating points.

We compare the performance of our implementation with MATLAB R2007A's (64-bit version) implementation of the classical algorithm. Sparse matrix multiplication is a built-in function in MATLAB, so there are no interpretation overheads associated with it. We are simply comparing our C++ code with the underlying precompiled C code used in MATLAB.

All of our experiments are performed on a single core of Opteron 2.2 Ghz with 64 GB main memory, where we simulate the execution of a parallel SpGEMM. The simulation is done by dividing the input matrices of size $n \times n$ into p submatrices of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ using the 2D block decomposition, as explained in Section 3.3.2 and shown in Figure 3.5.

Expressing the matrix multiplication as algebraic operations on submatrices instead of individual elements, we see that each submatrix of the product is computed using $\mathbf{C}_{ij} = \sum_{k=1}^{\sqrt{p}} \mathbf{A}_{ik} \mathbf{B}_{kj}$. Since we are primarily concerned with the sequential sparse matrix multiplication kernel, we will exclude the cost of submatrix additions and other parallel overheads. That is to say, we will only time the

submatrix multiplications, exactly plotting

$$time(p, \mathbf{A}, \mathbf{B}) = \sum_{i=1}^{\sqrt{p}} \sum_{j=1}^{\sqrt{p}} \sum_{k=1}^{\sqrt{p}} time(\mathbf{A}_{ik}\, \mathbf{B}_{kj}),$$

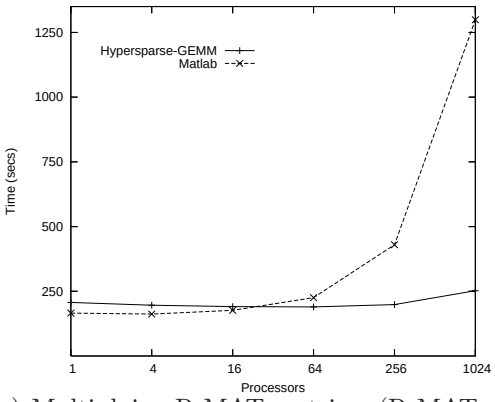
which is equal to the amount of work done by a parallel matrix multiplication algorithm such as SUMMA [100].

Increasing p in this case does not mean we use more processors to compute the product. Instead, it means we use smaller and smaller blocks while computing the product on a single processor. Therefore, a perfectly scalable algorithm would yield flat timing curves as p increases. We expect our hypersparse algorithm to outperform the classical algorithm as p increases due to reasons explained in Section 3.2.1. We label the classical algorithm MATLAB, and our algorithm HYPERSPARSE_GEMM in the plots. The second input is only transposed once because this is what would happen in a parallel implementation.

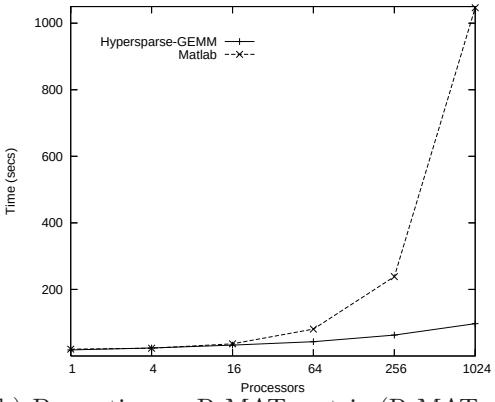
In all experiments in this section, the input matrices have dimensions $2^{23} \times 2^{23}$, i.e. the input graphs have around 8 million vertices.

Synthetic R-MAT Graphs

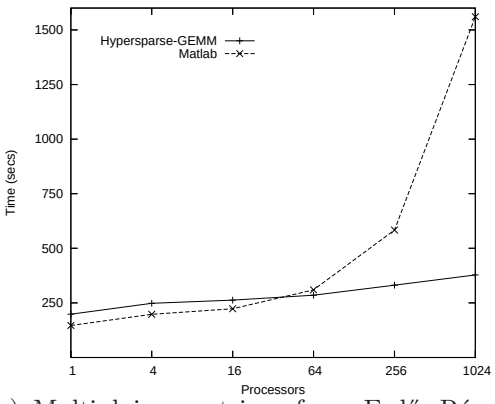
We ran two main sets of multiplication experiments with R-MAT matrices, one where both input matrices are R-MAT, and one where $\mathbf{A}$ is a R-MAT matrix and $\mathbf{B}$ is a permutation matrix. The results are shown in Figures 3.24(a) and 3.24(b).



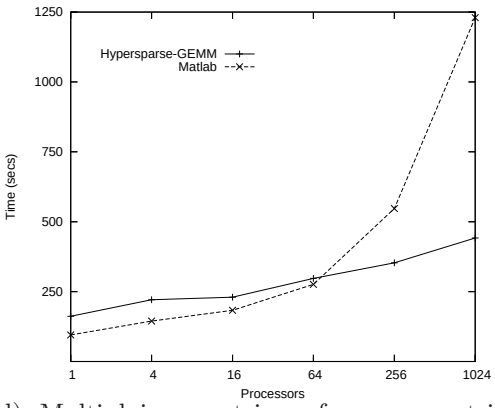
(a) Multiplying R-MAT matrices (R-MAT × R-MAT)



(b) Permuting an R-MAT matrix (R-MAT × Perm)



(c) Multiplying matrices from Erdős-Rényi graphs (Rand × Rand)



(d) Multiplying matrices from geometric graphs (`grid3d` × `grid3d`)

Figure 3.24: Model of scalability of SpGEMM kernels

In the case of R-MAT × R-MAT, the classical sequential algorithm is initially faster than Hypersparse_GEMM. For $p > 64$, however, the classical algorithm starts performing poorly because submatrices start getting hypersparse. To see why, consider the ratio of nnz to n for each submatrix:

$$\frac{nnz(\mathbf{A}_{ij})}{n/\sqrt{p}} = \frac{8\,n/p}{n/\sqrt{p}} = \frac{8}{\sqrt{p}}$$

This ratio is smaller than 1 for $p > 64$, making submatrices hypersparse. For $p = 1024$, our algorithm performs more than 5 times faster than the classical algorithm. Its scaling is also very good, showing almost flat curves.

In the case of multiplying an R-MAT matrix with a permutation matrix (R-MAT × Perm), poor scalability of the classical algorithm is more apparent. Our algorithm starts to outperform for as low as $p > 4$. The break-even point after which our algorithm dominates is lower in this case because permutation matrices are more sparse with only 1 nonzero per column/row.

Erdős-Rényi Random Graphs

We have conducted a single set of experiments where we multiply two matrices representing Erdős-Rényi random graphs. Looking at the timings shown in Figure 3.24(c), we see that the Hypersparse_GEMM dominates the classical algorithm (as implemented in Matlab) for most values for $p > 64$, when used as the sequential kernel of a 2D parallel SpGEMM. More importantly, when we reach

thousands of processors, our algorithms show their scalability for these input types as well. In particular, HYPERSPARSE_GEMM is more than 4 times faster than the classical algorithm for 1024 processors when multiplying Erdős-Rényi random matrices.

Regular 3D Grids

For our last set of experiments, we have used `grid3d` matrices. These matrices have a banded structure, which makes them unsuitable for 2D block decomposition since the off-diagonal processors sit idle without storing any nonzeros and performing any computation. Even though we are just timing the computational costs, ignoring parallelization overheads in this modeling study, the imbalance has an effect on the timing of submatrix multiplications. In particular, the heavy diagonals avoid hypersparsity to emerge, thus favoring the classical algorithm in this unrealistic setting.

To remedy this problem, we perform random permutations of vertices on both inputs before performing the multiplication. In other words, instead of computing $\mathbf{C} = \mathbf{AB}$, we compute $\mathbf{C}' = \mathbf{A}'\mathbf{B}' = (\mathbf{PAP^T})(\mathbf{PBP^T}) = \mathbf{PCP^T}$. Even after applying random symmetric permutations, submatrices in the diagonal are expected to have more nonzeros than others. This is because symmetric permutations es-

sentially relabel the vertices of the underlying graph, so they are unable to scatter the nonzeros in the diagonal.

Multiplications among diagonal blocks favor the classical sequential kernel because diagonal blocks can never become hypersparse no matter how much p increases. Multiplication among off-diagonal blocks are more suitable for our hypersparse kernel. More technically, our observation means

$$\text{flops}(\mathbf{A}_{ii}\,\mathbf{B}_{ii}) > \text{flops}(\mathbf{A}_{ii}\,\mathbf{B}_{ij}) > \text{flops}(\mathbf{A}_{ik}\,\mathbf{B}_{kj}).$$

Therefore, the variances in timings of submatrix multiplications are large compared with other sets of test matrices.

Asymptotic behavior of the algorithms is also slightly different in this case as it can be seen in Figure 3.24(d). Yet, our algorithm is around 4 times faster than the classical algorithm for $p = 1024$.

3.6 Parallel Scaling of Sparse SUMMA

3.6.1 Experimental Design

We have implemented two versions of the 2D parallel SpGEMM algorithms in C++. The first one is directly based on Sparse SUMMA and synchronous in nature. It does not use any MPI-2 features. The second implementation is asynchronous and uses one-sided communication features of MPI-2. In this section,

we report on the performance of the synchronous implementation only and leave the results of the asynchronous implementation to Section 3.7.3. We ran our code on the TACC's Ranger Cluster, which has four 2.3GHz quad-core processors in each node (16 cores/node). It has an Infiniband interconnect with 1GB/sec unidirectional point-to-point bandwidth and 2.3 microseconds max latency. We have experimented with multiple compilers and MPI implementations. We report our best results, which we achieved using `OpenMPI v1.3b` and `GNU Compiler (g++ v4.4)` with flag `-O3`.

For both implementations, our sequential HYPERSPARSE_GEMM routines return a set of intermediate triples that are kept in memory up to a certain threshold without being merged immediately. This allows for a more balanced merging, thus eliminating some unnecessary scans that degraded performance in a preliminary implementation [48].

In our experiments, instead of using random matrices (matrices from Erdős-Rényi random graphs), we used synthetically generated RMAT matrices, in order to achieve results closer to reality. The average number of nonzeros per column is 8 for those synthetically generated graphs.

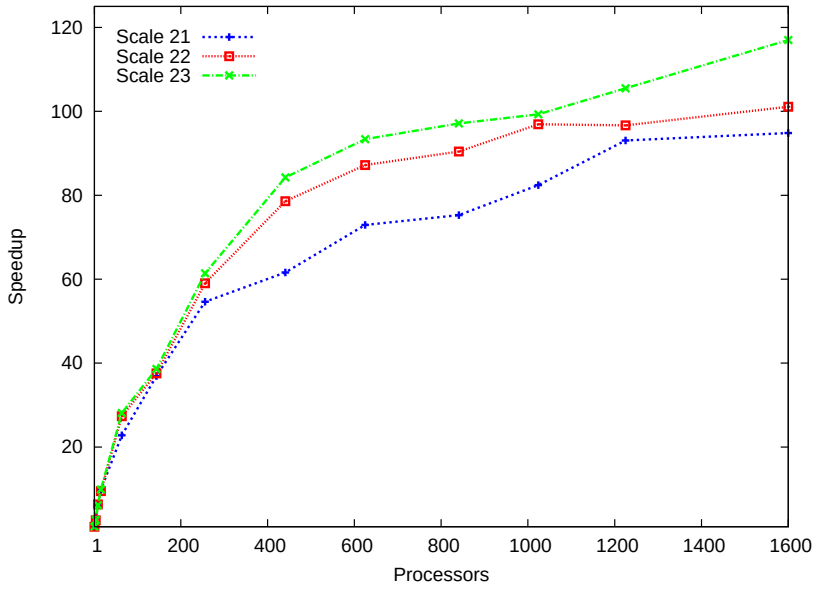


Figure 3.25: Observed speedup of of synchronous Sparse SUMMA for the R-MAT $\times$ R-MAT product on matrices having dimensions $2^{21} - 2^{23}$. Both axes are normal scale.

3.6.2 Experimental Results

Square Sparse Matrix Multiplication

In the first set of experiments, we multiply two R-MAT matrices that are structurally similar. This square multiplication is representative of the expansion operation used in the Markov clustering algorithm [82]. It is also a challenging case for our implementation due to high skew in nonzero distribution. We performed strong scaling experiments for different matrix dimensions ranging from 2^{21} to 2^{23}. Figure 3.25 shows the speedup we achieved. The graph shows linear speedup

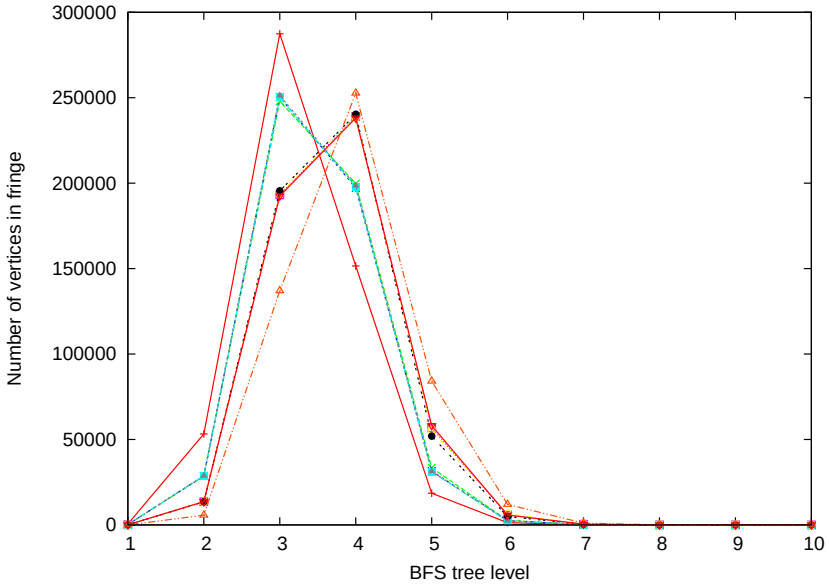


Figure 3.26: Fringe size per level during breadth-first search. Each one of ten plots is an average of 256 independent BFS operations on a graph of 1 million vertices and 8 million edges

(with slope 0.5) until around 50 processors; afterwards the speedup is proportional to the square root of the number of processors. Both results are in line with our analysis in Section 3.4.2.

Tall Skinny Right Hand Side Matrix

The second set of experiments involves multiplication of R-MAT matrices by tall skinny matrices of varying sparsity. This set of experiments serves multiple purposes. Together with the next set of experiments, they reveal the sensitivity of our algorithm to matrix orientations. It also examines the sensitivity to sparsity,

because we vary the sparsity of the right hand side matrix. Lastly, it is representative of the parallel breadth-first search that lies in the heart of our betweenness centrality implementation in Section 5.1. We varied the sparsity of the right hand side matrix from approximately 1 nonzero per column to 10^5 nonzeros per column, with multiplicative increments of 10. Our reasoning is application driven: at each level of breadth-first search, the current frontier (fringe) has as low as a few vertices but it can have as high as 300000 vertices. Figure 3.26 plots the number of vertices in the fringe at each level of the breadth-first search for 10 different runs (with different starting vertices) on a network of 1 million vertices and 8 million edges.

For our experiments, the R-MAT matrices on the left hand side have $c_1 = 8$ nonzeros per column and their dimensions vary from $n = 2^{20}$ to $n = 2^{26}$. The right hand side matrix is of size n-by-k, and its number of nonzeros per column, c_2 is varied from 1 to 10^5, with multiplicative increments of 10. Its width, k, varies from 128 to 8192 that grows proportionally to its length n. Hence, the total work is $W = O(c_1 c_2 k)$, the total memory consumption is $M = O(c_1 n + c_2 k)$, and total bandwidth requirement is is $O(M\sqrt{p})$.

We performed scaled speedup experiments where keep both $n/p = 2^{14}$ and $k/p = 2$ constant. This way, we were able to keep both memory consumption per

processor and work per processor constant at the same time. However, bandwidth requirements per processor increases by a factor of $\sqrt{p}$.

Figure 3.27 shows the three-dimensional performance graph. The timings for each slice along the XZ-plane (i.e. for every $c_2 = \{1, 10, ..., 10^5\}$ contour), is normalized to its running time on $p = 64$ processors. We do not cross-compare the absolute performances using different c_2 values, as our focus in this section is parallel scaling. The graph demonstrates that, except for the outlier case $c_2 = 1000$, we achieve the expected $\sqrt{p}$ slowdown due to communication costs. The performance we achieved for these large scale experiments, where we ran our code on up to 4096 processors, is remarkable.

Multiplication with the Restriction Operator

The multilevel method is widely used in the solution of numerical and combinatorial problems [194]. The method constructs smaller problems by successive coarsening of the problem domain. The simplest coarsening is perhaps graph contraction. One contraction step chooses two or more vertices in the original graph G to become a single aggregate vertex in the contracted graph G'. The edges of G that used to be incident to any of the vertices forming the aggregate now become incident to the new aggregate vertex in G'.

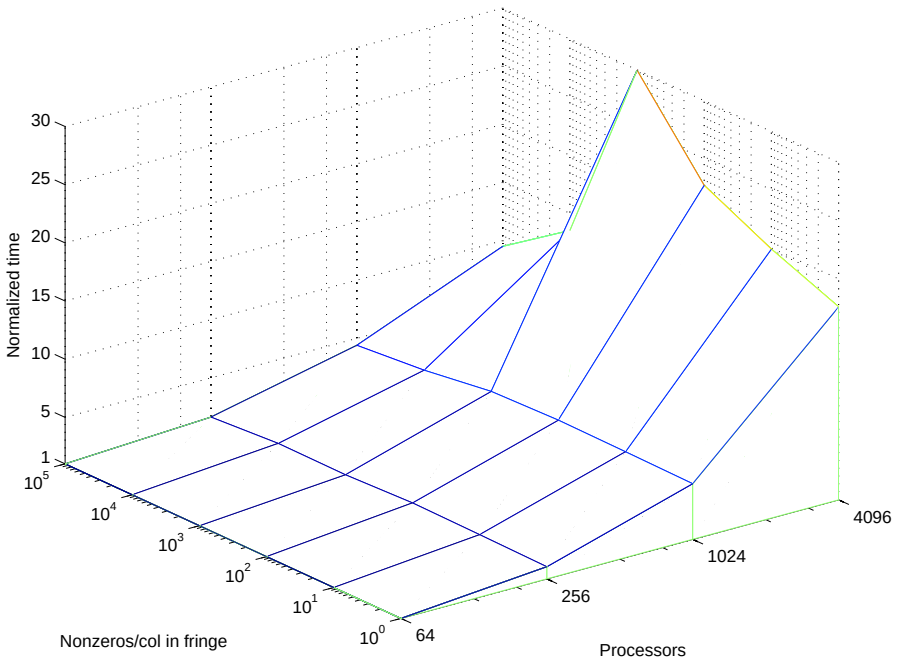


Figure 3.27: Weak scaling of R-MAT times a tall skinny Erdős-Rényi matrix. x (processors) and y (nonzeros per column on fringe) axes are logarithmic, whereas z (normalized time) axis is normal scale.

Constructing a coarser grid during the V-cycle of the Algebraic Multigrid (AMG) method [42] or graph partitioning [118] is a generalized graph contraction operation. Different algorithms need different coarsening operators. For example, a weighted (as opposed to strict) aggregation [58] might be preferred for partitioning problems. In general, coarsening can be represented as multiplication of the matrix representing the original fine domain (grid, graph, or hypergraph) by the restriction operator.

In this experiments, we use a simple restriction operation to perform graph contraction. Gilbert et al. [105] describe how to perform contraction using SpGEMM. Their elegant algorithm creates a special sparse matrix $\mathbf{S}$ with n nonzeros. The triple product $\mathbf{SAS}^\mathsf{T}$ contracts the whole graph at once. Making $\mathbf{S}$ smaller in the first dimension while keeping the number of nonzeros same changes the restriction order. For example, we contract the graph into half by using $\mathbf{S}$ having dimensions $n/2 \times n$, which is said to be of order 2.

Figure 3.28 shows the strong scaling of the operation $\mathbf{AS}^\mathsf{T}$ for R-MAT graphs of scale 23. We used restrictions of order 2, 4, and 8. Changing the interpolation order results in minor changes in performance. The experiment shows good scaling for up to 1024 processors.

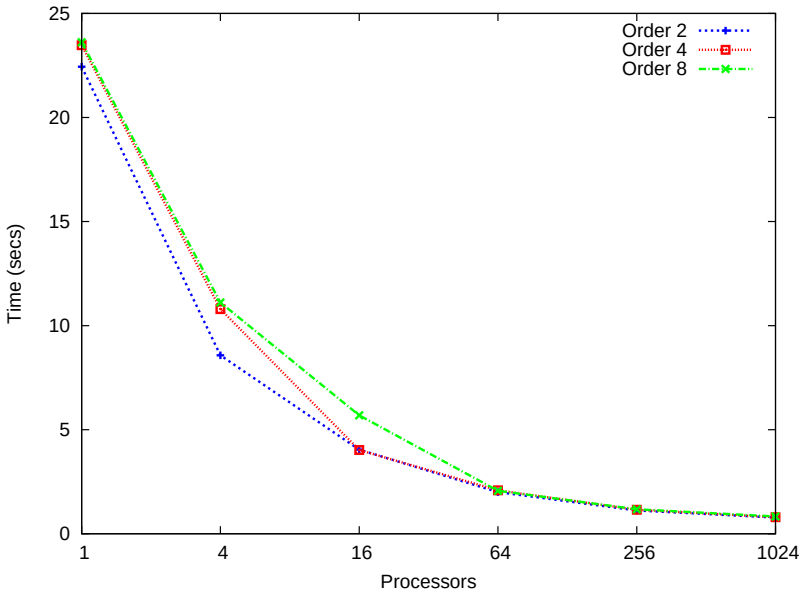


Figure 3.28: Strong scaling of multiplication with the restriction operator on the right, $\mathbf{A}' \leftarrow \mathbf{AS}^{\mathsf{T}}$. The graph is logarithmic on the x-axis.

3.7 Alternative Parallel Approaches

3.7.1 Load Balancing and Asynchronous Algorithms

In distributed memory dense matrix-matrix multiplication algorithms, each processor performs a total of W/p work where $W = N^3$. The sparse inputs are not so naturally balanced. Our experiments with randomly relabeling vertices (in matrix terms, applying a symmetric permutation) showed good premise where the maximum overall work for a single processor was only 9% more than the average work per processor, even when the initial matrix has significantly skewed degree

distribution[1]. Aiming for perfect load balance via graph or hypergraph partitioning [54, 55, 203] seems impractical whenever the matrices are not reused. Even when one of the matrices are fixed throughout the computation, load balance for SpGEMM can not be determined solely based on one operand, unlike SpMV. We do not know of any applications where both matrix operands have fixed structure for several subsequent multiplication operations, which might have justified complex load balancing.

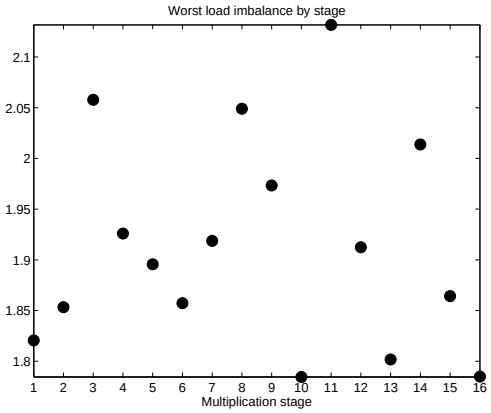
The sparse 2D algorithms presented in previous sections execute in a synchronous manner in s stages in their naive form. For sparse matrices, achieving good load balance per stage is harder than achieving load balance for the whole computation. This is because a local submatrix update such as $\mathbf{C}_{i,j} \leftarrow \mathbf{C}_{i,j} + \mathbf{A}_{i,k}\mathbf{B}_{k,j}$ might have significantly more work to do than another update at the same stage, say $\mathbf{C}_{i+1,j} \leftarrow \mathbf{C}_{i+1,j} + \mathbf{A}_{i+1,k}\mathbf{B}_{k,j}$. However, in a subsequent stage the roles of the (i, j)th and the $(i+1, j)$th processor might swap; hence balancing the load across stages. On the other hand, a barrier synchronization at each stage forces everyone to wait for the slowest update until they can proceed to the next stage. Hence, we expect an asynchronous algorithm to perform better than a synchronous one for matrices with highly skewed nonzero distribution.

[1]For sufficiently large matrices on 256 processors, as shown in Figure 3.30

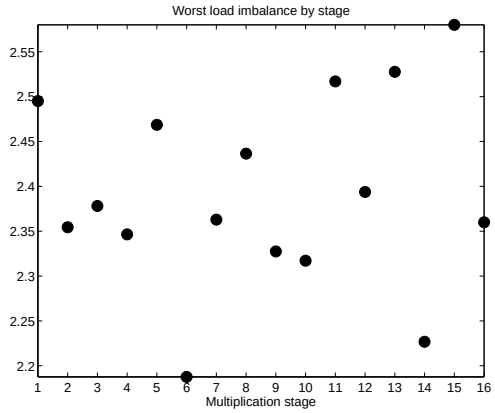
In order to quantify the severity of load imbalance, we performed a simulation of the Sparse Cannon algorithm that accounts for the computation (in terms of the number of actual flops only) and communication (*nnz* only) done by each processor. We varied the matrix dimension and the number of processors while the number of nonzeros per row/column were kept constant. For RMAT matrices with 8 nonzeros per column, the per-stage load imbalance with 256 processors is shown in Figure 3.29. Load imbalance is defined as the ratio of the maximum number of flops performed by any processor to the average number of flops. These plots are typical in the sense that we permuted the input matrices multiple times with different random permutations and plotted the results of the permutation that resulted in the median load imbalance.

Figure 3.30(a) shows the overall load imbalance for increasing matrix sizes on 256 processors. The problem becomes well balanced (i.e. it has less 10% load imbalance) for R-MAT inputs of scale 20 and larger. On the other hand, Figure 3.30(b) shows a comparison of the trends of overall and per-stage imbalances (average over all stages) with increasing number of processors and a fixed problem size.

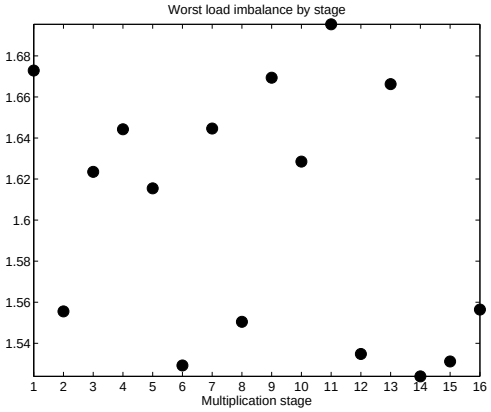
These results on Figures 3.29 and 3.30 suggest that per-stage load balance is significantly harder to achieve than load balance for the overall computation. Both tend to decrease as the problem size gets bigger, although per-stage load



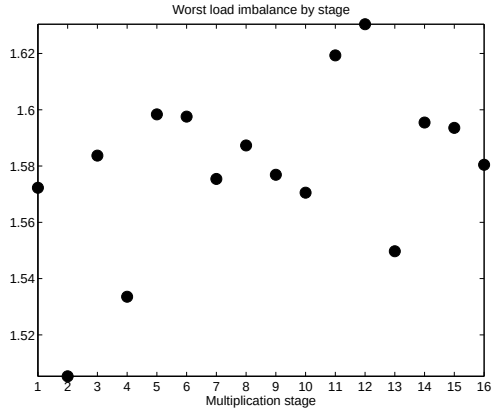
(a) RMAT Scale-15



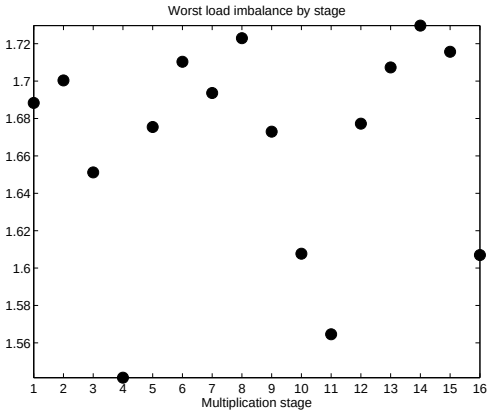
(b) RMAT Scale-16



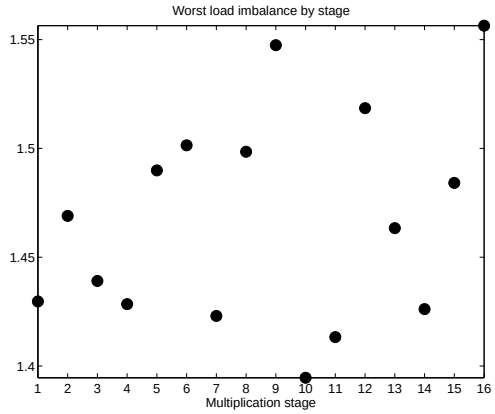
(c) RMAT Scale-17



(d) RMAT Scale-18

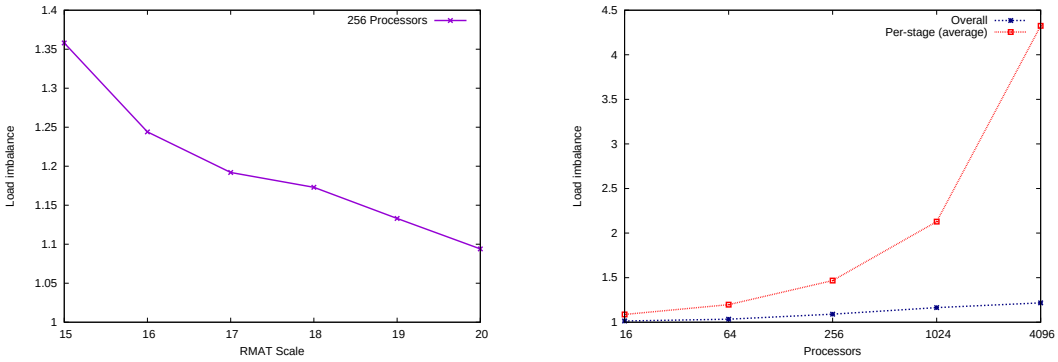


(e) RMAT Scale-19



(f) RMAT Scale-20

Figure 3.29: Load imbalance per stage for multiplying two RMAT matrices on 256 processors using Sparse Cannon



(a) Overall imbalance on a fixed number of ($p = 256$) processors

(b) Overall vs. per-stage imbalance for a fixed problem size (R-MAT scale 20)

Figure 3.30: Load imbalance during parallel multiplication of two RMAT matrices

imbalance has much wider variance and tends to decrease less smoothly than the overall load imbalance. The average per-stage load imbalance across all stages is 1.46 for inputs of scale 20. This means that a synchronous Sparse Cannon is likely to achieve 46% less speedup than we estimated in Section 3.5.1. By contrast, a perfectly asynchronous implementation would only pay 9% performance penalty due to load imbalance.

One-sided communication is the most suitable paradigm for implementing asynchronous SpGEMM. We used one-sided MPI-2 routines for portability, as GASNet [39] and ARMCI [153] are not as widely supported on supercomputers. It is still worth mentioning that even MPI poses some complications due to immaturity of implementations and vagueness in parts of the standard. We report our performance results using the passive target synchronization [152]. Explorations

on different one-sided approaches and issues associated with them can be found in Appendix A.

MPI-1 standard is inadequate to address the asynchronous implementation challenge. The blocking operations do trivially synchronize, and the non-blocking operations buffer the message and revert to a synchronous mode whenever the data is too large to fit in the buffers [172]. The basic requirement of an asynchronous SpGEMM is that the (i, j)th processor should be able to fetch its required submatrix from its original owner regardless of its computation stage at that moment. Although this can be achieved by the use of a helper thread that waits on the Send() operation, ready to serve any incoming Recv() requests, this approach has two drawbacks. Firstly, there is a substantial performance loss due to oversubscribing the processor. Secondly, general multithreaded MPI support is still in its infancy[2].

3.7.2 Overlapping Communication with Computation

In order to hide communication costs as much as possible, each processor starts prefetching one submatrix ahead while computing its current submatrix product. More concretely, processor $P(i, j)$ starts prefetching $\mathbf{A}_{i,k+1}$ and $\mathbf{B}_{k+1,j}$ while computing $\mathbf{A}_{i,k}\mathbf{B}_{k,j}$. To keep the memory footprint the same as the synchronous

[2]OpenMPI's MPI_THREAD_MULTIPLE support, which failed in our tests, is known to be untested

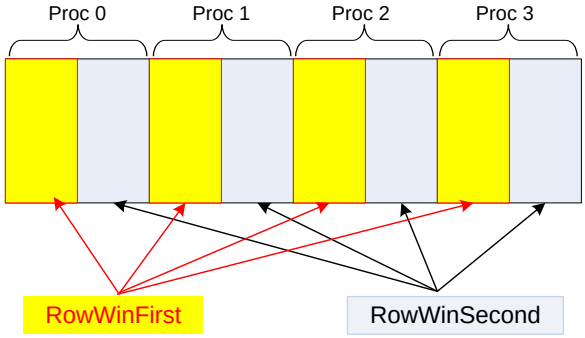


Figure 3.31: The split distribution of matrix **A** on a single processor row

Sparse SUMMA, we split the submatrices in half, so that each processor performs $2\sqrt{p}$ submatrix multiply-adds instead of $\sqrt{p}$. The distribution of matrix **A** on a single processor row is shown in Figure 3.31.

3.7.3 Performance of the Asynchronous Implementation

The pseudocode for our asynchronous implementation (in MPI/C++ notation) is shown in Figure 3.32. This implementation achieves two goals at once. It overlaps communication with computation as much as possible by prefetching next submatrices. It also achieves better load balance because it allows each processor to proceed independently without any global synchronizations.

Figure 3.33 compares the performance of the asynchronous implementation with the synchronous Sparse SUMMA implementation for the scale 22 R-MAT ×

```
// M1 is the first half of the local matrix M, M2 is the second
vector<Win> rwf = CreateWindows(RowWorld, A1);
vector<Win> rws = CreateWindows(RowWorld, A2);
vector<Win> cwf = CreateWindows(ColWorld, A1);
vector<Win> cws = CreateWindows(ColWorld, A2);

// Each window is made accessible to its neighbors in their
// respective processor row (in the case of A) and
// processor column (in the case of B)
ExposeWindows();

/* Perform initial two fetches and multiply first halfs */

for(int i = 1; i < stages; ++i)      // main loop
{
      CResult += SpGEMM(*ARecv1, *BRecv1, false, true);

      // wait for the previous second halfs to complete
      CompleteFetch(rws);
      CompleteFetch(cws);

      Aowner = (i+Aoffset) % stages;
      Bowner = (i+Boffset) % stages;

      // start fetching the current first half
      StartFetch(ARecv1, Aowner, rwf);
      StartFetch(BRecv1, Bowner, cwf);

      // while multiplying (completed) previous second halfs
      CResult = SpGEMM(*ARecv2, *BRecv2, false, true);

      /* now wait for the current first half to complete */
      /* start prefetching the current second half */
}
/* perform the last pieces of computation */
```

Figure 3.32: Partial C++ code partial for asynchronous SpGEMM using one-sided communication and split prefetching for overlapping communication with computation

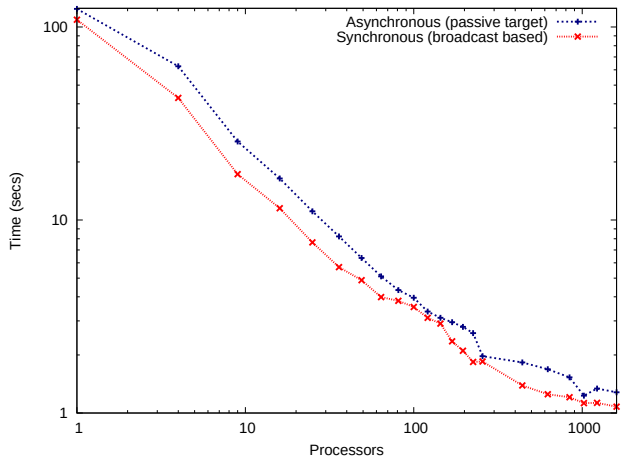


Figure 3.33: Performances of the asynchronous and synchronous implementations of the Sparse SUMMA. In this experiment, we multiply two R-MAT matrices of scale 22. Both axes are on log-scale

R-MAT product. Although they scale similarly well, the synchronous implementation is $6 - 47\%$ faster.

Overall poor performance of the asynchronous implementation is partly due to the extra operations such splitting and joining matrices. However, their share in the computation time goes down as we increase the number of processors, so this does not explain the performance difference on large number of processors.

We first thought the performance hit was due to the progress threads that are used by MPI implementations on Infiniband [183] to ensure asynchronous progress. On the other hand, we ran the same code using 4 threads per node so that the progress threads will not oversubscribe the individual cores. Figure 3.34

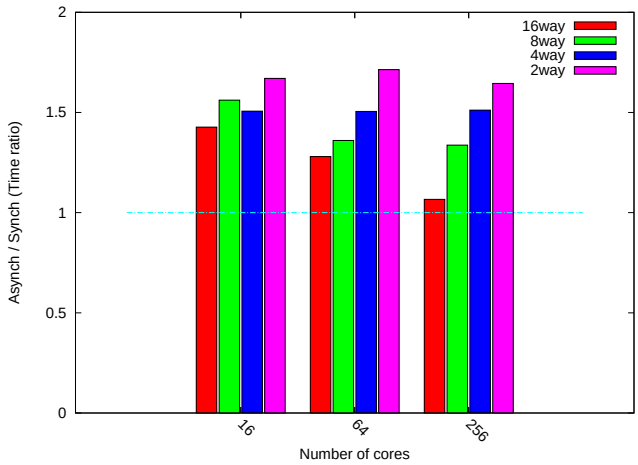


Figure 3.34: Performance comparison of the asynchronous and synchronous implementations using different number of cores per node. The vertical axis shows the ratio of time it takes to execute SpGEMM with the asynchronous implementation to the time it takes with the synchronous implementation. A t-way run on p cores is performed using p/t nodes.

shows that the performance difference between the synchronous and asynchronous implementations grows as we use less cores per node. Either our asynchronous implementation, which uses one-sided point-to-point communication instead of blocking collective communication, or the underlying MPI implementation does not take full advantage of the extra bandwidth available per core.

We are not able to explain the load imbalance that happens in practice. Let T_p be the time to complete the SpGEMM procedure on p processors. If T_i is the time for the ith processor to complete its local procedure, then $T_p = \max(T_i)$ over all i

due to wait times. For the asychronous implementation, our preliminary profiling (on 256 cores) revealed that the fastest processor spends more time waiting for the other processors than doing useful computation. On average, a processor spent about 1/3rd of its time waiting.

The slowdown due to the asynchronous execution was previously experienced on the Connection Machine CM5 [139] on programs with regular communication patterns. Brewer and Kuszmaul [41] found out that an initial skew of processors slowed down the overall computation on the CM5, as receiver queues started to back off. The CM-5 data network is similar to Ranger's, in the sense that they both use a fat-tree [137] interconnect. However, the problem with the CM-5 was the contention on the receivers due to the computational cost of receiving packets. Ranger's Infiniband interconnect, on the other hand, has RDMA support for this task. However, we do not know whether MPI-2 functions have been implemented to fully take advantage of the network's capabilities. In conclusion, revealing the exact cause of the poorer performance of the asynchronous implementation needs further research and more performance profiling.

3.8 Future Work

Our mathematical modeling of the parallel algorithms in Section 3.3 is an average-case analysis assuming independent uniform random distribution of nonzeros, which translates into the Erdős-Rényi random graph model. More realistic models should assume skewed nonzero distributions, such as power-law distributions. Ultimately, average case analysis has its limitations because it needs to assume an underlying distribution. On the other hand, worst case analysis does not make a lot of sense for our problem, because there are certain sparse matrix pairs that will create a dense output when multiplied. Therefore, a smoothed analysis [186] of the sparse matrix multiplication algorithms, both sequentially and in parallel, would be a significant advancement.

Load imbalance is not severe for sufficiently large matrices, even in the absence of asynchronous progress. Our one-sided communication approach was based on remote get operations in order to avoid fence synchronization. Given the acceptable load balance for large matrices, it is worth exploring an option with fence synchronization and remote put operations. This proposed implementation will still use one-sided communication but all processors in the processor row/column will need to synchronize after the put operation. We expect better performance

because it only takes one trip to complete a remote put operation whereas remote get requires a roundtrip.

Our SpGEMM routine might be extended to handle matrix chain products. In particular, the sparse matrix triple product (RAP) is heavily used in the coarsening phase of the algebraic multigrid method [4]. Sparse matrix indexing and parallel graph contraction also require sparse matrix triple product [105]. The support for sparse matrix chain products eliminates temporary intermediate products and allows more optimizations, such as performing structure prediction [63] and finding the optimal parenthesization based on the sparsity of the inputs.

Finally, there is a need for hierarchical parallelism due to vast differences in the costs of inter-node and intra-node communication. The flat parallelism model does not only lose the opportunity to exploit the faster on-chip network, but it also increases the contention on the off-chip links. We observed that the inter-node communication becomes slower as the number of cores per node increases because more processes are competing for the same network link. Therefore, designing a hierarchically parallel Sparse GEMM algorithm is an important future direction.

Chapter 4

The Combinatorial BLAS: Design and Implementation

> *My impression was and is that many programming languages and tools represent solutions looking for problems, and I was determined that my work should not fall into that category*
>
> Bjarne Stroustrup

The Combinatorial BLAS library is a parallel library for graph computations. It is intented to provide a common interface for high-performance graph kernels. It is unique among other graph libraries for combining scalability with distributed memory parallelism. We borrowed ideas from the domain of parallel numerical analysis and applied them to parallel graph computations.

4.1 Motivation

The Matlab reference implementation of the HPCS Scalable Synthetic Compact Applications graph analysis (SSCA#2) benchmark was an important step

towards using linear algebra operations for implementing graph algorithms. Although it was a success in terms of expressibility and ease of implementation, its performance was about 50% worse than the best serial implementation. Mostly, the slowdown was due to limitations of Matlab for performing integer operations [1]. The parallel scaling was also limited on most parallel Matlab implementations.

The idea of having a BLAS-like library for doing graph computation is driven by the desire to create a general purpose library that supports rapid implementation of graph algorithms using a small yet important subset of linear algebra operations. The library should also be in parallel and scale well due to the massive size of graphs in many modern applications.

4.2 Design Philosophy

4.2.1 The Overall Design

The first class citizens of the Combinatorial BLAS are distributed sparse matrices. Application domain interactions that are abstracted into a graph are concretely represented as a sparse matrix. Therefore, all non-auxiliary functions are designed to operate on sparse matrix objects. There are three other types of objects that are used by some of the functions: dense matrices, dense vectors,

[1] Matlab does not support integer data elements for its sparse matrices

sparse vectors. Concrete data structures for these objects are explained in detail in Section 4.3.

We follow some of the guiding design principles of the popular and successful PETSc package [21]. We achieve extensibility by defining a common abstraction for all sparse matrix storage formats, making it possible to implement a new format and plug it in without changing rest of the library. For scalability, it is possible (and encouraged) to create objects by passing special MPI communicator objects instead of using the default COMM::WORLD object that includes all the processes in the system. This use of communicators also helps to avoid inter-library and intra-library collisions. We do not attempt to create the illusion of a flat address space; communication is internally handled by parallel classes of the library. Likewise, we do not always provide storage independence due to our emphasis on high performance. Some operations have different semantics depending on whether the underlying object is sparse or dense.

The Combinatorial BLAS routines (API functions) are supported both sequentially and in parallel. The communication is managed within the parallel versions (i.e., versions operating on parallel objects) of these high-level operations, which call the sequential versions for computation on local data. This symmetry of function prototypes has a nice effect on interoperability. The parallel objects can just treat their internally stored sequential objects as black boxes supporting the API

functions. Conversely, any sequential class becomes fully compatible with the rest of the library as long as it supports the API functions and allows access to its internal arrays through an adapter object. This decoupling of parallel logic from sequential parts of the computation is one of the distinguishing features of the Combinatorial BLAS.

4.2.2 The Combinatorial BLAS Routines

We selected the operations to be supported by the API by a top-down, application driven process. Commonly occurring computational patterns in many graph algorithms are abstracted into a few linear algebraic kernels that can be efficiently mapped into the architecture of distributed memory computers. The API is not intended to be final and will be extended as more applications are analyzed and new algorithms are invented.

We address the tension between generality and performance by the zero overhead principle: Our primary goal is to provide work-efficiency for the targeted graph algorithms. The interface is kept general, simple, and clean so long as doing so does not add significant overhead to the computation. The guiding principles in the design of the API are listed below, each one illustrated with an example.

(1) *If multiple operations can be handled by a single function prototype without degrading the asymptotic performance of the algorithm they are to be part*

of, then we provide a generalized single prototype. Otherwise, we provide multiple prototypes.

For example, it is tempting to define a single function prototype for elementwise operations on sparse matrices. Although it seems achievable by passing a *binop* parameter, semantics of sparse matrices differ depending on the binary operation. For instance, ignoring numerical cancellation, elementwise addition is most efficiently implemented as a union of two sets while multiplication is the intersection. Other elementwise operations between two sparse matrices are handled similarly, but using different functions. If it proves to be efficiently implementable (using either function object traits or run-time type information), all elementwise operations between two sparse matrices may have a single function prototype in the future.

On the other hand, the data access patterns of matrix-matrix and matrix-vector multiplications are independent of the underlying semiring. In fact, many connectivity problems use the *Boolean semiring* $(\{0,1\}, \vee, \wedge, 0, 1)$ and many shortest path algorithms use the *tropical semiring* $(\mathbb{R}^+, \min, +, \infty, 0)$. As a result, the sparse matrix-matrix multiplication routine SpGEMM and the sparse matrix-vector multiplication routine SpMV accept a parameter representing the semiring. The SpGEMM function also expects two additional parameters, trA and trB,

and depending on those, computes one of the following operations: $\mathbf{C} \leftarrow \mathbf{A} \cdot \mathbf{B}$, $\mathbf{C} \leftarrow \mathbf{A}^{\mathsf{T}} \cdot \mathbf{B}$, $\mathbf{C} \leftarrow \mathbf{A} \cdot \mathbf{B}^{\mathsf{T}}$, $\mathbf{C} \leftarrow \mathbf{A}^{\mathsf{T}} \cdot \mathbf{B}^{\mathsf{T}}$.

(2) *If an operation can be efficiently implemented by composing a few simpler operations, then we do not provide a special function for that operator.*

For example, making a nonzero matrix $\mathbf{A}$ column stochastic can be efficiently implemented by first calling REDUCE on $\mathbf{A}$ to get a dense row vector $\mathbf{v}$ that contains the sums of columns, then obtaining the multiplicative inverse of each entry in $\mathbf{v}$ by calling the APPLY function with the unary function object that performs $f(v_i) = 1/v_i$ for every v_i it is applied to, and finally calling SCALE($\mathbf{v}$) on $\mathbf{A}$ to effectively divide each nonzero entry in a column by its sum. Consequently, we do not provide a special function to make a matrix column stochastic.

On the other hand, a commonly occurring operation is to zero out some of the nonzeros of a sparse matrix. This often comes up in graph traversals, where $\mathbf{X}^k$ represents the set of vertices that are discovered during the kth iteration (i.e. the kth frontier). Multiplying $\mathbf{X}^k$ by the adjacency matrix of the graph yields another matrix that might include some of the previously discovered vertices in addition to the set of vertices that are discovered during the $(k+1)$st iteration. Those old vertices need to be pruned from the frontier before starting the next iteration. One way to implement this pruning operation is to keep a matrix $\mathbf{Y}$ that includes a zero for every vertex that has been discovered before, and nonzeros

elsewhere. Performing an elementwise multiplication with $\mathbf{Y}$ yields the desired frontiers matrix $\mathbf{X}^{k+1}$. However, this approach might not be work-efficient since $\mathbf{Y}$ will often be dense, especially in the early stages of the graph traversal.

Consequently, we provide a generalized function SpEWIseX that performs the elementwise multiplication of sparse matrices $op(\mathbf{A})$ and $op(\mathbf{B})$. It also accepts two auxiliary parameters, notA and notB, that are used to negate the sparsity structure of $\mathbf{A}$ and $\mathbf{B}$. If notA is true, then $op(\mathbf{A})(i, j) = 0$ for every nonzero $\mathbf{A}(i, j) \neq 0$ and $op(\mathbf{A})(i, j) = 1$ for every zero $\mathbf{A}(i, j) = 0$. The role of notB is identical. Direct support for the logical NOT operations is crucial to avoid the explicit construction of the dense $not(\mathbf{B})$ object.

(3) *To avoid expensive object creation and copying, many functions also have in-place versions. For operations that can be implemented in place, we deny access to any other variants only if those increase the running time.*

For example, Scale($\mathbf{B}$) is a member function of the sparse matrix class that takes a dense matrix as a parameter. When called on the sparse matrix $\mathbf{A}$, it replaces each $\mathbf{A}(i, j) \neq 0$ with $\mathbf{A}(i, j) \cdot \mathbf{B}(i, j)$. This operation is implemented only in-place because $\mathbf{B}(i, j)$ is guaranteed to exist for a dense matrix, allowing us to perform a single scan of the nonzeros of $\mathbf{A}$ and update them by doing fast lookups on $\mathbf{B}$. Not all elementwise operations can be efficiently implemented in-place (for example elementwise addition of a sparse matrix and a dense matrix will

Table 4.1: Summary of the current API for the Combinatorial BLAS

Function	Applies to	Parameters	Returns
SpGEMM	Sparse Matrix (as friend)	**A**, **B**: sparse matrices trA: transpose **A** if true trB: transpose **B** if true	Sparse Matrix
SpMV	Sparse Matrix (as friend)	**A**: sparse matrices **x**: dense vector(s) trA: transpose **A** if true	Sparse Matrix
SpEWiseX	Sparse Matrices (as friend)	**A**, **B**: sparse matrices notA: negate **A** if true notB: negate **B** if true	Sparse Matrix
Reduce	Any Matrix (as method)	dim: dimension to reduce *binop*: reduction operator	Dense Vector
SpRef	Sparse Matrix (as method)	**p**: row indices vector **q**: column indices vector	Sparse Matrix
SpAsgn	Sparse Matrix (as method)	**p**: row indices vector **q**: column indices vector **B**: matrix to assign	none
Scale	Any Matrix (as method)	**rhs**: any object (except a sparse matrix)	none
Scale	Any Vector (as method)	**rhs**: any vector	none
Apply	Any Object (as method)	*unop*: unary operator (applied to nonzeros)	none

produce a dense matrix), so the implementer is free to declare them as members of the dense matrix class or declare them as global functions returning a new object.

(4) *In-place operations have slightly different semantics depending on whether the operands are sparse or dense. In particular, the semantics favor leaving the sparsity pattern of the underlying object intact as long as another function (possibly not in-place) handles the more conventinal semantics that introduces/deletes nonzeros.*

For example, SCALE is an overloaded method, available for all objects. It does not destroy sparsity when called on sparse objects and it does not introduce sparsity when called on dense objects. The semantics of the particular SCALE method is dictated by its the class object and its operand. Called on a sparse matrix $\mathbf{A}$ with a vector $\mathbf{v}$, it independently scales nonzero columns (or rows) of the sparse matrix. For $\mathbf{v}$ being a row vector [2], SCALE replaces every nonzero $\mathbf{A}(i, j)$ with $\mathbf{v}(j) \cdot \mathbf{A}(i, j)$. The parameter $\mathbf{v}$ can be dense or sparse. In the latter case, only a portion of the sparse matrix is scaled. That is, $\mathbf{v}(j)$ being zero for a sparse vector does not zero out the corresponding jth column of $\mathbf{A}$. The SCALE operation never deletes columns from $\mathbf{A}$; deletion of columns is handled by the more expensive SPASGN function.

[2]Row/column vector distinction changes the semantics only. Row and column vectors (even distributed versions) are stored the same way.

SpAsgn and SpRef are generalized sparse matrix assignment and indexing operations. They are very powerful primitives that take vectors $\mathbf{p}$ and $\mathbf{q}$ of row and column indices. When called on the sparse matrix $\mathbf{A}$, SpRef returns a new sparse matrix whose rows are the $\mathbf{p}(i)$th rows of $\mathbf{A}$ for $i = 0, ..., length(\mathbf{p})-1$ and whose columns are the $\mathbf{q}(j)$th columns of $\mathbf{A}$ for $j = 0, ..., length(\mathbf{q})-1$. SpAsgn has similar syntax, except that it returns a reference (an modifiable lvalue) to some portion of the underlying object as opposed to returning a new object. Internally, the implementer is free (and encouraged) to use different subroutines for special cases such as row-wise ($\mathbf{A}(i,:)$), column-wise ($\mathbf{A}(:,j)$), and element-wise ($\mathbf{A}(i,j)$) indexing/assignment.

4.3 A Reference Implementation

4.3.1 The Software Architecture

In our reference implementation, the main data structure is a distributed sparse matrix object ***SpDistMat*** which HAS-A local sparse matrix that can be implemented in various ways as long as it supports the interface of the base class ***SpMat***. All features regarding distributed-memory parallelization, such as the communication patterns and schedules, are embedded into the distributed objects (sparse and dense) through the ***CommGrid*** object. Global properties of

distributed objects, such as the total number of nonzeros and the overall matrix dimensions, are not explicitly stored. They are computed by reduction operations whenever necessary. The software architecture for matrices is illustrated in Figure 4.1. Although the inheritance relationships are shown in the traditional way (via inclusion polymorphism [53]), the class hierarchies are static, obtained by the parameterizing the base class with its subclasses as explained below.

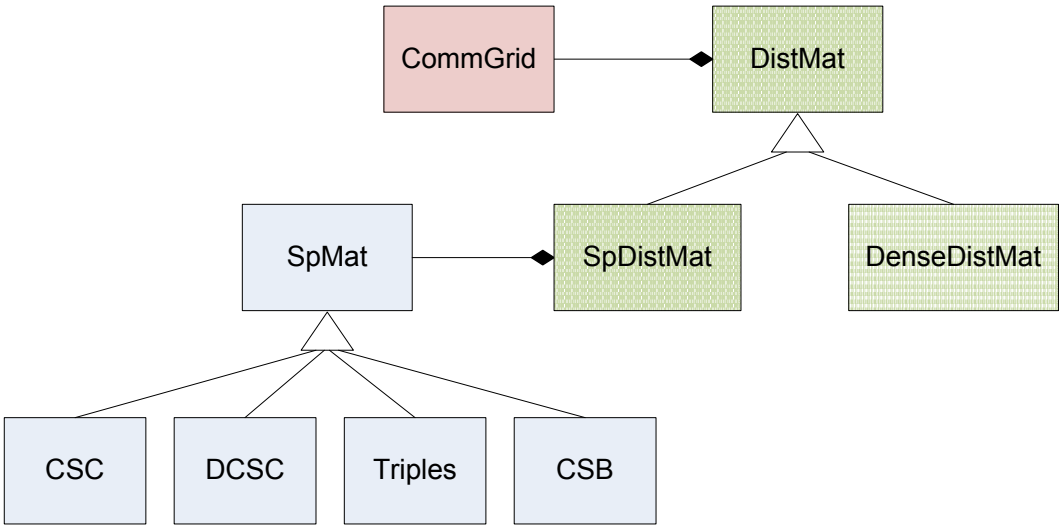


Figure 4.1: Software architecture for matrix classes

To enforce a common interface as defined by the API, all types of objects derive from their corresponding base classes. The base classes only serve to dictate the interface. This is achieved through static object oriented programming (OOP) techniques [51] rather than expensive dynamic dispatch. A trick known as the

Curiously Recurring Template Pattern (CRTP) [66] emulates dynamic dispatch statically, with some limitations. These limitations, such as the inability to use heterogeneous lists of objects that share the same type class, however, are not crucial for the Combinatorial BLAS. In CRTP, the base class accepts a template parameter of the derived class.

The SpMat base class implementation is given as an example in Figure 4.2. As all exact types are known at compile time, there are no runtime overheads arising from dynamic dispatch. In the presence of covariant arguments, static polymorphism through CRTP automatically allows for better type checking of parameters. In the SpGEMM example, with classical OOP, one would need to dynamically inspect the actual types of **A** and **B** to see whether they are compatible; and if they are, to call the right subroutines. This requires expensive run-time type information queries and *dynamic_cast()* operations. More problematically, it is unsafe, as any unconforming set of parameters will lead to a run-time error or an exception. Static OOP catches any such incompatibilities in compile time.

The SpMat object is local to a node but it need not be sequential. It can be implemented as a shared-memory data structure, amenable to thread-level parallelization. This flexibility will allow future versions of the Combinatorial BLAS algorithms to support hybrid parallel programming paradigms. The main distinguishing feature of SpMat is the contiguous storage of its sparse matrix,

```
// Abstract base class for all derived serial sparse matrix classes
// Contains no data members, hence no copy ctor/assignment operator
// Uses static polymorphism through curiously recurring templates
// Template parameters:
//  IT (index type), NT (numerical type), DER (derived class type)
template <class IT, class NT, class DER>
class SpMat
{
    typedef SpMat<IT,NT,DER> SpMatIns;
public:
    // Standard destructor, copy ctor and assignment are
    //  generated by compiler, they all do nothing !
    // Default constructor also exists, and does nothing more
    //  than creating Base<Derived>() and Derived() objects
    // One has to call the Create function to get a nonempty object
    void Create (const vector<IT>& essentials);

    SpMatIns operator() (const vector<IT>& ri, const vector<IT>& ci);

    template <typename SR>       // SR: Semiring object
    void SpGEMM (SpMatIns & A, SpMatIns & B, bool TrA, bool TrB);

    template <typename NNT>      // NNT: New numeric type
    operator SpMatIns() const;

    void Split (SpMatIns & partA, SpMatIns & partB);
    void Merge (SpMatIns & partA, SpMatIns & partB);

    Arr<IT,NT> GetArrays() const;
    vector<IT> GetEssentials() const;

    void Transpose();

    bool operator== (const SpMatIns & rhs) const;

    ofstream& put(ofstream& outfile) const;
    ifstream& get(ifstream& infile);

    bool isZero() const;
    IT getnrow() const;
    IT getncol() const;
    IT getnnz() const;
}
```

Figure 4.2: Partial C++ interface of the base SpMat class

making it accessible by all other components (threads/processes). In this regard, it is different from the SpDistMat, which distributes the storage of its sparse matrices.

We observe that the commonality between all sparse matrix storage formats is their use of multiple arrays. Therefore, the parallel classes handle object creating and communication through what we call an ***Essentials*** object, which is an adapter for the actual sparse matrix object. The Essentials of a sparse matrix object is its dimensions, number of nonzeros, starting addresses of its internal arrays and the sizes of those arrays. Through *GetEssentials*() and *Create*(*Essentials ess*) functions, any SpDistMat object can internally have any SpMat object. For example, communication can be overlapped with computation in the SpGEMM function by prefetching the internal arrays through one sided communication. Alternatively, another SpDistMat class that uses a completely different communication library, such as GASNet [39] or ARMCI [153], can be implemented without requiring any changes to the sequential SpMat object.

Most combinatorial operations use more than the traditional floating-point arithmetic, with integer and boolean operations being prevalent. To provide the user the flexibility to define their objects (matrices and vectors) with any scalar type, all of our classes and functions are templated. A practical issue is to be able perform operations between two objects holding different scalar types, e.g.,

multiplication of a boolean sparse matrix by an integer sparse matrix. Explicit upcasting of one of the operands to a temporary object might have jeopardized performance due to copying of such big objects. The template mechanism of C++ provided a neat solution to the mixed mode arithmetic problem by providing automatic type promotion through trait classes [25]. Arbitrary semiring support for matrix-matrix and matrix-vector products is allowed by passing a class (with static `add` and `multiply` functions) as a template parameter to corresponding SpGEMM and SpMV functions.

4.3.2 Management of Distributed Objects

The processors are logically organized as a two-dimensional grid in order to limit most of the communication to take place along a processor column or row with $\sqrt{p}$ processors, instead of communicating potentially with all p processors. The partitioning of distributed matrices (sparse and dense) follows this processor grid organization, using a 2D block decomposition, also called the checkerboard partitioning [107]. Figure 4.3 shows this for the sparse case.

Portions of dense matrices are stored locally as two dimensional dense arrays in each processor. Sparse matrices (SpDistMat objects), on the other hand, have many possible representations, and the right representation depends on the particular setting or the application. The problems with using the popular compressed

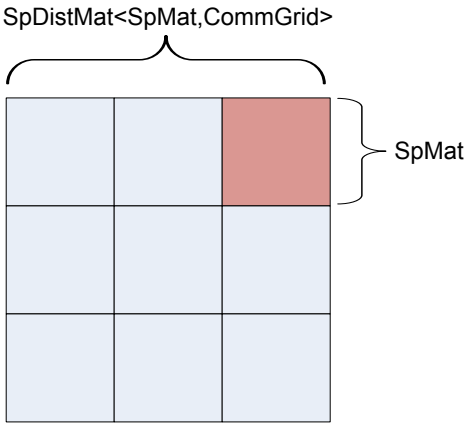


Figure 4.3: Distributed sparse matrix class and storage

sparse rows (CSR) or compressed sparse columns (CSC) representations in a 2D block decomposition are explained in Section 3.2.1. The triples format does not have the same problems but it falls short of efficiently supporting some of the fundamental operations. Therefore, our reference implementation uses the DCSC format, which is explained in detail in Section 3.2.1. As previously mentioned, this choice is by no means exclusive and anybody can replace the underlying sparse matrix storage format with his or her favorite format without needing to change other parts of the library, as long as the format implements the fundamental sequential API calls mentioned in the previous section.

For distributed vectors, data is stored only on the diagonal processors of the 2D processor grid. This way, we achieve a symmetry in performance of matrix-vector and vector-matrix multiplications. The high level structure and parallelism

of sparse and dense vectors are the same, the only difference being how the local data is stored in processors. A dense vector naturally uses a dense array, while a sparse vector is internally represented as a list of index-value pairs.

Chapter 5

The Combinatorial BLAS: Applications and Performance Analysis

> *The joy of life consists in the exercise of one's energies, continual growth, constant change, the enjoyment of every new experience. To stop means simply to die. The eternal mistake of mankind is to set up an attainable ideal.*
>
> Aleister Crowley (1875-1947)

This chapter presents two applications of the Combinatorial BLAS library. We report the performance of two algorithms on distributed-memory clusters, implemented using the Combinatorial BLAS primitives. The code for these applications, along with an alpha release of the complete library, can be freely obtained from `http://gauss.cs.ucsb.edu/code/index.shtml`.

5.1 Betweenness Centrality

Betweenness centrality [96], a centrality metric based on shortest paths, is the main computation on which we evaluate the performance of our proof-of-concept implementation of the Combinatorial BLAS. There are two reasons for this choice. Firstly, it is a widely-accepted metric that is used to quantify the relative importance of vertices in the graph. Betweenness centrality (BC) of a vertex captures the normalized ratio of the number of shortest paths that pass through a vertex to the total number of shortest paths in the graph. This is formalized in Equation 5.1 where σ_{st} denote the number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of such paths passing through vertex v.

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{5.1}$$

A vertex v with a high betweenness centrality is therefore an important one based on at least two different interpretations. From the point of view of other vertices, it is a highly sought-after hop for reaching others as quickly as possible. The second possible interpretation is that v itself is the best-situated vertex to reach others as quickly as possible.

The second reason for presenting the betweenness centrality as a success metric is its quantifiability. It is part of the HPC Scalable Graph Analysis Benchmarks

(formerly known as the HPCS Scalable Synthetic Compact Applications #2 [16]) and various implementations on different platforms exist [18, 144, 190] for comparison.

We compute betweenness centrality using Brandes' algorithm [40]. It computes single source shortest paths from each node in the network and increases the respective BC score for nodes on the path. The algorithm requires $O(nm)$ time for unweighted graphs and $O(nm + n^2 \log n)$ time for weighted graphs, where n is the number of nodes and m is the number of edges in the graph. The sizes of real-world graphs are prohibitive for exact calculation, so we resort to efficient approximations. An unbiased estimator of the betweenness centrality score has been proposed based on sampling nodes from which to compute single-source shortest paths [19]. The resulting scores approximate a uniformly scaled version of the actual betweenness centrality score. We only focus on unweighted graphs in this performance study.

Following the specification of the graph analysis benchmark [16], we used R-MAT matrices as inputs. Due to the prohibitive cost of the exact algorithm, we used the approximate algorithm with 8192 starting vertices. We measure the performance using the Traversed Edges Per Second (TEPS) rate, which is an algorithmic performance count that is independent of the particular implementation [16]. We randomly relabeled the vertices in the generated graph, before

storing it for subsequent runs. For reproducibility of results, we chose starting vertices using a deterministic process, specifically excluding disconnected vertices as starting vertices (which would have boosted the TEPS scores artificially).

We implemented an array-based formulation of the Brandes' algorithm, due to Robinson and Kepner. A reference MATLAB implementation is publicly available from the Graph Analysis webpage [17]. The workhorse of the algorithm is a parallel breadth-first search that is performed from multiple source vertices. In Combinatorial BLAS, one step of the breadth-first search is implemented as the multiplication of the transpose of the adjacency matrix of the graph with a rectangular matrix $\mathbf{X}$ where the ith column of $\mathbf{X}$ represents the current frontier of the ith independent breadth-first search tree. Inially, each column of $\mathbf{X}$ has only one nonzero that represents the starting vertex of the breadth-first search, i.e. $\mathbf{X}(:, i) = e_j$ where vertex j is the i-th starting vertex for the current iteration. Similarly, the tallying step is also implemented as an SpGEMM operation.

For the performance results presented in this section, we use a synchronous implementaton of the Sparse SUMMA algorithm described in Section 3.3.5, because it is the most portable SpGEMM implementation that relies only on simple MPI-1 features. The other Combinatorial BLAS primitives that are used for implementing the betweenness centrality algorithm are reductions along one dimension and elementwise operations for sparse/sparse, sparse/dense, and dense/sparse input

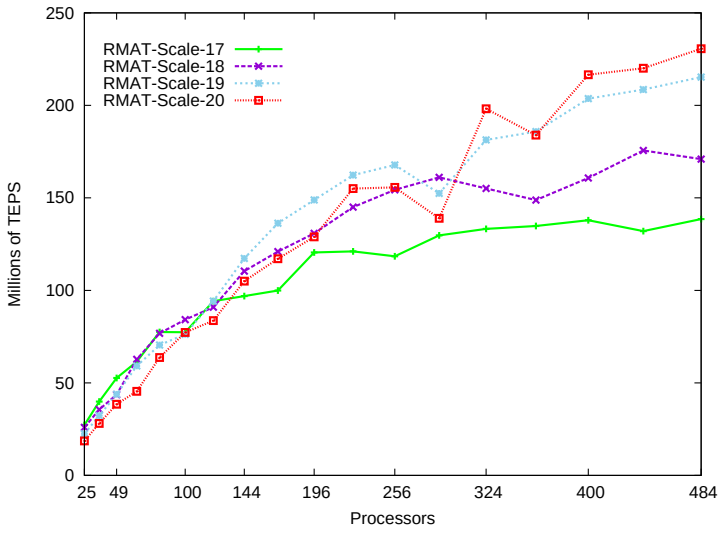


Figure 5.1: Parallel strong scaling of the distributed-memory betweenness centrality implementation (smaller input sizes)

pairs. The experiments are run on TACC's Lonestar cluster, which is composed of dual-socket dual-core nodes, connected by an Infiniband interconnect. Each individual processor is an Intel Xeon 5100, clocked at 2.66 GHz. We used the recommended Intel C++ compilers (version 10.1), and the MVAPICH2 implementation of the MPI.

5.1.1 Parallel Strong Scaling

Figure 5.1 shows how the betweenness centrality algorithm, implemented using the combinatorial BLAS primitives, scale for graphs of smaller size. Figure 5.2

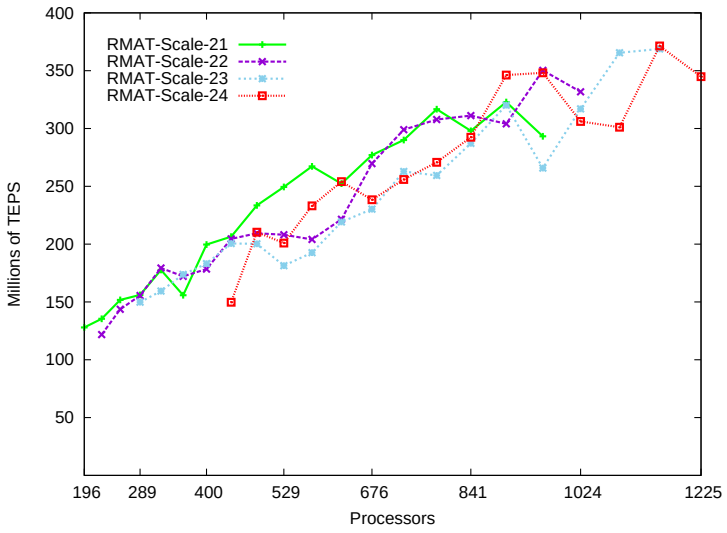


Figure 5.2: Parallel strong scaling of the distributed-memory betweenness centrality implementation (bigger input sizes)

shows the same algorithm on larger graphs, with larger number of processors. Both results show good scaling for this challenging tightly coupled algorithm. According to the best of our knowledge, ours are the first distributed memory performance results for betweenness centrality. The performance results on larger than 500 processors are not smooth, but the overall upward trend is clear. The heteregenous execution (soft errors, OS interrupts, etc.) on these large numbers of processors, probably causes the unsmooth timing results. The expensive computation prohibited us to run more experiments, which would have smoothed out the results by averaging.

The best performance results for this problem is due to Madduri et al. [144] using an optimized implementation specifically tailored for massively multithreaded architectures. They report a maximum of 160 million TEPS for an R-MAT graph of scale 24 on the 16-processor XMT machine. On the MTA-2 machine, which is the predecessor to the XMT, the same optimized code achieved 353 million TEPS on 40 processors. Our code, on the other hand, is truely generic and contains no problem and machine specific optimizations. We did not even attempt to optimize our primitives for the skewed aspect ratio (ratio of dimensions) of most of the matrices involved. For this problem instance, 900 processors of Lonestar was equivalent to 40 processors of MTA-2. The cost and power efficiency comparisons of these two solutions does not exist yet.

5.1.2 Sensitivity to Batch Processing

As mentioned earlier, most of the parallelism comes from the coarse-grained SpGEMM operation that is used to perform breadth-first searches from multiple source vertices. By changing the *batchsize*, the number of source vertices that are processed together, we obtain a trade-off between space usage and potential parallelism. The space increases linearly with increasing batchsize. As we show experimentally, the performance also increases substantially, especially for large number of processors. In Figure 5.3, we show the strong scaling of our betweenness

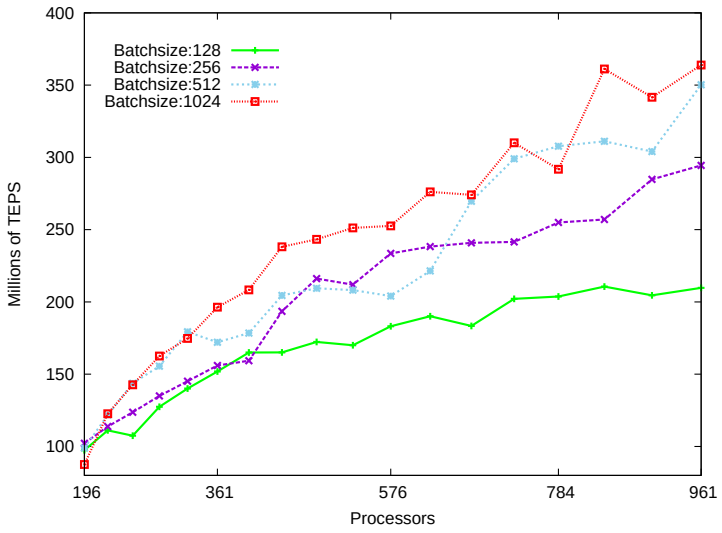


Figure 5.3: The effect of batch processing on the performance of the distributed-memory betweenness centrality implementation

centrality information on an RMAT graph of scale 22 (approximately 4 million vertices and 32 million edges), using different batchsizes. The average performance gain of using 256, 512 and 1024 starting vertices, over using 128 vertices, is 18.2%, 29.0%, and 39.7%, respectively. The average is computed over the performance on $p = \{196, 225, ..., 961\}$ (perfect squares) processors. For larger number of processors, the performance gain of using a large batchsize are more substantial. For example, for $p = 961$, the performance increases by 40.4%, 67.0%, and 73.5%, when using 256, 512 and 1024 starting vertices instead of 128.

5.2 Markov Clustering

Markov clustering (MCL) algorithm [82] is a flow based graph clustering algorithm that has been extremely popular in computational biology, among other fields. It simulates a Markov process to the point where clusters can be identified by simple interpretation of the modified adjacency matrix of the graph. Computationally, it alternates between an expansion step where the adjacency matrix is raised to its nth power (typically $n = 2$, so it is practically a squaring operation), and an inflation step where in every column of the adjacency matrix, the scalar entries are raised to the dth power ($d > 1$) and renormalized within the column. Inflation operation boosts the larger entries and effectively sends the smaller entries to close to zero. MCL achieves scalability and storage efficiency by maintaining sparsity of its matrix. This is done by pruning up close to zero entries after the inflation.

Implementing the MCL algorithm using the Combinatorial BLAS primitives generates a very concise code that feels natural. Full MCL code, except for the interpretation part, is shown in Figure 5.5, while the inflation subroutine is shown in Figure 5.4.

Van Dongen [202] provides a fast sequential implementation of the MCL algorithm. We do not attempt an apples-to-apples comparison with the original

```
template <typename IT, typename NT, typename DER>
void Inflate(SpParMat<IT,NT,DER> & A, double power)
{
    A.Apply(bind2nd(exponentiate(), power));

    /* reduce to Row, columns are collapsed to single entries */
    DenseParVec<IT,NT> colsums = Reduce(Row, plus<NT>(), 0.0);

    colsums.Apply(bind1st(divides<double>(), 1));

    /* scale each Column with the given row vector */
    A.DimScale(colsums, Column);
}
```

Figure 5.4: Inflation code using the Combinatorial BLAS primitives

```
int main()
{
    SpParMat<unsigned, double, SpDCCols<unsigned, double> > A;
    A.ReadDistribute(``inputmatrix'');

    oldchaos = Chaos(A);
    newchaos = oldchaos;

    // while there is an epsilon improvement
    while((oldchaos - newchaos) > EPS)
    {
        A.Square(); // expand
        Inflate(A,2);    // inflate (and renormalize)
        A.Prune(bind2nd(less<double>(), 0.0001));
        oldchaos = newchaos;
        newchaos = Chaos(A);
    }
    Interpret(A);
}
```

Figure 5.5: MCL code using the Combinatorial BLAS primitives

implementation, as the official `mcl` software has many options, something we can not replicate in our 10-15 lines prototype. The sequential `mcl` code is twice as fast as our parallel implementation running on a single processor. This is mostly due to its finer control over sparsity parameters, such as limiting the number of nonzeros in each row/column. Serial performance is not a bottleneck, as our code achieves superlinear speedup until $p = 1024$.

Figure 5.6 shows the speedup of the three most expensive iterations, that together make up more than 99% of the total running time. The input is a permuted R-MAT graph of scale 14, with self loops added for convergence. On 4096 processors, we were able to cluster this graph in less than a second. The same graph takes more than half an hour to cluster on a single processor. Note that iteration #4 takes only 70 milliseconds using 1024 processors, which is hard to scale further due to parallelization overheads on thousands of processors. We were able to cluster gigascale graphs using our implementation of MCL using the Combinatorial BLAS. We report on a smaller instance in order to provide a complete strong scaling result.

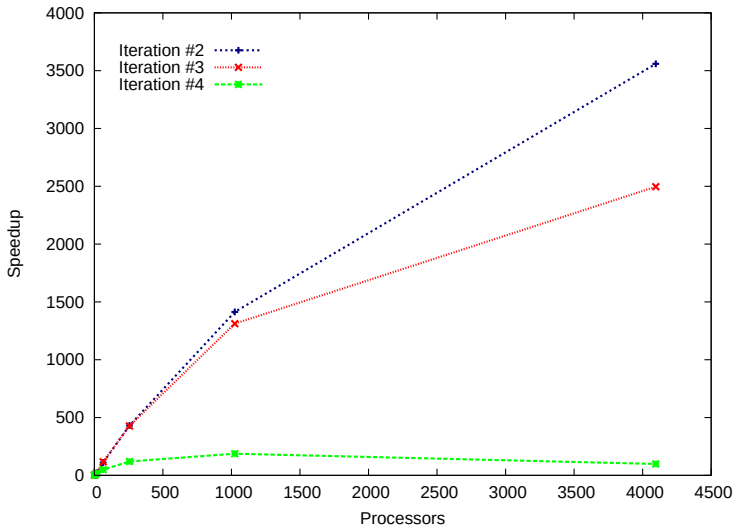


Figure 5.6: Strong scaling of the three most expensive iterations while clustering an R-MAT graph of scale 14 using the MCL algorithm implemented using the Combinatorial BLAS

Chapter 6

Parallel Sparse $y \leftarrow Ax$ and $y \leftarrow A^{\mathsf{T}}x$ Using Compressed Sparse Blocks

Abstract

This chapter introduces a storage format for sparse matrices, called ***compressed sparse blocks (CSB)***, which allows both Ax and $A^{\mathsf{T}}x$ to be computed efficiently in parallel, where A is an $n \times n$ sparse matrix with $nnz \geq n$ nonzeros and x is a dense n-vector. Our algorithms use $\Theta(nnz)$ work (serial running time) and $\Theta(\sqrt{n}\lg n)$ span (critical-path length), yielding a parallelism of $\Theta(nnz/\sqrt{n}\lg n)$, which is amply high for virtually any large matrix. The storage requirement for CSB is esssentially the same as that for the more-standard compressed-sparse-rows (CSR) format, for which computing Ax in parallel is easy but $A^{\mathsf{T}}x$ is difficult. Benchmark results indicate that on one processor, the CSB algorithms for Ax and $A^{\mathsf{T}}x$ run just as fast as the CSR algorithm for Ax, but the CSB algorithms also scale up linearly with processors until limited by off-chip memory bandwidth.

This chapter is based on a paper [47] by Buluç et al. from SPAA 2009. The indices are zero based, different from the rest of the thesis.

6.1 Introduction

When multiplying a large $n \times n$ sparse matrix A having nnz nonzeros by a dense n-vector x, the memory bandwidth for reading A can limit overall performance. Consequently, most algorithms to compute Ax store A in a compressed format. One simple "tuple" representation stores each nonzero of A as a triple consisting of its row index, its column index, and the nonzero value itself. This representation, however, requires storing $2\,nnz$ row and column indices, in addition to the nonzeros. The current standard storage format for sparse matrices in scientific computing, ***compressed sparse rows (CSR)*** [171], is more efficient, because it stores only $n+nnz$ indices or pointers. This reduction in storage of CSR compared with the tuple representation tends to result in faster serial algorithms.

In the domain of parallel algorithms, however, CSR has its limitations. Although CSR lends itself to a simple parallel algorithm for computing the matrix-vector product Ax, this storage format does not admit an efficient parallel algorithm for computing the product $A^{\mathsf{T}}x$, where A^{T} denotes the transpose of the matrix A — or equivalently, for computing the product $x^{\mathsf{T}}A$ of a row vector

x^{T} by A. Although one could use ***compressed sparse columns (CSC)*** to compute $A^{\mathsf{T}}x$, many applications, including iterative linear system solvers such as biconjugate gradients and quasi-minimal residual [171], require both Ax and $A^{\mathsf{T}}x$. One could transpose A explicitly, but computing the transpose for either CSR or CSC formats is expensive. Moreover, since matrix-vector multiplication for sparse matrices is generally limited by memory bandwidth, it is desirable to find a storage format for which both Ax and $A^{\mathsf{T}}x$ can be computed in parallel without performing more than nnz fetches of nonzeros from the memory to compute either product.

This paper presents a new storage format called ***compressed sparse blocks (CSB)*** for representing sparse matrices. Like CSR and CSC, the CSB format requires only $n+nnz$ words of storage for indices. Because CSB does not favor rows over columns or vice versa, it admits efficient parallel algorithms for computing either Ax or $A^{\mathsf{T}}x$, as well as for computing Ax when A is symmetric and only half the matrix is actually stored.

Previous work on parallel sparse matrix-vector multiplication has focused on reducing communication volume in a distributed-memory setting, often by using graph or hypergraph partitioning techniques to find good data distributions for particular matrices ([54, 203], for example). Good partitions generally exist for matrices whose structures arise from numerical discretizations of partial differen-

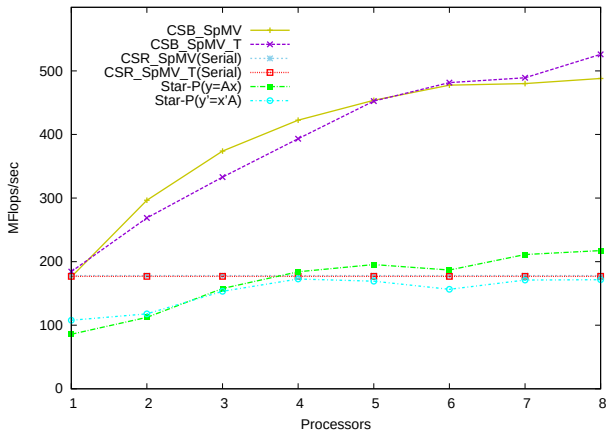


Figure 6.1: Average performance of Ax and $A^{\mathsf{T}}x$ operations on 13 different matrices from our benchmark test suite. CSB_SpMV and CSB_SpMV_T use compressed sparse blocks to perform Ax and $A^{\mathsf{T}}x$, respectively. CSR_SpMV (Serial) and CSR_SpMV_T (Serial) use OSKI [205] and compressed sparse rows without any matrix-specific optimizations. Star-P (y=Ax) and Star-P (y'=x'A) use Star-P [179], a parallel code based on CSR. The experiments were run on a ccNUMA architecture powered by AMD Opteron 8214 (Santa Rosa) processors.

tial equations in two or three spatial dimensions. Our work, by contrast, is motivated by multicore and manycore architectures, in which parallelism and memory bandwidth are key resources. Our algorithms are efficient in these measures for matrices with arbitrary nonzero structure.

Figure 6.1 presents an overall summary of achieved performance. The serial CSR implementation uses plain OSKI [205] without any matrix-specific optimiza-

tions. The graph shows the average performance over all our test matrices except for the largest, which failed to run on Star-P [179] due to memory constraints. The performance is measured in Mflops (Millions of FLoating-point OPerationS) per second. Both Ax and $A^{\mathsf{T}}x$ take 2 nnz flops. To measure performance, we divide this value by the time it takes for the computation to complete. Section 6.7 provides detailed performance results.

The remainder of this paper is organized as follows. Section 6.2 discusses the limitations of the CSR/CSC formats for parallelizing Ax and $A^{\mathsf{T}}x$ calculations. Section 6.3 describes the CSB format for sparse matrices. Section 6.4 presents the algorithms for computing Ax and $A^{\mathsf{T}}x$ using the CSB format, and Section 6.5 provides a theoretical analysis of their parallel performance. Section 6.6 describes the experimental setup we used, and Section 6.7 presents the results. Section 6.8 offers some concluding remarks.

6.2 Conventional storage formats

This section describes the CSR and CSC sparse-matrix storage formats and explores their limitations when it comes to computing both Ax and $A^{\mathsf{T}}x$ in parallel. We review the work/span formulation of parallelism and show that performing Ax with CSR (or equivalently $A^{\mathsf{T}}x$ with CSC) yields ample parallelism. We consider

various strategies for performing $A^{\mathsf{T}}x$ in parallel with CSR (or equivalently Ax with CSC) and why they are problematic.

The compressed sparse row (CSR) format stores the nonzeros (and ideally only the nonzeros) of each matrix row in consecutive memory locations, and it stores an index to the first stored element of each row. In one popular variant [77], CSR maintains one floating-point array $val[nnz]$ and two integer arrays, $col\_ind[nnz]$ and $row\_ptr[n]$ to store the matrix $A = (a_{ij})$. The row_ptr array stores the index of each row in val. That is, if $val[k]$ stores matrix element a_{ij}, then $row\_ptr[i] \leq k < row\_ptr[i+1]$. The col_ind array stores the column indices of the elements in the val array. That is, if $val[k]$ stores matrix element a_{ij}, then $col\_ind[k] = j$.

The compressed sparse column (CSC) format is analogous to CSR, except that the nonzeros of each column, instead of row, are stored in contiguous memory locations. In other words, the CSC format for A is obtained by storing A^{T} in CSR format.

The earliest written description of CSR that we have been able to divine from the literature is an unnamed "scheme" presented in Table 1 of the 1967 article [195] by Tinney and Walker, although in 1963 Sato and Tinney [175] allude to what is probably CSR. Markowitz's seminal paper [148] on sparse Gaussian elimination does not discuss data structures, but it is likely that Markowitz used such a

```
CSR_SpMV(A, x, y)
1  n ← A.rows
2  for i ← 0 to n − 1 in parallel
3       do y[i] ← 0
4          for k ← A.row_ptr[i] to A.row_ptr[i + 1] − 1
5               do y[i] ← y[i] + A.val[k] · x[A.col_ind[k]]
```

Figure 6.2: Parallel procedure for computing $y \leftarrow Ax$, where the $n \times n$ matrix A is stored in CSR format.

format as well. CSR and CSC have since become ubiquitous in sparse matrix computation [74, 83, 86, 102, 104, 171].

The following lemma states the well-known bound on space used by the index data in the CSR format (and hence the CSC format as well). By index data, we mean all data other than the nonzeros — that is, the *row_ptr* and *col_ind* arrays.

Lemma 1. *The CSR format uses $n \lg nnz + nnz \lg n$ bits of index data for an $n \times n$ matrix.*

For a CSR matrix A, computing $y \leftarrow Ax$ in parallel is straightforward, as shown in Figure 6.2. Procedure CSR_SpMV in the figure computes each element of the output array in parallel, and it does not suffer from race conditions, because each parallel iteration i writes to a single location $y[i]$ which is not updated by any other iteration.

We shall measure the complexity of this code, and other codes in this paper, in terms of ***work*** and ***span*** [69, Ch. 27]:

- The ***work***, denoted by T_1, is the running time on 1 processor.
- The ***span***,[1] denoted by T_∞, is running time on an infinite number of processors.

The ***parallelism*** of the algorithm is T_1/T_∞, which corresponds to the maximum possible speedup on any number of processors. Generally, if a machine has somewhat fewer processors than the parallelism of an application, a good scheduler should be able to achieve linear speedup. Thus, for a fixed amount of work, our goal is to achieve a sufficiently small span so that the parallelism exceeds the number of processors by a reasonable margin.

The work of CSR_SpMV is $\Theta(nnz)$, assuming, as we shall, that $nnz \geq n$, because the body of the outer loop starting in line 2 executes for n iterations, and the body of the inner loop starting in line 4 executes for the number of nonzeros in the ith row, for a total of nnz times.

The span of CSR_SpMV depends on the maximum number nr of nonzeros in any row of the matrix A, since that number determines the worst-case time of any iteration of the loop in line 4. The n iterations of the parallel loop in line 2

[1] The literature also uses the terms ***depth*** [34] and ***critical-path length*** [35].

contribute $\Theta(\lg n)$ to the span, assuming that loops are implemented as binary recursion. Thus, the total span is $\Theta(nr + \lg n)$.

The parallelism is therefore $\Theta(nnz\,/(nr + \lg n))$. In many common situations, we have $nnz = \Theta(n)$, which we will assume for estimation purposes. The maximum number nr of nonzeros in any row can vary considerably, however, from a constant, if all rows have an average number of nonzeros, to n, if the matrix has a dense row. If $nr = O(1)$, then the parallelism is $\Theta(nnz\,/\lg n)$, which is quite high for a matrix with a billion nonzeros. In particular, if we ignore constants for the purpose of making a ballpark estimate, we have $nnz\,/\lg n \approx 10^9/(\lg 10^9) > 3\times10^7$, which is much larger than any number of processors one is likely to encounter in the near future. If $nr = \Theta(n)$, however, as is the case when there is even a single dense row, we have parallelism $\Theta(nnz\,/n) = \Theta(1)$, which limits scalability dramatically. Fortunately, we can parallelize the inner loop (line 4) using divide-and-conquer recursion to compute the sparse inner product in $\lg(nr)$ span without affecting the asymptotic work, thereby achieving parallelism $\Theta(nnz\,/\lg n)$ in all cases.

Computing $A^{\mathsf{T}}x$ serially can be accomplished by simply interchanging the row and column indices [79], yielding the pseudocode shown in Figure 6.3. The work of procedure CSR_SpMV_T is $\Theta(nnz)$, the same as CSR_SpMV.

```
CSR_SpMV_T(A, x, y)
1  n ← A.cols
2  for i ← 0 to n − 1
3        do y[i] ← 0
4  for i ← 0 to n − 1
5        do for k ← A.row_ptr[i] to A.row_ptr[i + 1] − 1
6        do y[A.col_ind[k]] ← y[A.col_ind[k]] + A.val[k] · x[i]
```

Figure 6.3: Serial procedure for computing $y \leftarrow A^{\mathsf{T}}x$, where the $n \times n$ matrix A is stored in CSR format.

Parallelizing CSR_SpMV_T is not straightforward, however. We shall review several strategies to see why it is problematic.

One idea is to parallelize the loops in lines 2 and 5, but this strategy yields minimal scalability. First, the span of the procedure is $\Theta(n)$, due to the loop in line 4. Thus, the parallelism can be at most $O(nnz\,/n)$, which is a small constant in most common situations. Second, in any practical system, the communication and synchronization overhead for executing a small loop in parallel is much larger than the execution time of the few operations executed in line 6.

Another idea is to execute the loop in line 4 in parallel. Unfortunately, this strategy introduces race conditions in the read/modify/write to $y[A.\,col\_ind[k]]$ in line 6.[2] These races can be addressed in two ways, neither of which is satisfactory.

[2]In fact, if $nnz > n$, then the "pigeonhole principle" guarantees that the program has at least one race condition.

The first solution involves locking column $col\_ind[k]$ or using some other form of atomic update.[3] This solution is unsatisfactory because of the high overhead of the lock compared to the cost of the update. Moreover, if A contains a dense column, then the contention on the lock is $\Theta(n)$, which completely destroys any parallelism in the common case where $nnz = \Theta(n)$.

The second solution involves splitting the output array y into multiple arrays y_p in a way that avoids races, and then accumulating $y \leftarrow \Sigma_p y_p$ at the end of the computation. For example, in a system with P processors (or threads), one could postulate that processor p only operates on array y_p, thereby avoiding any races. This solution is unsatisfactory because the work becomes $\Theta(nnz + Pn)$, where the last term comes from the need to initialize and accumulate P (dense) length-n arrays. Thus, the parallel execution time is $\Theta((nnz + Pn)/P) = \Omega(n)$ no matter how many processors are available.

A third idea for parallelizing $A^{\mathsf{T}}x$ is to compute the transpose explicitly and then use CSR_SpMV. Unfortunately, parallel transposition of a sparse matrix in CSR format is costly and encounters exactly the same problems we are trying to avoid. Moreover, every element is accessed at least twice: once for the transpose, and once for the multiplication. Since the calculation of a matrix-vector product

[3]No mainstream hardware supports atomic update of floating-point quantities, however.

tends to be memory-bandwidth limited, this strategy is generally inferior to any strategy that accesses each element only once.

Finally, of course, we could store the matrix A^{T} in CSR format, that is, storing A in CSC format, but then computing Ax becomes difficult.

To close this section, we should mention that if the matrix A is symmetric, so that only about half the nonzeros need be stored — for example, those on or above the diagonal — then computing Ax in parallel for CSR is also problematic. For this example, the elements below the diagonal are visited in an inconvenient order, as if they were stored in CSC format.

6.3 The CSB storage format

This section describes the CSB storage format for sparse matrices and shows that it uses the same amount of storage space as the CSR and CSC formats. We also compare CSB to other blocking schemes.

For a given ***block-size parameter*** β, CSB partitions the $n \times n$ matrix A into n^2/β^2 equal-sized $\beta \times \beta$ square ***blocks***[4]

$$A = \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,n/\beta-1} \\ A_{10} & A_{11} & \cdots & A_{1,n/\beta-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n/\beta-1,0} & A_{n/\beta-1,1} & \cdots & A_{n/\beta-1,n/\beta-1} \end{pmatrix},$$

[4]The CSB format may be easily extended to nonsquare $n \times m$ matrices. In this case, the blocks remain as square $\beta \times \beta$ matrices, and there are nm/β^2 blocks.

where the block A_{ij} is the $\beta \times \beta$ submatrix of A containing elements falling in rows $i\beta, i\beta+1, \ldots, (i+1)\beta-1$ and columns $j\beta, j\beta+1, \ldots, (j+1)\beta-1$ of A. For simplicity of presentation, we shall assume that β is an exact power of 2 and that it divides n; relaxing these assumptions is straightforward.

Many or most of the individual blocks A_{ij} are ***hypersparse*** [49], meaning that the ratio of nonzeros to matrix dimension is asymptotically 0. For example, if $\beta = \sqrt{n}$ and $nnz = cn$, the average block has dimension $\sqrt{n}$ and only c nonzeros. The space to store a block should therefore depend only on its nonzero count, not on its dimension.

CSB represents a block A_{ij} by compactly storing a triple for each nonzero, associating with the nonzero data element a row and column index. In contrast to the column index stored for each nonzero in CSR, the row and column indices lie within the submatrix A_{ij}, and hence require fewer bits. In particular, if $\beta = \sqrt{n}$, then each index into A_{ij} requires only half the bits of an index into A. Since these blocks are stored contiguously in memory, CSB uses an auxiliary array of pointers to locate the beginning of each block.

More specifically, CSB maintains a floating-point array $val[nnz]$, and three integer arrays $row\_ind[nnz]$, $col\_ind[nnz]$, and $blk\_ptr[n^2/\beta^2]$. We describe each of these arrays in turn.

The *val* array stores all the nonzeros of the matrix and is analogous to CSR's array of the same name. The difference is that CSR stores *rows* contiguously, whereas CSB stores *blocks* contiguously. Although each block must be contiguous, the ordering among blocks is flexible. Let $f(i, j)$ be the bijection from pairs of block indices to integers in the range $0, 1, \ldots, n^2/\beta^2 - 1$ that describes the ordering among blocks. That is, $f(i, j) < f(i', j')$ if and only if A_{ij} appears before $A_{i'j'}$ in *val*. We discuss choices of ordering later in this section.

The *row_ind* and *col_ind* arrays store the row and column indices, respectively, of the elements in the *val* array. These indices are relative to the block containing the particular element, not the entire matrix, and hence they range from 0 to $\beta - 1$. That is, if $val[k]$ stores the matrix element $a_{i\beta+r,j\beta+c}$, which is located in the rth row and cth column of the block A_{ij}, then $row\_ind = r$ and $col\_ind = c$. As a practical matter, we can pack a corresponding pair of elements of *row_ind* and *col_ind* into a single integer word of $2 \lg \beta$ bits so that they make a single array of length *nnz*, which is comparable to the storage needed by CSR for the *col_ind* array.

The *blk_ptr* array stores the index of each block in the *val* array, which is analogous to the *row_ptr* array for CSR. If $val[k]$ stores a matrix element falling in block A_{ij}, then $blk\_ptr[f(i, j)] \leq k < blk\_ptr[f(i, j) + 1]$.

The following lemma states the storage used for indices in the CSB format.

Lemma 2. *The CSB format uses $(n^2/\beta^2) \lg nnz + 2\, nnz \lg \beta$ bits of index data.*

Proof. Since the *val* array contains *nnz* elements, referencing an element requires $\lg nnz$ bits, and hence the *blk_ptr* array uses $(n^2/\beta^2) \lg nnz$ bits of storage.

For each element in *val*, we use $\lg \beta$ bits to represent the row index and $\lg \beta$ bits to represent the column index, requiring a total of $nnz \lg \beta$ bits for each of *row_ind* and *col_ind*. Adding the space used by all three indexing arrays completes the proof. □

To better understand the storage requirements of CSB, we present the following corollary for $\beta = \sqrt{n}$. In this case, both CSR (Lemma 1) and CSB use the same storage.

Corollary 3. *The CSB format uses $n \lg nnz + nnz \lg n$ bits of index data when $\beta = \sqrt{n}$.* □

Thus far, we have not addressed the ordering of elements within each block or the ordering of blocks. Within a block, we use a Z-Morton ordering [151], storing first all those elements in the top-left quadrant, then the top-right, bottom-left, and finally bottom-right quadrants, using the same layout recursively within each quadrant. In fact, these quadrants may be stored in any order, but the recursive ordering is necessary for our algorithm to achieve good parallelism within a block.

The choice of storing the nonzeros within blocks in a recursive layout is opposite to the common wisdom for storing dense matrices [87]. Although most compilers and architectures favor conventional row/column ordering for optimal prefetching, the choice of layout within the block becomes less significant for sparse blocks as they already do not take full advantage of such features. More importantly, a recursive ordering allows us to efficiently determine the four quadrants of a block using binary search, which is crucial for parallelizing individual blocks.

Our algorithm and analysis do not, however, require any particular ordering among blocks. A Z-Morton ordering (or any recursive ordering) seems desirable as it should get better performance in practice by providing spatial locality, and it matches the ordering within a block. Computing the function $f(i, j)$, however, is simpler for a row-major or column-major ordering among blocks.

Comparison with other blocking methods

A blocked variant of CSR, called ***BCSR***, has been used for improving register reuse [122]. In BCSR, the sparse matrix is divided into small dense blocks that are stored in consecutive memory locations. The pointers are maintained to the first block on each row of blocks. BCSR storage is converse to CSB storage, because BCSR stores a sparse collection of dense blocks, whereas CSB stores a

dense collection of sparse blocks. We conjecture that it would be advantageous to apply BCSR-style register blocking to each individual sparse block of CSB.

Nishtala *et al.* [154] have proposed a data structure similar to CSB in the context of cache blocking. Our work differs from theirs in two ways. First, CSB is symmetric without favoring rows over columns. Second, our algorithms and analysis for CSB are designed for parallelism instead of cache performance. As shown in Section 6.5, CSB supports ample parallelism for algorithms computing Ax and $A^{\mathsf{T}}x$, even on sparse and irregular matrices.

Blocking is also used in dense matrices. The Morton-hybrid layout [5, 142], for example, uses a parameter equivalent to our parameter β for selecting the block size. Whereas in CSB we store elements in a Morton ordering within blocks and an arbitrary ordering among blocks, the Morton-hybrid layout stores elements in row-major order within blocks and a Morton ordering among blocks. The Morton-hybrid layout is designed to take advantage of hardware and compiler optimizations (within a block) while still exploiting the cache benefits of a recursive layout. Typically the block size is chosen to be 32×32, which is significantly smaller than the $\Theta(\sqrt{n})$ block size we propose for CSB. The Morton-hybrid layout, however, considers only dense matrices, for which designing a matrix-vector multiplication algorithm with good parallelism is significantly easier.

6.4 Matrix-vector multiplication using CSB

This section describes a parallel algorithm for computing the sparse-matrix dense-vector product $y \leftarrow Ax$, where A is stored in CSB format. This algorithm can be used equally well for computing $y \leftarrow A^{\mathsf{T}}x$ by switching the roles of row and column. We first give an overview of the algorithm and then describe it in detail.

At a high level, the CSB_SpMV multiplication algorithm simply multiplies each "blockrow" by the vector x in parallel, where the ith ***blockrow*** is the row of blocks $(A_{i0}A_{i1}\cdots A_{i,n/\beta-1})$. Since each blockrow multiplication writes to a different portion of the output vector, this part of the algorithm contains no races due to write conflicts.

If the nonzeros were guaranteed to be distributed evenly among block rows, then the simple blockrow parallelism would yield an efficient algorithm with n/β-way parallelism by simply performing a serial multiplication for each blockrow. One cannot, in general, guarantee that distribution of nonzeros will be so nice, however. In fact, sparse matrices in practice often include at least one dense row containing roughly n nonzeros, whereas the number of nonzeros is only $nnz \approx cn$ for some small constant c. Thus, performing a serial multiplication for each blockrow yields no better than c-way parallelism.

To make the algorithm robust to matrices of arbitrary nonzero structure, we must parallelize the blockrow multiplication when a blockrow contains "too many" nonzeros. This level of parallelization requires care to avoid races, however, because two blocks in the same blockrow write to the same region within the output vector. Specifically, when a blockrow contains $\Omega(\beta)$ nonzeros, we recursively divide it "in half," yielding two subblockrows, each containing roughly half the nonzeros. Although each of these subblockrows can be multiplied in parallel, they may need to write to the same region of the output vector. To avoid the races that might arise due to write conflicts between the subblockrows, we allocate a temporary vector to store the result of one of the subblockrows and allow the other subblockrow to use the output vector. After both subblockrow multiplications complete, we serially add the temporary vector into the output vector.

To facilitate fast subblockrow divisions, we first partition the blockrow into "chunks" of consecutive blocks, each containing at most $O(\beta)$ nonzeros (when possible) and $\Omega(\beta)$ nonzeros on average. The lower bound of $\Omega(\beta)$ will allow us to amortize the cost of writing to the length-β temporary vector against the nonzeros in the chunk. By dividing a blockrow "in half," we mean assigning to each subblockrow roughly half the chunks.

Figure 6.4 gives the top-level algorithm, performing each blockrow vector multiplication in parallel. The notation $x[a\,..\,b]$ means the subarray of x starting

```
CSB_SpMV(A, x, y)
 1  for i ← 0 to n/β − 1 in parallel                    ▷ For each blockrow.
 2        do Initialize a dynamic array R_i
 3           R_i[0] ← 0
 4           count ← 0                                  ▷ Count nonzeroes in chunk.
 5           for j ← 0 to n/β − 2
 6                 do count ← count + nnz(A_ij)
 7                    if count + nnz(A_i,j+1) > Θ(β)
 8                       then ▷ End the chunk, since the next block
                              ▷ makes it too large.
 9                            append j to R_i           ▷ Last block in chunk.
10                            count ← 0
11  append n/β − 1 to R_i
12  CSB_BlockrowV(A, i, R_i, x, y[iβ .. (i + 1)β − 1])
```

Figure 6.4: Pseudocode for the matrix-vector multiplication $y \leftarrow Ax$. The procedure CSB_BlockrowV (pseudocode for which can be found in Figure 6.5) as called here multiplies the blockrow by the vector x and writes the output into the appropriate region of the output vector y.

at index a and ending at index b. The function $nnz(A_{ij})$ is a shorthand for $A.\,blk\_ptr[f(i,j)+1] - A.\,blk\_ptr[f(i,j)]$, which calculates the number of nonzeros in the block A_{ij}. For conciseness, we have overloaded the $\Theta(\beta)$ notation (in line 7) to mean "a constant times β"; any constant suffices for the analysis, and we use the constant 3 in our implementation. The "**for** ... **in parallel do**" construct means that each iteration of the **for** loop may be executed in parallel with the others.

For each loop iteration, we partition the blockrow into chunks in lines 2–11 and then call the blockrow multiplication in line 12. The array R_i stores the indices of the last block in each chunk; specifically, the kth chunk, for $k > 0$, includes blocks $(A_{i,R_i[k-1]+1} A_{i,R_i[k-1]+2} \cdots A_{i,R_i[k]})$. A chunk consists of either a single block containing $\Omega(\beta)$ nonzeros, or it consists of many blocks containing $O(\beta)$ nonzeros in total. To compute chunk boundaries, just iterate over blocks (in lines 5–10) until enough nonzeros are accrued.

Figure 6.5 gives the parallel algorithm CSB_BlockrowV for multiplying a blockrow by a vector, writing the result into the length-β vector y. The **in parallel do** ... **do** ... construct indicates that all of the **do** code blocks may execute in parallel. The procedure CSB_BlockV (pseudocode for which can be found in Figure 6.6) calculates the product of the block and the vector in parallel. In lines 12–19 of CSB_BlockrowV, the algorithm recursively divides the

```
CSB_BlockrowV(A, i, R, x, y)
 1  if R.length = 2                          ▷ The subblockrow is a single chunk.
 2     then ℓ ← R[0] + 1                                  ▷ Leftmost block in chunk.
 3          r ← R[1]                                      ▷ Rightmost block in chunk.
 4          if ℓ = r
 5             then ▷ The chunk is a single (dense) block.
 6                  start ← A.blk_ptr[f(i, ℓ)]
 7                  end ← A.blk_ptr[f(i, ℓ) + 1] − 1
 8                  CSB_BlockV(A, start, end, β, x, y)
 9             else ▷ The chunk is sparse.
10                  multiply y ← (A_iℓ A_{i,ℓ+1} ··· A_ir)x serially
11  return
    ▷ Since the block row is "dense," split it in half.
12  mid ← ⌈R.length/2⌉ − 1                             ▷ Divide chunks in half.
    ▷ Calculate the dividing point in the input vector x.
13  xmid ← β · (R[mid] − R[0])
14  allocate a length-β temporary vector z, initialized to 0
15  in parallel
16        do CSB_BlockrowV(A, i, R[0 .. mid], x[0 .. xmid − 1], y)
17        do CSB_BlockrowV(A, i, R[mid .. R.length − 1],
                                  x[xmid .. x.length − 1], z)
18  for k ← 0 to β − 1
19        do y[k] ← y[k] + z[k]
```

Figure 6.5: Pseudocode for the subblockrow vector product $y \leftarrow (A_{i\ell}A_{i,\ell+1}\cdots A_{ir})x$.

blockrow such that each half receives roughly the same number of chunks. We find the appropriate middles of the chunk array R and the input vector x in lines 12 and 13, respectively. We then allocate a length-β temporary vector z (line 14) and perform the recursive multiplications on each subblockrow in parallel (lines 15–17), having one of the recursive multiplications write its output to z. When these recursive multiplications complete, we merge the outputs into the vector y (lines 18–19).

The recursion bottoms out when the blockrow consists of a single chunk (lines 2–11). If this chunk contains many blocks, it is guaranteed to contain at most $\Theta(\beta)$ nonzeros, which is sufficiently sparse to perform the serial multiplication in line 10. If, on the other hand, the chunk is a single block, it may contain as many as $\beta^2 \approx n$ nonzeros. A serial multiplication here, therefore, would be the bottleneck in the algorithm. Instead, we perform the parallel block-vector multiplication CSB_BlockV in line 8.

If the blockrow recursion reaches a single block, we perform a parallel multiplication of the block by the vector. Pseudocode for the subblock-vector product $y \leftarrow Mx$, is shown in Figure 6.6, where M is the list of tuples stored in $A.val[start\,..\,end]$, $A.row\_ind[start\,..\,end]$, and $A.col\_ind[start\,..\,end]$ in Z-morton order. The & operator is a bitwise AND of the two operands. The block-vector multiplication proceeds by recursively dividing the (sub)block M into

quadrants M_{00}, M_{01}, M_{10}, and M_{11}, each of which is conveniently stored contiguously in the Z-Morton-ordered *val*, *row_ind*, and *col_ind* arrays between indices *start* and *end*. We perform binary searches to find the appropriate dividing points in the array in lines 7–9.

To understand the pseudocode, consider the search for the dividing point s_2 between $M_{00}M_{01}$ and $M_{10}M_{11}$. For any recursively chosen $dim \times dim$ matrix M, the column indices and row indices of all elements have the same leading $\lg \beta - \lg dim$ bits. Moreover, for those elements in $M_{00}M_{01}$, the next bit in the row index is a 0, whereas for those in elements in $M_{10}M_{11}$, the next bit in the row index is 1. The algorithm does a binary search for the point at which this bit flips. The cases for the dividing point between M_{00} and M_{01} or M_{10} and M_{11} are similar, except that we focus on the column index instead of the row index.

After dividing the matrix into quadrants, we execute the matrix products involving matrices M_{00} and M_{11} in parallel (lines 10–12), as they do not conflict on any outputs. After completing these products, we execute the other two matrix products in parallel (lines 13–15).[5] This procedure resembles a standard parallel divide-and-conquer matrix multiplication, except that our base case of serial multiplication starts at a matrix containing $\Theta(dim)$ nonzeros (lines 2–5). Note that

[5]The algorithm may instead do M_{00} and M_{10} in parallel followed by M_{01} and M_{11} in parallel without affecting the performance analysis. Presenting the algorithm with two choices may yield better load balance.

```
CSB_BlockV(A, start, end, dim, x, y)
    ▷ A.val[start..end] is a dim × dim matrix M.
 1  if end − start ≤ Θ(dim)
 2     then ▷ Perform the serial computation y ← y + Mx.
 3          for k ← start to end
 4               do y[A.row_ind[k]] ← y[A.row_ind[k]]
                                      +A.val[k] · x[A.col_ind[k]]
 5  return
 6  ▷ Recurse. Find the indices of the quadrants.
 7  binary search start, start +1, ..., end for the smallest s2
          such that (A.row_ind[s2] & dim /2) ≠ 0
 8  binary search start, start +1, ..., s2 − 1 for the smallest s1
          such that (A.col_ind[s1] & dim /2) ≠ 0
 9  binary search s2, s2 + 1, ..., end for the smallest s3
          such that (A.col_ind[s3] & dim /2) ≠ 0
10  in parallel
11       do CSB_BlockV(A, start, s1 − 1, dim /2, x, y)        ▷ M00.
12       do CSB_BlockV(A, s3, end, dim /2, x, y)              ▷ M11.
13  in parallel
14       do CSB_BlockV(A, s1, s2 − 1, dim /2, x, y)           ▷ M01.
15       do CSB_BlockV(A, s2, s3 − 1, dim /2, x, y)           ▷ M10.
```

Figure 6.6: Pseudocode for the subblock-vector product $y \leftarrow Mx$.

although we pass the full length-β arrays x and y to each recursive call, the effective length of each array is halved implicitly by partitioning M into quadrants. Passing the full arrays is a technical detail required to properly compute array indices, as the indices $A.\mathit{row\_ind}$ and $A.\mathit{col\_ind}$ store offsets within the block.

The CSB_SpMV_T algorithm is identical to CSB_SpMV, except that we operate over blockcolumns rather than blockrows.

6.5 Analysis

In this section, we prove that for an $n \times n$ matrix with nnz nonzeros, CSB_SpMV operates with work $\Theta(nnz)$ and span $O(\sqrt{n}\lg n)$ when $\beta = \sqrt{n}$, yielding a parallelism of $\Omega(nnz\,/\sqrt{n}\lg n)$. We also provide bounds in terms of β and analyze the space usage.

We begin by analyzing block-vector multiplication.

Lemma 4. *On a $\beta \times \beta$ block containing r nonzeros,* CSB_BlockV *runs with work $\Theta(r)$ and span $O(\beta)$.*

Proof. The span for multiplying a $\mathit{dim} \times \mathit{dim}$ matrix can be described by the recurrence $S(\mathit{dim}) = 2S(\mathit{dim}\,/2) + O(\lg \mathit{dim}) = O(\mathit{dim})$. The $\lg \mathit{dim}$ term represents a loose upper bound on the cost of the binary searches. In particular, the

binary-search cost is $O(\lg z)$ for a submatrix containing z nonzeros, and we have $z \leq dim^2$, and hence $O(\lg z) = O(\lg dim)$, for a $dim \times dim$ matrix.

To calculate the work, consider the degree-4 tree of recursive procedure calls, and associate with each node the work done by that procedure call. We say that a node in the tree has height h if it corresponds to a $2^h \times 2^h$ subblock, i.e., if $dim = 2^h$ is the parameter passed into the corresponding CSB_BlockV call. Node heights are integers ranging from 0 to $\lg \beta$. Observe that each height-h node corresponds to a distinct $2^h \times 2^h$ subblock (although subblocks may overlap for nodes having different heights). A height-h leaf node (serial base case) corresponds to a subblock containing at most $z = O(2^h)$ nonzeros and has work linear in this number z of nonzeros. Summing across all leaves, therefore, gives $\Theta(r)$ work. A height-h internal node, on the other hand, corresponds to a subblock containing *at least* $z' = \Omega(2^h)$ nonzeros (or else it would not recurse further and be a leaf) and has work $O(\lg 2^h) = O(h)$ arising from the binary searches. There can thus be at most $O(r/2^h)$ height-h internal nodes having total work $O((r/2^h)h)$. Summing across all heights gives total work of $\sum_{h=0}^{\lg \beta} O((r/2^h)h) = r \sum_{h=0}^{\lg \beta} O(h/2^h) = O(r)$ for internal nodes. Combining the work at internal nodes and leaf nodes gives total work $\Theta(r)$. □

The next lemma analyzes blockrow-vector multiplication.

Lemma 5. *On a blockrow containing n/β blocks and r nonzeros,* CSB_BlockrowV *runs with work $\Theta(r)$ and span $O(\beta \lg(n/\beta))$.*

Proof. Consider a call to CSB_BlockrowV on a row that is partitioned into C chunks, and let $W(C)$ denote the work. The work per recursive call on a multichunk subblockrow is dominated by the $\Theta(\beta)$ work of initializing a temporary vector z and adding the vector z into the output vector y. The work for a CSB_BlockrowV on a single-chunk subblockrow is linear in the number of nonzeros in the chunk. (We perform linear work either in line 10 or in line 8 — see Lemma 4 for the work of line 8.) We can thus describe the work by the recurrence $W(C) \leq 2W(\lceil C/2 \rceil) + \Theta(\beta)$ with a base case of work linear in the nonzeros, which solves to $W(C) = \Theta(C\beta + r)$ for $C > 1$. When $C = 1$, we have $W(C) = \Theta(r)$, as we do not operate on the temporary vector z.

To bound work, it remains to bound the maximum number of chunks in a row. Notice that any two consecutive chunks contain at least $\Omega(\beta)$ nonzeros. This fact follows from the way chunks are chosen in lines 2–11: a chunk is terminated only if adding the next block to the chunk would increase the number of nonzeros to more than $\Theta(\beta)$. Thus, a blockrow consists of a single chunk whenever $r = O(\beta)$ and at most $O(r/\beta)$ chunks whenever $r = \Omega(\beta)$. Hence, the total work is $\Theta(r)$.

We can describe the span of CSB_BlockrowV by the recurrence $S(C) = S(\lceil C/2 \rceil) + O(\beta) = O(\beta \lg C) + S(1)$. The base case involves either serially mul-

tiplying a single chunk containing at most $O(\beta)$ nonzeros in line 10, which has span $O(\beta)$, or multiplying a single block in parallel in line 8, which also has span $O(\beta)$ from Lemma 4. We have, therefore, a span of $O(\beta \lg C) = O(\beta \lg(n/\beta))$, since $C \leq n/\beta$. □

We are now ready to analyze matrix-vector multiplication itself.

Theorem 6. *On an $n \times n$ matrix containing nnz nonzeros,* CSB_SpMV *runs with work $\Theta(n^2/\beta^2 + nnz)$ and span $O(\beta \lg(n/\beta) + n/\beta)$.*

Proof. For each blockrow, we add $\Theta(n/\beta)$ work and span for computing the chunks, which arise from a serial scan of the n/β blocks in the blockrow. Thus, the total work is $O(n^2/\beta^2)$ in addition to the work for multiplying the blockrows, which is linear in the number of nonzeros from Lemma 5.

The total span is $O(\lg(n/\beta))$ to parallelize all the rows, plus $O(n/\beta)$ per row to partition the row into chunks, plus the $O(\beta \lg(n/\beta))$ span per blockrow from Lemma 5. □

The following corollary gives the work and span bounds when we choose β to yield the same space for the CSB storage format as for the CSR or CSC formats.

Corollary 7. *On an $n \times n$ matrix containing $nnz \geq n$ nonzeros, by choosing $\beta = \Theta(\sqrt{n})$,* CSB_SpMV *runs with work $\Theta(nnz)$ and span $O(\sqrt{n} \lg n)$, achieving a parallelism of $\Omega(nnz / \sqrt{n} \lg n)$.* □

Since CSB_SpMV_T is isomorphic to CSB_SpMV, we obtain the following corollary.

Corollary 8. *On an $n \times n$ matrix containing $nnz \geq n$ nonzeros, by choosing $\beta = \Theta(\sqrt{n})$,* CSB_SpMV_T *runs with work $\Theta(nnz)$ and span $O(\sqrt{n}\lg n)$, achieving a parallelism of $\Omega(nnz\,/\sqrt{n}\lg n)$.* □

The work of our algorithm is dominated by the space of the temporary vectors z, and thus the space usage on an infinite number of processors matches the work bound. When run on fewer processors however, the space usage reduces drastically. We can analyze the space in terms of the ***serialization*** of the program, which corresponds to the program obtained by removing all **parallel** keywords.

Lemma 9. *On an $n \times n$ matrix, by choosing $\beta = \Theta(\sqrt{n})$, the serialization of* CSB_SpMV *requires $O(\sqrt{n}\lg n)$ space (not counting the storage for the matrix itself).*

Proof. The serialization executes one blockrow multiplication at a time. There are two space overheads. First, we use $O(n/\beta) = O(\sqrt{n})$ space for the chunk array. Second, we use β space to store the temporary vector z for each outstanding recursive call to CSB_BlockrowV. Since the recursion depth is $O(\lg n)$, the total space becomes $O(\beta \lg n) = O(\sqrt{n}\lg n)$. □

A typical work-stealing scheduler executes the program in a depth-first (serial) manner on each processor. When a processor completes all its work, it "steals" work from a different processor, beginning a depth-first execution from some unexecuted parallel branch. Although not all work-stealing schedulers are space efficient, those maintaining the ***busy-leaves property*** [36] (e.g., as used in the Cilk work-stealing scheduler [35]) are space efficient. The "busy-leaves" property roughly says that if a procedure has begun (but not completed) executing, then there exists a processor currently working on that procedure or one of its descendants procedures.

Corollary 10. *Suppose that a work-stealing scheduler with the busy-leaves property schedules an execution of* CSB_SpMV *on an $n \times n$ matrix with the choice $\beta = \sqrt{n}$. Then, the execution requires $O(P\sqrt{n}\lg n)$ space.*

Proof. Combine Lemma 9 and Theorem 1 from [35]. □

The work overhead of our algorithm may be reduced by increasing the constants in the $\Theta(\beta)$ threshold in line 7. Specifically, increasing this threshold by a constant factor reduces the number of reads and writes to temporaries by the same constant factor. As these temporaries constitute the majority of the work overhead of the algorithm, doubling the threshold nearly halves the overhead. In-

creasing the threshold, however, also increases the span by a constant factor, and so there is a trade-off.

6.6 Experimental design

This section describes our implementation of the CSB_SpMV and CSB_SpMV_T algorithms, the benchmark matrices we used to test the algorithms, the machines on which we ran our tests, and the other codes with which we compared our algorithms.

Implementation

We parallelized our code using Cilk++ [62], which is a faithful extension of C++ for multicore and shared-memory parallel programming. Cilk++ is based on the earlier MIT Cilk system [98], and it employs dynamic load balancing and provably optimal task scheduling. The CSB code used for the experiments is freely available for academic use at `http://gauss.cs.ucsb.edu/~aydin/software.html`.

The *row_ind* and *col_ind* arrays of CSB, which store the row and column indices of each nonzero within a block (i.e., the lower-order bits of the row and column indices within the matrix A), are implemented as a single ***index array***

by concatenating the two values together. The higher-order bits of *row_ind* and *col_ind* are stored only implicitly, and are retrieved by referencing the *blk_ptr* array.

The CSB blocks themselves are stored in row-major order, while the nonzeros within blocks are in Z-Morton order. The row-major ordering among blocks may seem to break the overall symmetry of CSB, but in practice it yields efficient handling of block indices for look-up in $A.\,blk\_ptr$ by permitting an easily computed look-up function $f(i, j)$. The row-major ordering also allowed us to count the nonzeros in a subblockrow more easily when computing $y \leftarrow Ax$. This optimization is not symmetric, but interestingly, we achieved similar performance when computing $y \leftarrow A^{\mathsf{T}}x$, where we must still aggregate the nonzeros in each block. In fact, in almost half the cases, computing $A^{\mathsf{T}}x$ was faster than Ax, depending on the matrix structure.

The Z-Morton ordering on nonzeros in each block is equivalent to first interleaving the bits of *row_ind* and *col_ind*, and then sorting the nonzeros using these bit-interleaved values as the keys. Thus, it is tempting to store the index array in a bit-interleaved fashion, thereby simplifying the binary searches in lines 7–9. Converting to and from bit-interleaved integers, however, is expensive with current hardware support,[6] which would be necessary for the serial base case in lines

[6]Recent research [166] addresses these conversions.

2–5. Instead, the kth element of the index array is the concatenation of $row\_ind[k]$ and $col\_ind[k]$, as indicated earlier. This design choice of storing concatenated, instead of bit-interleaved, indices requires either some care when performing the binary search (as presented in Figure 6.6) or implicitly converting from the concatenated to interleaved format when making a binary-search comparison. Our preliminary implementation does the latter, using a C++ function object for comparisons [188]. In practice, the overhead of performing these conversions is small, since the number of binary-search steps is small.

Performing the actual address calculation and determining the pointers to x and y vectors are done by masking and bit-shifting. The bitmasks are determined dynamically by the CSB constructor depending on the input matrix and the data type used for storing matrix indices. Our library allows any data type to be used for matrix indices and handles any type of matrix dynamically. For the results presented in Section 6.7, nonzero values are represented as double-precision floating-point numbers, and indices are represented as 32-bit unsigned integers. Finally, as our library aims to be general instead of matrix specific, we did not employ speculative low-level optimizations such as software prefetching, pipelining, or matrix-specific optimizations such as index and/or value compression [131, 207], but we believe that CSB and our algorithms should not adversely affect incorporation of these approaches.

Choosing the block size β

We investigated different strategies to choose the block size that achieves the best performance. For the types of loads we ran, we found that a block size slightly larger than $\sqrt{n}$ delivers reasonable performance. Figure 6.7 shows the effect of different blocksizes on the performance of the $y \leftarrow Ax$ operation with the representative matrix Kkt_power. The closest exact power of 2 to $\sqrt{n}$ is 1024, which turns out to be slightly suboptimal. In our experiments, the overall best performance was achieved when β satisfies the equation $\lceil \lg \sqrt{n} \rceil \leq \lg \beta \leq 3 + \lceil \lg \sqrt{n} \rceil$.

Merely setting β to a hard-coded value, however, is not robust for various reasons. First, the elements stored in the index array should use the same data type as that used for matrix indices. Specifically, the integer $\beta - 1$ should fit in 2 bytes so that a concatenated *row_ind* and *col_ind* fit into 4 bytes. Second, the length-β regions of the input vector x and output vector y (which are accessed when multiplying a single block) should comfortably fit into L2 cache. Finally, to ensure speedup on matrices with evenly distributed nonzeros, there should be enough parallel slackness for the parallelization across blockrows (i.e., the highest level parallelism). Specifically, when β grows large, the parallelism is roughly bounded by $O(nnz\,/(\beta \lg(n/\beta)))$ (by dividing the work and span from Theorem 6).

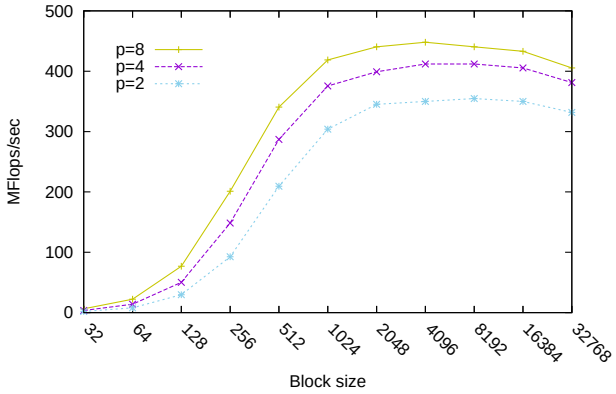


Figure 6.7: The effect of block size parameter β on SpMV performance using the Kkt_power matrix. For values $\beta > 32768$ and $\beta < 32$, the experiment failed to finish due to memory limitations. The experiment was conducted on the AMD Opteron.

Thus, we want $nnz\,/(\beta \lg(n/\beta))$ to be "large enough," which means limiting the maximum magnitude of β.

We adjusted our CSB constructor, therefore, to automatically select a reasonable block-size parameter β. It starts with $\beta = 3 + \lceil \lg \sqrt{n} \rceil$ and keeps decreasing it until the aforementioned constraints are satisfied. Although a research opportunity may exist to autotune the optimal block size with respect to a specific matrix and architecture, in most test matrices, choosing $\beta = \sqrt{n}$ degraded performance by at most 10%–15%. The optimal β value barely shifts along the x-axis when running on different numbers of processors and is quite stable overall.

An optimization heuristic for structured matrices

Even though CSB_SpMV and CSB_SpMV_T are robust and exhibit plenty of parallelism on most matrices, their practical performance can be improved on some sparse matrices having regular structure. In particular, a block diagonal matrix with equally sized blocks has nonzeros that are evenly distributed across blockrows. In this case, a simple algorithm based on blockrow parallelism would suffice in place of the more complicated recursive method from CSB_BlockV. This divide-and-conquer within blockrows incurs overhead that might unnecessarily degrade performance. Thus, when the nonzeros are evenly distributed across the blockrows, our implementation of the top-level algorithm (given in Figure 6.4) calls the serial multiplication in line 12 instead of the CSB_BlockrowV procedure.

To see whether a given matrix is amenable to the optimization, we apply the following "balance" heuristic. We calculate the imbalance among blockrows (or blockcolumns in the case of $y \leftarrow A^{\mathsf{T}}x$) and apply the optimization only when no blocks have more than twice the average number of nonzeros per blockrow. In other words, if $\max(nnz(A_i)) < 2 \cdot \text{mean}(nnz(A_i))$, then the matrix is considered to have balanced blockrows and the optimization is applied. Of course, this optimization is not the only way to achieve a performance boost on structured matrices.

Optimization of temporary vectors

One of the most significant overheads of our algorithm is the use of temporary vectors to store intermediate results when parallelizing a blockrow multiplication in CSB_BlockRowV. The "balance" heuristic above is one way of reducing this overhead when the nonzeros in the matrix are evenly distributed. For arbitrary matrices, however, we can still reduce the overhead in practice. In particular, we only need to allocate the temporary vector z (in line 14) if both of the subsequent multiplications (lines 15–17) are scheduled in parallel. If the first recursive call completes before the second recursive call begins, then we instead write directly into the output vector for both recursive calls. In other words, when a blockrow multiplication is scheduled serially, the multiplication procedure detects this fact and mimics a normal serial execution, without the use of temporary vectors. Our implementation exploits an undocumented feature of Cilk++ to test whether the first call has completed before making the second recursive call, and we allocate the temporary as appropriate. This test may also be implemented using Cilk++ reducers [97].

Sparse-matrix test suite

We conducted experiments on a diverse set of sparse matrices from real applications including circuit simulation, finite-element computations, linear program-

Name Description	Spy Plot	Dimensions Nonzeros	CSC (mean/max) CSB (mean/max)
Asic_320k circuit simulation		321K × 321K 1,931K	6.0 / 157K 4.9 / 2.3K
Sme3Dc 3D structural mechanics		42K × 42K 3,148K	73.3 / 405 111.6 / 1368
Parabolic_fem diff-convection reaction		525K × 525K 3,674K	7.0 / 7 3.5 / 1,534
Mittelmann LP problem		1,468K × 1,961K 5,382K	2.7 / 7 2.0 / 3,713
Rucci Ill-conditioned least-squares		1,977K × 109K 7,791K	70.9 / 108 9.4 / 36
Torso Finite diff, 2D model of torso		116K × 116K 8,516K	73.3 / 1.2K 41.3 / 36.6K
Kkt_power optimal power flow, nonlinear opt.		2.06M × 2.06M 12.77M	6.2 / 90 3.1 / 1,840
Rajat31 circuit simulation		4.69M × 4.69M 20.31M	4.3 / 1.2K 3.9 / 8.7K
Ldoor structural prob.		952K × 952K 42.49M	44.6 / 77 49.1 / 43,872
Bone010 3D trabecular bone		986K × 986K 47.85M	48.5 / 63 51.5 / 18,670
Grid3D200 3D 7-point finite-diff mesh		8M × 8M 55.7M	6.97 / 7 3.7 / 9,818
RMat23 Real-world graph model		8.4M × 8.4M 78.7M	9.4 / 70.3K 4.7 / 222.1K
Cage15 DNA electrophoresis		5.15M × 5.15M 99.2M	19.2 / 47 15.6 / 39,712
Webbase2001 Web connectivity		118M × 118M 1,019M	8.6 / 816K 4.9 / 2,375K

Figure 6.8: Structural information on the sparse matrices used in our experiments, ordered by increasing number of nonzeros.

ming, and web-connectivity analysis. These matrices not only cover a wide range of applications, but they also greatly vary in size, density, and structure. The test suite contains both rectangular and square matrices. Almost half of the square matrices are asymmetric. Figure 6.8 summarizes the 14 test matrices. The first ten matrices and Cage15 are from the University of Florida sparse matrix collection [73]. Grid3D200 is a 7-point finite difference mesh generated using the Matlab Mesh Partitioning and Graph Separator Toolbox [103]. The RMat23 matrix [140], which models scale-free graphs, is generated by using repeated Kronecker products [16]. We chose parameters $A = 0.7$, $B = C = D = 0.1$ for RMat23 in order to generate skewed matrices. Webbase2001 is a crawl of the World Wide Web from the year 2001 [59].

Included in Figure 6.8 is the load imbalance that is likely to occur for an SpMV algorithm parallelized with respect to columns (CSC) and blocks (CSB). In the last column, the average (mean) and the maximum number of nonzeros among columns (first line) and blocks (second line) are shown for each matrix. The sparsity of matrices can be quantified by the average number of nonzeros per column, which is equivalent to the mean of CSC. The sparsest matrix (Rajat31) has 4.3 nonzeros per column on the average while the densest matrix has about 73 nonzeros per column (Sme3Dc and Torso). For CSB, the reported mean/max

values are obtained by setting the block dimension β to be approximately $\sqrt{n}$, so that they are comparable with statistics from CSC.

Architectures and comparisons

We ran our experiments on three multicore superscalar architectures. Opteron is a ccNUMA architecture powered by AMD Opteron 8214 (Santa Rosa) processors clocked at 2.2 GHz. Each core of Opteron has a private 1 MB L2 cache, and each socket has its own integrated memory controller. Although it is an 8-socket dual-core system, we only experimented with up to 8 processors. Harpertown is a dual-socket quad-core system running two Intel Xeon X5460's, each clocked at 3.16 GHz. Each socket has 12 MB of L2 cache, shared among four cores, and a front-side bus (FSB) running at 1333 MHz. Nehalem is a single-socket quad-core Intel Core i7 920 processor. Like Opteron, Nehalem has an integrated memory controller. Each core is clocked at 2.66 GHz and has a private 256 KB L2 cache. The four cores share an 8 MB L3 cache.

While Opteron has 64 GB of RAM, Harpertown and Nehalem have only 8 GB and 6 GB, respectively, which forced us to exclude our biggest test matrix (Webbase2001) from our runs on Intel architectures. We compiled our code using `gcc 4.1` on Opteron and Harpertown and with `gcc 4.3` on Nehalem, all with optimization flags `-O2 -fno-rtti -fno-exceptions`.

To evaluate our code on a single core, we compared its performance with "pure" OSKI matrix-vector multiplication [205] running on one processor of Opteron. We did not enable OSKI's preprocessing step, which chooses blockings for cache and register usage that are tuned to a specific matrix. We conjecture that such matrix-specific tuning techniques can be combined advantageously with our CSB data structure and parallel algorithms.

To compare with a parallel code, we used the matrix-vector multiplication of Star-P [179] running on Opteron. Star-P is a distributed-memory code that uses CSR to represent sparse matrices and distributes matrices to processor memories by equal-sized blocks of rows.

6.7 Experimental results

Figures 6.9 and 6.10 show how CSB_SpMV and CSB_SpMV_T, respectively, scale for the seven smaller matrices on Opteron, and Figures 6.11 and 6.12 show similar results for the seven larger matrices. In most cases, the two codes show virtually identical performance, confirming that the CSB data structure and algorithms are equally suitable for both operations. In all the parallel scaling graphs, only the values $p = 1, 2, 4, 8$ are reported. They should be interpreted as perfor-

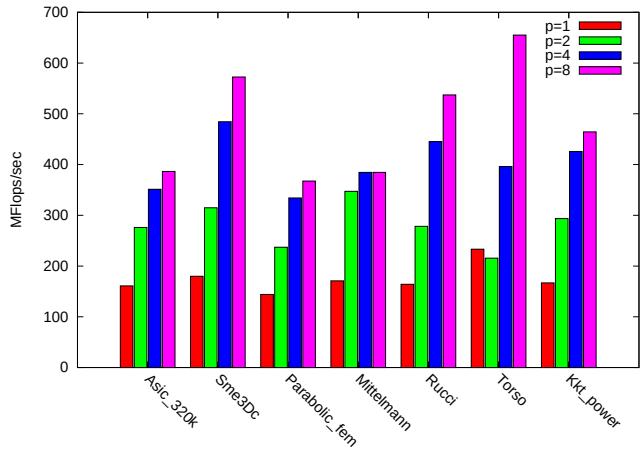


Figure 6.9: CSB_SpMV performance on Opteron (smaller matrices).

mance achievable by doubling the number of cores instead of as the exact performance on p threads (e.g. , $p = 8$ is the best performance achieved for $5 \leq p \leq 8$).

In general, we observed better speedups for larger problems. For example, the average speedup of CSB_SpMV for the first seven matrices was 2.75 on 8 processors, whereas it was 3.03 for the second set of seven matrices with more nonzeros. Figure 6.13 summarizes these results. The speedups are relative to the CSB code running on a single processor, which Figure 6.1 shows is competitive with serial CSR codes. In another study [208] on the same Opteron architecture, multicore-specific parallelization of the CSR code for 4 cores achieved comparable speedup to what we report here, albeit on a slightly different sparse-matrix test suite. That study does not consider the $y \leftarrow A^{\mathsf{T}}x$ operation, however, which

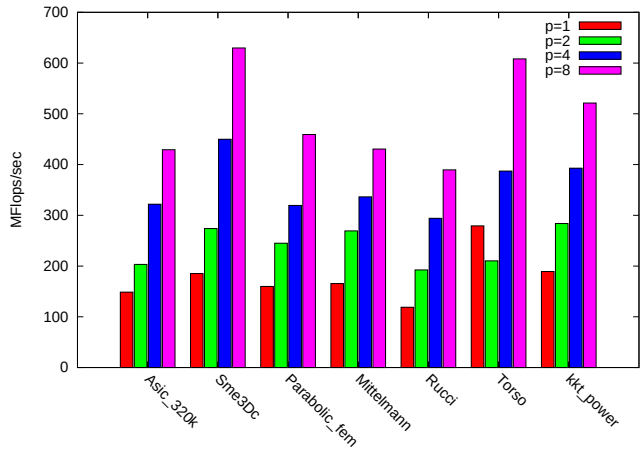


Figure 6.10: CSB_SpMV_T performance on Opteron (smaller matrices).

is difficult to parallelize with CSR but which achieves the same performance as $y \leftarrow Ax$ when using CSB.

For CSB_SpMV on 4 processors, CSB reached its highest speedup of 2.80 on the RMat23 matrix, showing that this algorithm is robust even on a matrix with highly irregular nonzero structure. On 8 processors, CSB_SpMV reached its maximum speedup of 3.93 on the Webbase2001 matrix, indicating the code's ability to handle very large matrices without sacrificing parallel scalability.

Sublinear speedup occurs only after the memory-system bandwidth becomes the bottleneck. This bottleneck occurs at different numbers of cores for different matrices. In most cases, we observed nearly linear speedup up to 4 cores. Although the speedup is sublinear beyond 4 cores, in every case (except CSB_SpMV on Mittelmann), we see some performance improvement going from 4 to 8 cores on

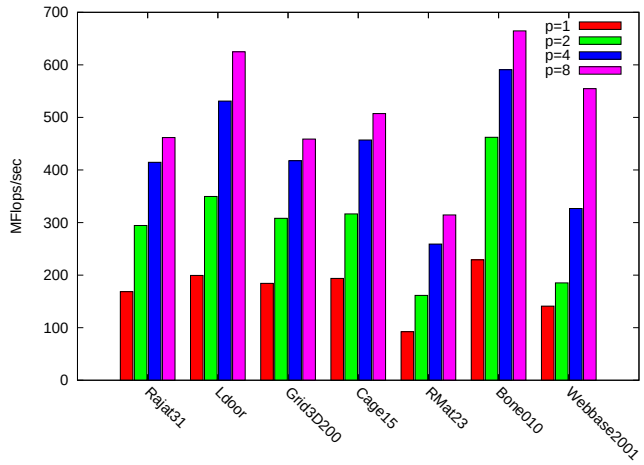


Figure 6.11: CSB_SpMV performance on Opteron (larger matrices).

Opteron. Sublinear speedup of SpMV on superscalar multicore architectures has been noted by others as well [208].

We conducted an additional experiment to verify that performance was limited by the memory-system bandwidth, not by lack of parallelism. We repeated each scalar multiply-add operation of the form $y_i \leftarrow y_i + A_{ij}x_j$ a fixed number t of times. Although the resulting code computes $y \leftarrow tAx$, we ensured that the compiler did not optimize away any multiply-add operations. Setting $t = 10$ did not affect the timings significantly—flops are indeed essentially free—but, for $t = 100$, we saw almost perfect linear speedup up to 16 cores, as shown in Figure 6.14. We performed this experiment with Asic_320k, the smallest matrix in the test suite, which should exhibit the least parallelism. Asic_320k is also irregular in structure,

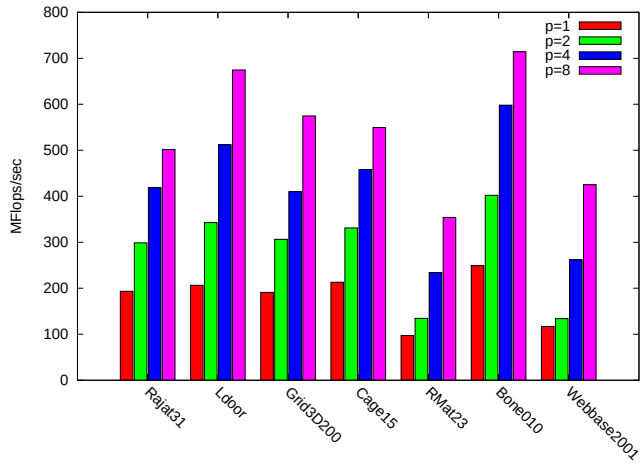


Figure 6.12: CSB_SpMV_T performance on Opteron (larger matrices).

Processors	CSB_SpMV		CSB_SpMV_T	
	1–7	8–14	1–7	8–14
$P = 2$	1.65	1.70	1.44	1.49
$P = 4$	2.34	2.49	2.07	2.30
$P = 8$	2.75	3.03	2.81	3.16

Figure 6.13: Average speedup results for relatively smaller (1–7) and larger (8–14) matrices. These experiments were conducted on Opteron.

which means that our balance heuristic does not apply. Nevertheless, CSB_SpMV scaled almost perfectly given enough flops per byte.

The parallel performance of CSB_SpMV and CSB_SpMV_T is generally not affected by highly uneven row and column nonzero counts. The highly skewed matrices RMat23 and Webbase2001 achieved speedups as good as for matrices with flat row and column counts. An unusual case is the Torso matrix, where both CSB_SpMV and CSB_SpMV_T were actually slower on 2 processors than

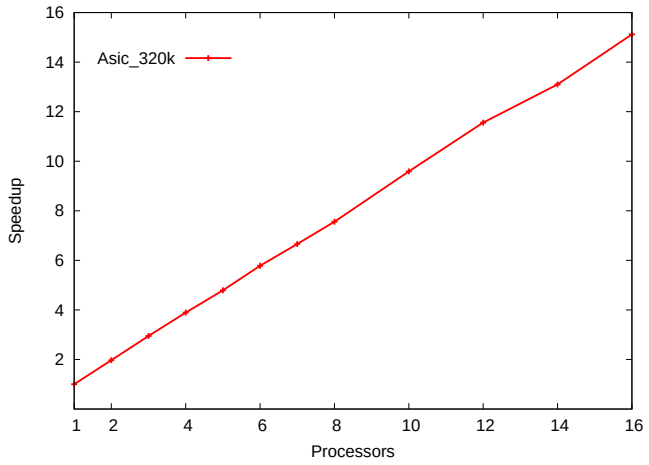


Figure 6.14: Parallelism test for CSB_SpMV on Asic_320k obtained by artificially increasing the flops per byte. The test shows that the algorithm exhibits substantial parallelism and scales almost perfectly given sufficient memory bandwidth.

serially. This slowdown does not, however, mark a plateau in performance, since Torso speeds up as we add more than 2 processors. We believe this behavior occurs because the overhead of intrablock parallelization is not amortized for 2 processors. Torso requires a large number of intrablock parallelization calls, because it is unusually irregular and dense.

Figure 6.15 shows the performance of CSB_SpMV on Harpertown for a subset of test matrices. We do not report performance for CSB_SpMV_T, as it was consistently close to that of CSB_SpMV. The performance on this platform levels off beyond 4 processors for most matrices. Indeed, the average Mflops/sec on 8

processors is only 3.5% higher than on 4 processors. We believe this plateau results from insufficient memory bandwidth. The continued speedup on Opteron is due to its higher ratio of memory bandwidth (bytes) to peak performance (flops) per second.

Figure 6.16 summarizes the performance results of CSB_SpMV for the same subset of test matrices on Nehalem. Despite having only 4 physical cores, for most matrices, Nehalem achieved scaling up to 8 threads thanks to hyperthreading. Running 8 threads was necessary to utilize the processor fully, because hyperthreading fills the pipeline more effectively. We observed that the improvement from oversubscribing is not monotonic, however, because running more threads reduces the effective cache size available to each thread. Nehalem's point-to-point interconnect is faster than Opteron's (a generation old Hypertransport 1.0), which explains its better speedup values when comparing the 4-core performance of both architectures. Its raw performance is also impressive, beating both Opteron and Harpertown by large margins.

To determine CSB's competitiveness with a conventional CSR code, we compared the performance of the CSB serial code with plain OSKI using no matrix-specific optimizations such as register or cache blocking. Figures 6.17 and 6.18 present the results of the comparison. As can be seen from the figures, CSB achieves similar serial performance to CSR.

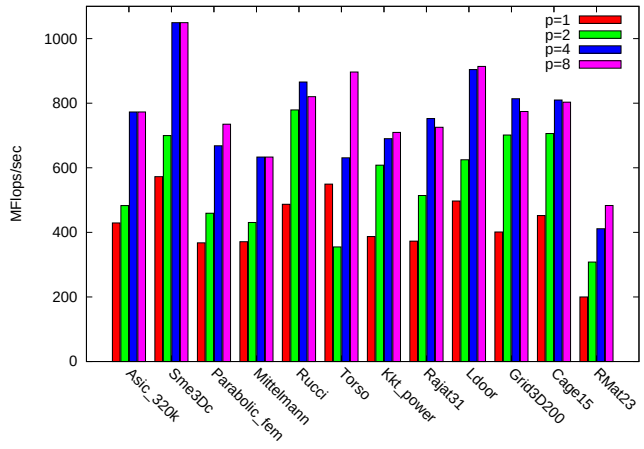


Figure 6.15: CSB_SpMV performance on Harpertown.

In general, CSR seems to perform best on ***banded matrices***, all of whose nonzeros are located near the main diagonal. (The maximum distance of any nonzero from the diagonal is called the matrix's ***bandwidth***, not to be confused with memory bandwidth.) If the matrix is banded, memory accesses to the input vector x tend to be regular and thus favorable to cacheline reuse and automatic prefetching. Strategies for reducing the bandwidth of a sparse matrix by permuting its rows and columns have been studied extensively (see [70, 197], for example). Many matrices, however, cannot be permuted to have low bandwidth. For matrices with scattered nonzeros, CSB outperforms CSR, because CSR incurs many cache misses when accessing the x vector. An example of this effect occurs for the RMat23 matrix, where the CSB implementation is almost twice as fast as CSR.

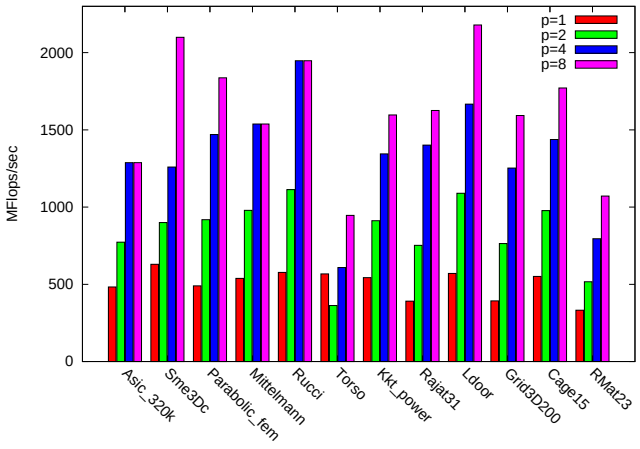


Figure 6.16: CSB_SpMV performance on Nehalem.

Figure 6.19 compares the parallel performance of the CSB algorithms with Star-P. Star-P's blockrow data distribution does not afford any flexibility for load-balancing across processors. Load balance is not an issue for matrices with nearly flat row counts, including finite-element and finite-difference matrices, such as Grid3D200. Load balance does become an issue for skewed matrices such as RMat23, however. Our performance results confirm this effect. CSB_SpMV is about 500% faster than Star-P's SpMV routine for RMat23 on 8 cores. Moreover, for any number of processors, CSB_SpMV runs faster for all the matrices we tested, including the structured ones.

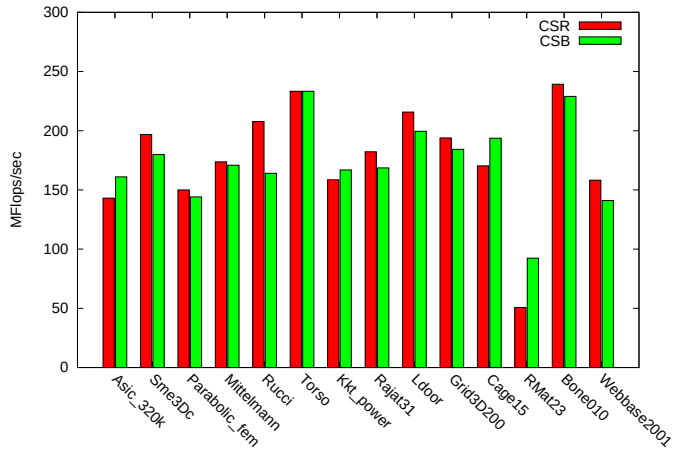


Figure 6.17: Serial performance comparison of SpMV for CSB and CSR.

6.8 Conclusion

Compressed sparse blocks allow parallel operations on sparse matrices to proceed either row-wise or column-wise with equal facility. We have demonstrated the efficacy of the CSB storage format for SpMV calculations on a sparse matrix or its transpose. It remains to be seen, however, whether the CSB format is limited to SpMV calculations or if it can also be effective in enabling parallel algorithms for multiplying two sparse matrices, performing LU-, LUP-, and related decompositions, linear programming, and a host of other problems for which serial sparse-matrix algorithms currently use the CSC and CSR storage formats.

The CSB format readily enables parallel SpMV calculations on a symmetric matrix where only half the matrix is stored, but we were unable to attain one

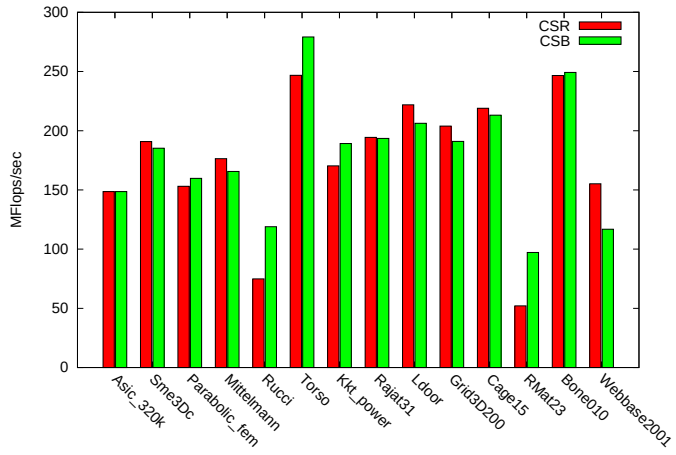


Figure 6.18: Serial performance comparison of SpMV_T for CSB and CSR.

optimization that serial codes exploit in this situation. In a typical serial code that computes $y \leftarrow Ax$, where $A = (a_{ij})$ is a symmetric matrix, when a processor fetches $a_{ij} = a_{ji}$ out of memory to perform the update $y_i \leftarrow y_i + a_{ij}x_j$, it can also perform the update $y_j \leftarrow y_j + a_{ij}x_i$ at the same time. This strategy halves the memory bandwidth compared to executing CSB_SpMV on the matrix, where $a_{ij} = a_{ji}$ is fetched twice. It remains an open problem whether the 50% savings in storage for sparse matrices can be coupled with a 50% savings in memory bandwidth, which is an important factor of 2, since it appears that the bandwidth between multicore chips and DRAM will scale more slowly than core count.

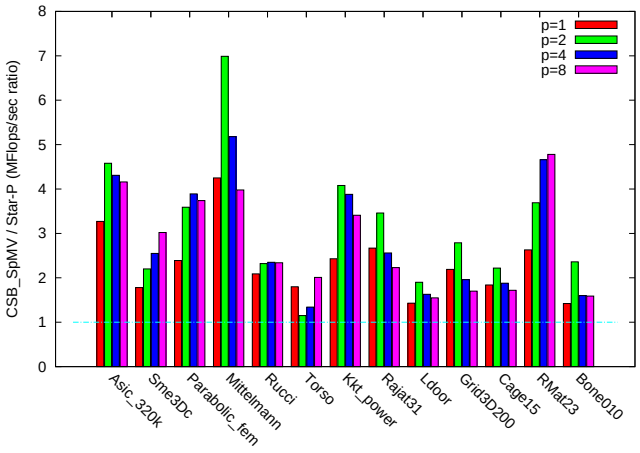


Figure 6.19: Performance comparison of parallel CSB_SpMV with Star-P, which is a parallel-dialect of Matlab. The vertical axis shows the performance ratio of CSB_SpMV to Star-P. A direct comparison of CSB_SpMV_T with Star-P was not possible, because Star-P does not natively support multiplying the transpose of a sparse matrix by a vector.

Chapter 7

Solving Path Problems on the GPU

Abstract

We consider the computation of shortest paths on Graphic Processing Units (GPUs). The blocked recursive elimination strategy we use is applicable to a class of algorithms (such as all-pairs shortest-paths, transitive closure, and LU decomposition without pivoting) having similar data access patterns. Using the all-pairs shortest-paths problem as an example, we uncover potential gains over this class of algorithms. The impressive computational power and memory bandwidth of the GPU make it an attractive platform to run such computationally intensive algorithms. Although improvements over CPU implementations have previously been achieved for those algorithms in terms of raw speed, the utilization of the underlying computational resources was quite low. We implemented a recursively partitioned all-pairs shortest-paths algorithm that harnesses the power of GPUs better than existing implementations. The alternate schedule of path computations allowed us to cast almost all operations into matrix-matrix multiplications on a semiring. Since matrix-matrix multiplication is highly optimized and has a high ratio of computation to communication, our implementation does not suffer from the premature saturation of bandwidth resources as iterative algorithms do. By increasing temporal locality, our implementation runs more than two orders of magnitude faster on an NVIDIA 8800 GPU than on an Opteron. Our work provides evidence that programmers should rethink algorithms instead of directly porting them to GPU.

This chapter is based on a paper [50] by Buluç et al. from Parallel Computing.

7.1 Introduction

The massively parallel nature of GPUs makes them capable of yielding theoretically much higher GFlops rates than current state-of-the-art CPUs. GPU performance also grows much faster than CPU performance due to specialized explicit parallelism. The amount of computational power to be harvested has also attracted the high-performance computing (HPC) community, and we have seen many scientific applications successfully implemented with significant performance gains on the GPU [37, 178].

Implementing HPC applications to run on a GPU requires significant expertise, even with the recently introduced C-like APIs such as Nvidia's Cuda platform [141]. The key to performance is to hide the data access latency by having many threads on the fly. The performance is usually fragile and requires careful craftmanship from the programmer's side. It is up to the programmer to make sure that the registers and other levels of cache are neither underutilized nor overpressured. Several papers are devoted to the issue of achieving the right balance to get optimal performance on GPUs [170, 204], relying on novel programming techniques that are not necessarily intuitive to the existing HPC programmer.

An important class of algorithms with triple nested loops, which will be subsequently mentioned as Gaussian Elimination (GE) based algorithms, have very similar data access patterns. Examples include LU decomposition without pivoting, Cholesky factorization, all-pairs shortest paths (APSP), and transitive closure. The similarity among those problems has led researchers to approach them in a unified manner. For example, the Gaussian Elimination Paradigm of Chowdhury and Ramachandran provides a cache-oblivious framework for these problems [60]. In this paper, we specifically focus on the APSP problem because it usually operates on single precision floating point data, making it suitable to current generation GPUs. On the contrary, factorizations such as LU and Cholesky require double precision arithmetic that was not available on the GPUs until very recently (with AMD FireStream 9170 and Nvidia GeForce GTX 280). Even now, the double precision performance is 4-8 times slower than single precision, and the limited global memory of current generation GPUs discourage the use of double precision floating point numbers. Furthermore, numerical LU decomposition without pivoting is unstable [106] at best (it may not even exist), and pivoting strategies on the GPU are beyond the scope of this paper. Volkov and Demmel did an excellent job of implementing LU, QR, and Cholesky factorizations on the GPU, albeit in single precision [204]. It is worth noting that even though our implementation computes only the distance version of the APSP problem, it is

possible to obtain the actual minimal paths, at the cost of doubling the memory requirements, by keeping a predecessor matrix.

Our two main contributions in this paper are:

1. Recursive partitioning is used as a tool to express a different schedule of path computations that allows extensive use of highly optimized matrix-matrix operations. Specifically, we use matrix multiplication on semirings as a building block for GE based algorithms. By doing so, we increase data locality, which is even more important for high performance computing on the GPU than on the CPU

2. As a proof of concept, we provide an efficient implementation of the APSP algorithm on the GPU that is up to 480x faster than our reference CPU implementation, and up to 75x faster than an existing GPU implementation on a similar architecture.

Locality of reference has always been an issue in algorithm design, and it will be even more important with GPUs. This is because stream processors, such as GPUs, achieve efficiency through locality [72]. Our work highlights the importance of recursion as a technique for automatically creating locality of reference.

As minor contributions, we give an alternate (arguably simpler) proof of correctness based on path expressions for the recursively partitioned APSP algorithm.

On the GPU, we compare iterative, and recursive versions of the same algorithm and provide insights into their performance difference through micro benchmarks. Therefore, we provide evidence that Level 3 BLAS [80] routines on semirings can be used to speed up certain graph algorithms. Finally, we compare different CPUs and GPUs on their power efficiency in solving this problem.

The remainder of this paper is organized as follows. Section 7.2 describes the algorithms based on block-recursive elimination, starting from the well-known Gaussian Elimination procedure and using it as an analogy to explain block-recursive elimination on other algebraic structures. Most specifically, it shows how block-recursive elimination can be used to solve the all-pairs shortest-paths problem. Section 7.3 is devoted to GPU programming on the Cuda platform, showing difficulties and important points to achieve high performance on GPUs. Section 7.4 describes our implementation and evaluation strategies, and reports on the results of our experiments. Section 7.5 offers some concluding remarks.

7.2 Algorithms Based on Block-Recursive Elimination

Gaussian elimination is used to solve a system of linear equations $Ax = b$, where A is an $n \times n$ matrix of coefficients, x is a vector of unknowns, and b

is a vector of constants. Recursive blocked LU factorization is an efficient way of performing Gaussian elimination on architectures with deep memory hierarchies [87, 198]. This is mostly due to its extensive use of matrix-matrix operations (Level 3 BLAS [80]) that are optimized for the underlying architecture. Let A and its factors L and U be partitioned as

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & L_{22} \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} \\ & U_{22} \end{bmatrix} \tag{7.1}$$

Then, the block-recursive LU decomposition without pivoting can be written as

$$\begin{aligned} L_{11}, U_{11} &\leftarrow \texttt{LU}(A_{11}) \\ U_{12} &\leftarrow L_{11} \backslash A_{12} \\ L_{21} &\leftarrow A_{21} / U_{11} \\ L_{22}, U_{22} &\leftarrow \texttt{LU}(A_{22} - L_{21} U_{12}). \end{aligned} \tag{7.2}$$

In this pseudocode, `LU` is the recursive call to the function itself, \ and / denote triangular solve operations with multiple right hand sides (matrix division on the left and on the right, in MATLAB notation).

LU factorization operates on the field of real numbers, but the same algorithm can be used to solve a number of graph problems, albeit using a different

algebra. Specifically, closed semirings provide a general algebraic structure that can be used to solve a number of path problems on graphs [7, 192]. A semiring has all the properties of a ring, except that there might be elements without an additive inverse. One practical implication is that fast matrix multiplication algorithms that use additive inverses, such as the Strassen algorithm [187] and the Coppersmith-Winograd algorithm [67], do not apply to matrices over semirings.

A closed semiring is formally denoted by $(\mathbb{S}, \oplus, \otimes, 0, 1)$, where $\oplus$ and $\otimes$ are binary operations defined on the set $\mathbb{S}$ with identity elements 0 and 1 respectively. Fletcher [93] gives a complete definition of a closed semiring. Two important semirings used in this work are the *Boolean semiring*, formally defined as $(\{0, 1\}, \vee, \wedge, 0, 1)$ and the *tropical semiring*, formally defined as $(\mathbb{R}^+, \min, +, \infty, 0)$. A closed semiring is said to be *idempotent* if $a \oplus a = a$ for all $a \in \mathbb{S}$. Although idempotence of the semiring is not a requirement for the solution of path problems on graphs [93], the correctness of our in-place algorithms relies on idempotence. Both the Boolean semiring and the tropical semiring are idempotent, as $\min(a, a) = a$ for all $a \in \mathbb{R}^+$, and $0 \vee 0 = 0$, $1 \vee 1 = 1$.

7.2.1 The All-Pairs Shortest-Paths Problem

The all-pairs shortest-paths (APSP) is a fundamental graph problem. Given a directed graph $G = (V, E)$ with vertices $V = \{v_1, v_2, ..., v_n\}$ and edges $E =$

$\{e_1, e_2, ..., e_m\}$, the problem is to compute the length of the shortest path from v_i to v_j for all (v_i, v_j) pairs. APSP corresponds to finding the matrix closure $A^* = \sum_{i=0}^{\infty} A^i = \sum_{i=0}^{n} A^i = I \oplus A \oplus ... \oplus A^n$ on the tropical semiring. Note that we were able to avoid the problems with the infinite sum by converting it to a finite sum, because $A^{n+i} = A^n$ for $i > 0$ in any idempotent semiring.

APSP is the focus of this paper among the set of GE based algorithms due to its practical importance and the lack of fast implementations on the GPU. All the algorithms discussed in this paper take the adjacency matrix A of the graph, where $A(i, j)$ represents the length of the edge $v_i \rightarrow v_j$, as the input. They output A^*, where $A^*(i, j)$ represents the length of the shortest path from v_i to v_j. Edge weights can be arbitrary (positive, negative, or zero), but we assume that there are no negative cycles in the graph. Also, the cost of staying at the same vertex is zero, i.e., $A(i, i) = 0$. If not, we can delete any edge of the form $A(i, i) \neq 0$ as it will certainly not contribute to any shortest path. This is because shortest paths are simple when there are no negative cycles.

The standard algorithm for solving the APSP problem is the Floyd-Warshall (FW) algorithm. The pseudocode for the FW algorithm, in standard notation and in linear algebra notation, are given in Figures 7.1 and 7.2. It is especially well-suited for dense graphs due to its $O(n^3)$ complexity. It is a dynamic programming algorithm that consists of a triply nested loop similar to matrix multiplication. In

$A^* : \mathbb{R}^{\mathrm{N}\times\mathrm{N}} = \mathrm{FW}(A : \mathbb{R}^{\mathrm{N}\times\mathrm{N}})$
1 **for** $k \leftarrow 0$ **to** $N-1$
2 **do for** $i \leftarrow 0$ **to** $N-1$
3 **do for** $j \leftarrow 0$ **to** $N-1$
4 **do** $A(i,j) \leftarrow \min(A(i,j), A(i,k) + A(k,j))$
5 $A^* \leftarrow A$

Figure 7.1: FW algorithm in the standard notation

fact, computing the APSP problem is computationally equivalent to computing the product of two matrices on a semiring [7]. However, the order of the loops cannot be changed arbitrarily as in the case of matrix multiplication. In the linear algebra sense, the algorithm computes the outer product of the kth row and the kth column, and does rank-1 updates on the whole matrix, for $k = 1, 2, ..., n$. The order of the outer product updates cannot be changed, but one is free to compute the outer product in any order. This means that the k-loop should be the outermost loop, and the other loops can be freely interchanged. Although the added constraint on the order of loops hinders some of the loop-interchange optimizations that are applied to matrix multiplication, automatic program generators for the FW algorithm have been shown to provide formidable speedups [115].

For sparse graphs, Johnson's algorithm [126], which runs Dijkstra's single-source shortest paths algorithm from every vertex (after some preprocessing that

$A^* : \mathbb{R}^{N \times N} = \mathrm{FW}(A : \mathbb{R}^{N \times N})$
1 **for** $k \leftarrow 0$ **to** $N - 1$
2 **do** $A \leftarrow A \oplus A(:, k) \otimes A(k, :)$ ▷ Algebra on the (min,+) semiring
3 $A^* \leftarrow A$

Figure 7.2: FW algorithm in linear algebra notation

lets the algorithm run on graphs having edges with negative weights), is probably the algorithm of choice for an implementation on the CPU. However, as we demonstrate in Section 7.4, the GE based algorithm clearly outperforms both the FW algorithm and Johnson's algorithm when implemented on the GPU.

For unweighted graphs, it is possible to embed the semiring into the ring of integers and use a fast, sub-cubic matrix multiplication algorithm such as Strassen's [187]. For an undirected and unweighted graph, Seidel [177] gives a $O(M(n) \lg n)$ algorithm, where $M(n)$ is the time to multiply two $n \times n$ matrices on the ring of integers. This elegant algorithm repeatedly squares the adjacency matrix of the graph. However, it is not currently known how to generalize Seidel's algorithm to weighted or directed graphs [213].

7.2.2 Recursive In-Place APSP Algorithm

The closure of a matrix can be computed using an algorithm similar to recursive Gaussian elimination without pivoting. It is guaranteed to terminate on

$A^* : \mathbb{R}^{\mathrm{N}\times\mathrm{N}} = \mathrm{APSP}(A : \mathbb{R}^{\mathrm{N}\times\mathrm{N}})$

1 **if** $\mathrm{N} < \beta$
2 **then** $A \leftarrow \mathrm{FW}(A)$ ▷ Base case: perform iterative FW serially
3 **else**
4 $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$
5 $A_{11} \leftarrow \mathrm{APSP}(A_{11})$
6 $A_{12} \leftarrow A_{11}A_{12}$
7 $A_{21} \leftarrow A_{21}A_{11}$
8 $A_{22} \leftarrow A_{22} \oplus A_{21}A_{12}$
9 $A_{22} \leftarrow \mathrm{APSP}(A_{22})$
10 $A_{21} \leftarrow A_{22}A_{21}$
11 $A_{12} \leftarrow A_{12}A_{22}$
12 $A_{11} \leftarrow A_{11} \oplus A_{12}A_{21}$

Figure 7.3: Pseudocode for recursive in-place APSP

a closed semiring like the tropical semiring. The only subroutine of this algorithm is matrix multiplication on a semiring. The n-by-n adjacency matrix is recursively partitioned into four equal-sized $n/2$-by-$n/2$ submatrices as before; the pseudocode for the algorithm is shown in Figure 7.3. We use juxtaposition (AB) to denote the multiplication of A and B on the semiring. β is the threshold after which the algorithm performs iterative FW serially instead of recursing further. The algorithm does not require n to be even. If n is odd, the same decomposition in (7.1) works with $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$.

Both the original FW implementation given in Figures 7.1 and 7.2 as well as the recursive algorithm given in Figure 7.3 can be easily extended to obtain the

actual minimal paths. In this case, an additional integer matrix Π of predecessor vertices is maintained. Initially, $\Pi(i,j) \leftarrow i$ for all i. It is updated whenever a previously unknown path with shorter length is discovered, i.e., $\Pi(i,j) \leftarrow k$ whenever $A(i,k) + A(k,j) < A(i,j)$ during the computation. As FW and APSP are essentially performing the same computation, the discovered shortest path is guaranteed to be a path with minimal length. However, they may find different, yet equal in length, paths in the presence of multiple shortest paths for a source-destination pair. This is due to possibly different schedule of path computation.

Recursive formulations of APSP have been presented by many researchers over the years [71, 158, 196]. The connection to semiring matrix multiplication was shown by Aho et al. [7], but they did not present a complete algorithm. Ours is a modified version of the algorithm of Tiskin [196] and R-Kleene algorithm [71]. Especially, the in-place nature of the R-Kleene algorithm helped us avoid expensive global memory to global memory data copying. As the algorithm makes use of matrix multiplication as a subroutine, it has a much higher data reuse ratio while having asymptotically the same operation count.

The correctness of the recursive algorithm has been formally proven in various ways before [71, 158]. Here we present a simpler proof based on algebraic paths. As in Aho et al. [7], we partition the set of vertices into $V_1 = \{v_1, ..., v_{n/2}\}$ and $V_2 = \{v_{n/2+1}, ..., v_n\}$. Submatrix A_{11} represents the edges within V_1, submatrix

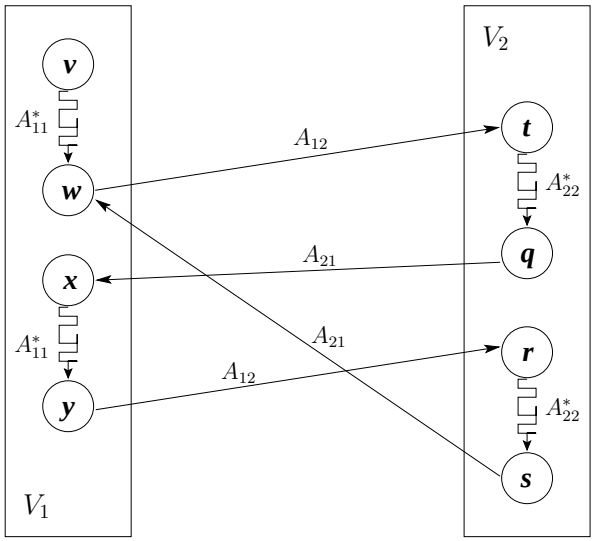


Figure 7.4: An example path in A_{11}^*

A_{12} the edges from V_1 to V_2, submatrix A_{21} the edges from V_2 to V_1, and submatrix A_{22} the edges within V_2.

Now, consider the paths in A_{11}^*. They can either travel within V_1 only or move from V_1 to V_2 following an edge in A_{12}, and then come back to V_2 through an edge in A_{21}, possibly after traveling within V_2 for a while by following edges in A_{22}. The regular expression for the latter path is $A_{12}A_{22}^*A_{21}$. This partial path can be repeated a number of times, possibly going through different vertices each time. An example path from v to w is shown in Figure 7.4. The complete regular expression becomes

$$A_{11}^* = (A_{11} \mid A_{12}A_{22}^*A_{21})^*. \tag{7.3}$$

On the other hand, the regular expression we get after the recursive algorithm terminates is

$$A_{11}^* = A_{11}^* \mid (A_{11}^* A_{12} (A_{22} \mid A_{21} A_{11}^* A_{12})^* A_{21} A_{11}^*). \tag{7.4}$$

These two regular expressions define the same language, hence represent the same set of paths [192]. By converting these regular expressions into deterministic finite automata (DFA), and minimizing them [121], we see that both have the same minimum-state DFA shown in Figure 7.5. Since the minimum-state DFA is unique for a language, this proves that the algorithm computes the correct set of paths.

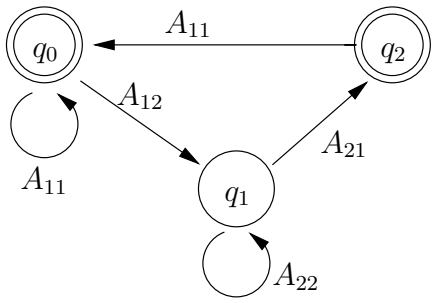


Figure 7.5: Minimum-state DFA for the path expressions in A_{11}^*, starting state is q_0

It is also possible to implement this algorithm in a blocked iterative way as previously done for transitive closure [200]. The percentage of work done iteratively (without using matrix multiplication) is the same, and corresponds to the block diagonal part of the matrix. However, the multiplications in the blocked

algorithm are always between matrices of size $B \times B$, where B is the blocking factor. This is potentially a limiting factor on GPUs because multiplication tends to get drastically faster as matrices get bigger (less than 20 GFlops/s when N=64 versus 200 GFlops/s when N=1024) [204]. With the recursive formulation, on the other hand, more work can be done during multiplication of large matrices.

Furthermore, the recursive algorithm does fewer kernel launches than the block iterative one. The block iterative algorithm launches $O((N/B)^3)$ kernels for matrix multiplications and $O(N/B)$ kernels for computing closures of $B \times B$ blocks on the diagonal. On the other hand, at each level of the recursion tree, the recursive algorithm launches 6 kernels for matrix multiplications, and does 2 recursive calls. This makes a total of only $O(N/B)$ kernel launches because the height of the recursion tree is $\lg(N/B)$, and the number of kernel launches doubles at each level ($\{6, 12, 24, ..., 6(N/B)\}$). The $O((N/B)^2)$ factor of improvement can be quite significant, as kernel launches incur significant overhead in CUDA.

One important feature of our implementation is that it is performed in place, overwriting the input with the output without constraining the order of loops in the matrix multiplication. For the matrix multiply-add operations $A_{22} \leftarrow A_{22} \oplus A_{21}A_{12}$ and $A_{11} \leftarrow A_{11} \oplus A_{12}A_{21}$, there are no issues of correctness. However, for other multiplications of the form $B \leftarrow BA$ or $B \leftarrow AB$, the order of evaluation (whether it is an ijk loop or an kji loop) matters on a general semiring.

This is because updating the output automatically updates the input, and the algorithm will now use a different input for the rest of the computation. As proved by D'Alberto and Nicolau [71], this is not a problem as long as the semiring is idempotent and A is a closure. The intuition is that if the algorithm prematurely overwrites its input, this just makes the algorithm find shortest paths quicker. In other words, it speeds up the information dissemination, but the correctness is preserved thanks to idempotence.

Note that four of the six multiplications at any level of the recursion tree are of the form $B \leftarrow BA$ or $B \leftarrow AB$. In other words, they perform multiply instead of multiply-add operations. Using $B \leftarrow B + BA$ or $B \leftarrow B + AB$ would be equally correct, but unnecessary. Remember that the cost of staying in a vertex is zero, i.e., $A(i, i) = 0$. Consider $B \leftarrow AB$: If B contains a path $v_i \Rightarrow v_j$ before the operation, AB generates a cost-equivalent path $v_i \Rightarrow v_i \Rightarrow v_j$ and safely overwrites B.

7.3 GPU Computing Model with CUDA

More and more applications that traditionally run on the CPU are now being reimplemented to run on the GPU, a technique called general-purpose computing on graphics processing units (GPGPU). Both Nvidia and AMD offer programming

interfaces for making GPGPU accessible to programmers who are not experts in computer graphics [1, 2]. Nvidia's Compute Unified Device Architecture (Cuda) offers a higher level C-like API, whereas AMD's Close-to-Metal (CTM) allows the programmers to access lower levels of hardware. As opposed to CTM, the Cuda platform is unified in the sense that it has no architectural division for vertex and pixel processing.

7.3.1 GPU Programming

The new generation of GPUs are basically multithreaded stream processors. They offer tremendous amounts of bandwidth and single-precision floating point arithmetic computation rates. In stream processing, a single data parallel function (kernel) is executed on a stream of data, and that is exactly how the Cuda programming model works. A Cuda program is composed of two parts: A host (CPU) code that makes kernel calls, and a device (GPU) code that actually implements the kernel. The host code is conceptually a serial C program, but the device code should be massively parallel in order to harness the power of the GPU.

The fundamental building block of Nvidia 8 and 9 series is the streaming multiprocessors (SMs), sometimes called the GPU chips. Each SM consists of 8 streaming processors (cores), but only one instruction fetch/decode unit. This implies that all 8 cores must simultaneously execute the same instruction. This

is why divergence in the device code should be avoided as much as possible. The memory hierarchy consists of multiple levels. Each SM has 8192 registers and 16KB on-chip shared memory, which is as fast as registers provided that bank conflicts are avoided. A high-latency (200-300 cycles) off-chip global memory provides the main storage for the application on the GPU. Part of the off-chip memory, called the local memory, is used for storing variables that are spilled from registers.

A kernel is executed by many threads on the GPU. These threads are organized as a grid of thread blocks, which are batches of threads that can cooperate/communicate through on-chip shared memory and synchronize their execution. Each thread block is executed by only one SM, but each SM can execute multiple thread blocks simultaneously.

The main scheduling unit in Cuda is a *warp*, a group of 32 threads from the same thread block. All threads in a warp execute the same instruction, and execution of an arithmetic instruction for the whole warp takes 4 clock cycles. The number of active warps in a block is an important factor in tolerating global memory access latency.

7.3.2 Experiences and Observations

Some limitations exist for the device code. For example, recursion and static variables are not allowed. These limitations do not apply to the host code, as it is just a regular C code running on the CPU. In fact, recursion in the host code is a powerful technique, since it naturally separates the recursion stack from the floating-point intensive part of the program. Although recursive divide-and-conquer algorithms are naturally cache efficient [114], they have traditionally not achieved their full performance due to the overheads associated with recursion. We do not have such a limitation with CUDA because the recursion stack, which is on the CPU, does not interfere with the kernel code on the GPU.

Code optimization on a GPU is a tedious job with many pitfalls. Performance on a GPU is often more fragile than performance on a CPU. It has been observed that small changes can cause huge effects on the performance [170]. For example, in the optimized GEMM routine of Volkov [204], each thread block is 16×4 and each thread uses 32 registers. This allows $8192/32 = 256$ threads and $256/64 = 4$ thread blocks can simultaneously be active on each SM. As there are two warps per thread block and it takes 4 cycles to execute an instruction for the whole warp, a latency of $8 \times 4 = 32$ cycles can be completely hidden. In the case that an extra variable is required, the compiler can either choose to spill it out to local memory and keep the register count intact, or increase the register usage per thread by

one. In the latter case, the number of active thread blocks decreases to 3. This introduces a 25% reduction in parallelism, but the former option may perform worse if the kernel has few instructions because access to a local variable will introduce one-time extra latency of 200-300 cycles. Whichever option is chosen, it is obvious that performance is fragile: by just adding one extra line, it is possible to drastically slow down the computation.

Another pitfall awaiting the programmer is bandwidth optimizations. In Cuda, peak bandwidth can only be achieved through memory coalescing, i.e., by making consecutively numbered threads access consecutive memory locations. One can heavily underutilize the GPU bandwidth by not paying attention to memory coalescing. However, the way memory coalescing works is quite counter-intuitive to a multicore programmer. Assume that one wants to scan a $16 \times N$ matrix stored in row-major order. On an SMP system with 16 cores, the most bandwidth-friendly way is to let each processor scan a different row of the matrix; in this case, each processor makes at most N/B cache misses, which is optimal. On an Nvidia GPU, on the other hand, this will create multiple memory accesses per warp since these threads do not access contiguous range of memory addresses. An example with $N = 8$ is shown in Figure 7.6. However, if the matrix were stored in column-major order, having each thread scan a different row would be optimal on an Nvidia GPU. This is because memory accesses at each step would

be coalesced into a single access by the NVCC compiler [155]. Consequently, the right programming practices for achieving high bandwidth are quite different for the GPU than for traditional parallel programming.

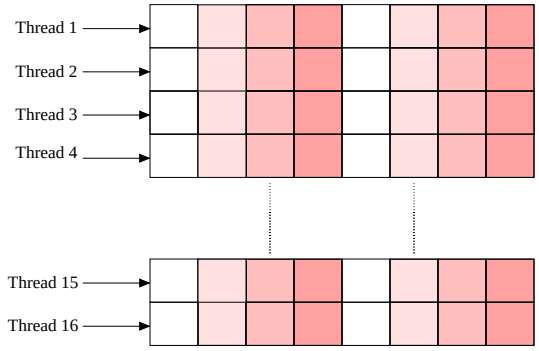


Figure 7.6: Stride-1 access per thread (row-major storage)

As a result, we advocate the use of optimized primitives as much as possible on the GPU. Harris et al. provide an excellent optimized scan primitive with Cuda and encourage its use as a building block for implementing parallel algorithms on Nvidia GPUs [117]. Here, we advocate the use of matrix-matrix multiplication as an important primitive, not only for solving systems of linear equations, but also for graph computations. In terms of performance, matrix multiplication has been claimed to be unsuitable to run on GPUs due to the lack of sufficient bandwidth [89]. The new generation GPUs, however, offer a tremendous bandwidth of more than 100 GB/s. Moreover, alternate implementations that are not band-

width bound achieved close to peak performance [204]. It would be wise to take advantage of such an efficient primitive whenever possible.

7.4 Implementation and Experimentation

7.4.1 Experimental Platforms

We ran our GPU code on an Nvidia GeForce 8800 Ultra with Cuda SDK 1.1 and GCC version 4.1. The graphics card driver installed in our system is Nvidia Unix x86_64 kernel module 169.09. The GeForce 8800 Ultra has 768 MB DRAM, a core clock of 612 MHz, a stream processor clock of 1.5 GHz, a memory clock of 1080 MHz, and an impressive bandwidth of 103.7 GB/s. It consists of 16 SMs, each containing 8 cores, making up a total of 128 cores. Each core can perform a multiply-add operation in a single cycle, which accounts for two floating-point operations (Flops). Therefore, it offers a peak multiply-add rate of $2 \times 1.5 \times 128 = 384$ GFlops/s (not counting the extra MUL operation that cores can issue only under certain circumstances).

For comparison, we ran our CPU experiments in three different settings:

1. Serial C++ code on Intel Core 2 Duo T2400 1.83 Ghz with 1 GB RAM running Windows XP. Two cores share a 2 MB L2 Cache.

2. Serial C++ code on AMD Opteron 8214 2.2 Ghz with 64 GB RAM running Linux kernel 2.6.18. Each core has a private 1 MB L2 cache.

3. Parallel Cilk++ code on a Numa machine (Neumann) with 64 GB RAM, and 8 dual-core Opteron processors clocked at 2.2 Ghz.

7.4.2 Implementation Details

We implemented both the recursive and the iterative algorithm on the GPU using Cuda. For the recursive algorithm, we experimented with two different versions: one that uses a simple GEMM kernel, and one that uses the optimized GEMM routine of Volkov [204]. When reporting experimental results, we call the latter *recursive optimized*. Both recursive codes implement the same algorithm given in Figure 7.3. Our recursive Cuda code is freely available at `http://gauss.cs.ucsb.edu/~aydin/apsp_cuda.html`.

Our iterative APSP implementation uses a logical 2D partitioning of the whole adjacency matrix. Such a decomposition was previously employed by Jenq and Sahni on a hypercube multiprocessor [125], and found to be more effective than 1D partitioning. However, keep in mind that there is no explicit data partitioning, only a logical mapping of submatrices to thread blocks. Host code invokes the kernel n times, where each thread block does a rank-1 update to its submatrix

per invocation. An initial snapshot of the execution is illustrated in Figure 7.7 from the viewpoint of $(2, 2)$ thread block.

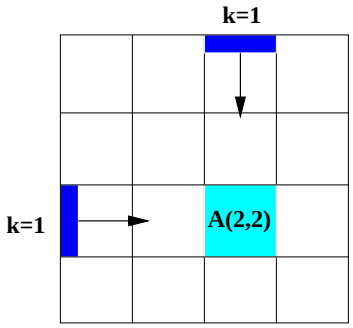


Figure 7.7: A shapshot from the execution of the iterative algorithm

Our serial iterative and recursive implementations run on the CPU as references. The iterative implementation is the standard implementation of FW, as shown in Figure 7.1. The recursive implementation is based on our recursive formulation shown in Figure 7.3. The recursive implementation stops the recursion when the submatrices completely fit into L1-cache to achieve better results.

Our reference parallel implementation runs on Neumann, a Numa machine with a total of 16 processor cores (8 dual-core 2.2 Ghz Opterons). We used Cilk++ [62] to parallelize our code, which enabled speedups up to 15x.

7.4.3 Performance Results

Timings for our APSP implementations on Cuda are given in Table 7.1. Please note the orders of magnitude difference among implementations.

Table 7.1: GPU timings on GeForce 8800 Ultra (in milliseconds)

Num. of Vertices	Iterative	Recursive	Recursive Optimized
512	2.51×10^2	1.62×10^1	6.43×10^0
1024	2.42×10^3	1.00×10^2	2.44×10^1
2048	4.60×10^4	7.46×10^2	1.41×10^2
4096	4.13×10^5	5.88×10^3	1.01×10^3
8192	5.47×10^6	5.57×10^4	7.87×10^3

Among our reference implementations, the best CPU performance is obtained on the Intel Core 2 Duo, even though the processor had a slower clock speed than the Opteron. We attribute this difference to the superior performance of MS Visual Studio's C++ compiler. Full listings of timings obtained on two different CPUs and various compilers can be found in Appendix B. Table 7.2 shows the speedup of various GPU implementations with respect to the best CPU performance achieved for the given number of vertices. The results are impressive, showing up to 480x speedups over our reference CPU implementation. Using an iterative formulation, only a modest 3.1x speedup is achieved for relatively small inputs.

Table 7.2: Speedup on 8800 Ultra w.r.t. the best CPU implementation

Num. of Vertices	Iterative	Recursive	Recursive Optimized
512	3.1	48.1	121.4
1024	3.0	73.4	301.5
2048	1.3	79.6	420.7
4096	1.2	81.5	473.2
8192	0.7	67.7	479.3

Figure 7.8 shows a log-log plot of running times of 5 different implementations. Iterative CPU and recursive CPU are timings obtained by our serial code running on Intel Core 2 Duo. For the rest of this section, we will be referring to the recursive optimized code as our best GPU code.

Although all of the APSP algorithms scale as n^3, the observed exponent of the recursive GPU implementation turned out to be slightly different than theoretical values. To reveal that, we performed a least-squares polynomial data fit on the log-log data. The input size($|V|$) - running time(t) relationship is of the form $t = c|V|^n$. This can be converted to $\lg t = \lg c + n \lg |V|$, on which we can do linear data fitting. The difference shows that in practice the performance is heavily affected by the memory traffic, not just the number of arithmetic operations performed. The observed exponents and constants are reported in Table 7.3.

Our best GPU implementation still outperforms the parallelized CPU code by a factor of 17-45x, even on 16 processors. Timings are listed in Table 7.4.

Table 7.3: Observed exponents and constants for the asymptotic behaviour of our APSP implementations with increasing problem size

$t = c\|V\|^n$	CPU (Intel Core 2 Duo)		GPU (GeForce 8800 Ultra)		
	Iterative	Recursive	Iterative	Recursive	Recur. Optimized
Exponent (n)	3.02	3.23	3.62	2.94	2.59
Constant (c)	5.5×10^{-6}	1.4×10^{-6}	3.6×10^{-8}	1.5×10^{-7}	4.7×10^{-7}

Table 7.4: Performance comparison of our best (optimized recursive) GPU implementation with parallel Cilk++ code running on Neumann, using all 16 cores

Num. of Vertices	Best GPU (secs)	Parallel CPU (secs)	GPU Speedup
512	0.00643	0.113	17.5×
1024	0.0244	0.708	29×
2048	0.141	5.146	36.5×
4096	1.01	40.36	40×
8192	7.87	354.9	45×

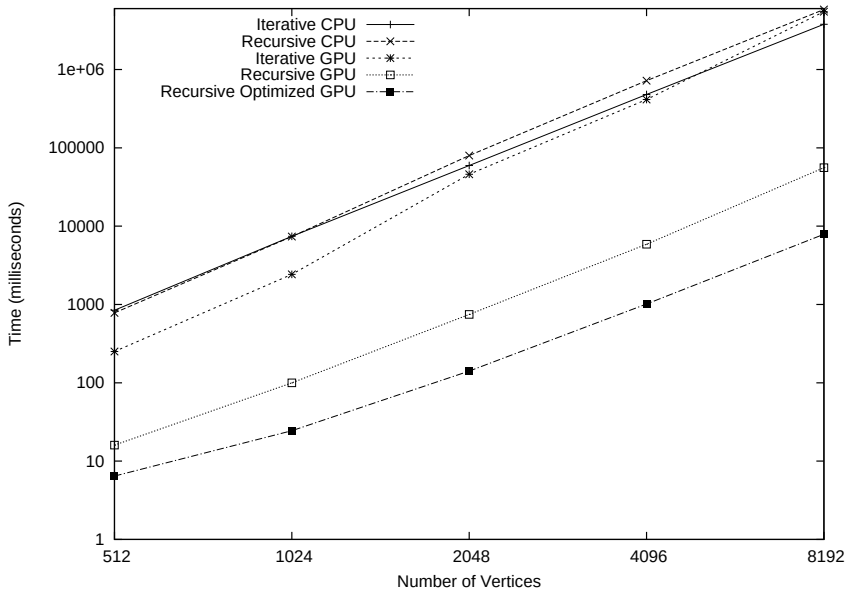


Figure 7.8: Log-log plot of absolute running times

7.4.4 Comparison with Earlier Performance Results

We compare the performance of our code with two previously reported results. One is an automatically generated, highly optimized serial program running on a 3.6 Ghz Pentium 4 CPU [115]. The other is due to Harish and Narayanan on a GPU platform very similar to ours [116]. Our GeForce 8800 Ultra is slightly faster than the GeForce 8800 GTX used by Harish and Narayanan, so we underclocked our GPU to allow a direct comparison in terms of absolute values.

On the GPU, Harish and Narayanan implemented two variants of APSP: one that uses the FW algorithm and one that runs Dijkstra's single source shortest paths (SSSP) algorithm for every vertex. For sparse graphs with $m = O(n)$,

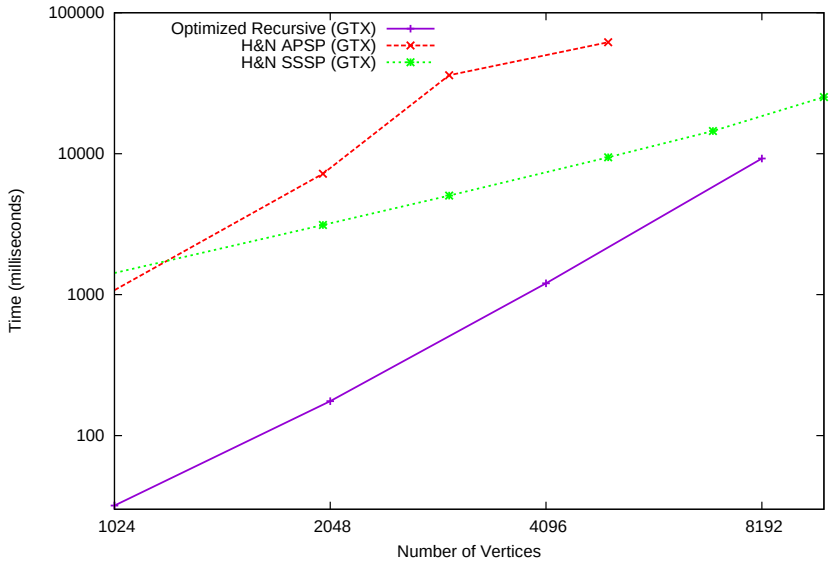


Figure 7.9: Comparison of different GPU implementations on 8800 GTX settings

the latter is theoretically faster than both the FW algorithm and our recursive formulation in the classical RAM model of computation [7]. It runs in $O(n^2 \lg n + nm)$ time using Fibonacci heaps [95].

As seen in Figure 7.9, our recursive implementation significantly outperforms both their FW implementation (H&N APSP) and Dijkstra based implementation (H&N SSSP) when implemented on a GPU. The running times for the H&N SSSP code are observed for randomly generated Erdős-Rényi graphs with an average vertex degree of 6. The running times of the other two implementations are not sensitive to sparsity. When timing our algorithm, we underclocked our GPU's clocks down to the speed of 8800 GTX for a head-to-head comparison. Due to the

adjacency matrix representation, our algorithm runs on graphs of at most 8192 vertices. Therefore, the H&N SSSP implementation is currently more favorable for large sparse graphs, although it lags behind in terms of raw speed. We plan to implement an out-of-core version of our algorithm for larger graphs. The asymptotic behavior (the slope of the curve) of the H&N SSSP implementation is also favorable but the test graphs used by them are extremely sparse, which helps the SSSP implementation whose complexity depends on the sparsity of the input.

The performance results for our iterative algorithm, given in Section 7.4.3, agree with the 2x-3x speedup over a CPU implementation achieved by H&N APSP. That implementation was also limited to 4096 vertices, while ours extends to 8192 with only a slowdown over the CPU implementation. Our best APSP code is faster than H&N APSP by a factor of 35-75x.

Comparing our results with the timings reported by Han et al. for the optimized code obtained using their auto generation tool Spiral [115], we also see significant speedups achieved by our best (optimized recursive) GPU implementation. Our comparisons are against their vectorized code (typically 4-5x faster than scalar code), and we see speedups up to 28x against Pentium 4, and 42x against Athlon 64. A detailed comparison can be found in Table 7.5. Those results also show that the GPU implementation scales better with increasing problem size, because the speedup we get over Spiral increases as the problem size increases.

Table 7.5: Comparisons of our best GPU implementation with the timings reported for Han et al. 's auto generation tool Spiral

Num. of Vertices	GFlops/s			Speedup of GeForce	
	GeForce 8800	Pentium 4	Athlon 64	Pentium 4	Athlon 64
512	38.6	5.08	3.17	7.6x	12.2x
1024	82.0	5.00	2.77	16.4x	29.6x
2048	113.5	4.78	2.73	23.7x	41.6x
4096	126.7	4.47	2.96	28.3x	42.8x

7.4.5 Scalability and Resource Usage

In this section, we try to identify the bottlenecks in our implementation in terms of resource usage and scalability. By using the NVIDIA Coolbits utility, we tweaked the frequencies of both the GPU core clock and the memory clock. The results reveal that our recursive implementation is not limited by the memory bandwidth to global GPU DRAM. For this implementation, the timings and GFlops/s rates with different clock rates are given in Table 7.6. When the memory clock is fixed, the slowdown of the computation closely tracks the slowdown of the GPU core clock (0-50% with increments of 12.5%). On the other hand, when the GPU core clock is fixed, little slowdown is observed when we underclock the memory clock. Coolbits reported the default clock speeds of 8800 Ultra as 648 Mhz for cores, and 1152 Mhz for memory, which are slightly different than the values reported in NVIDIA factsheets.

Table 7.6: Scalability of our optimized recursive GPU implementation. We tweaked core and memory clock rates using Coolbits.

$\|V\| = 4096$	GPU Clock	Memory Clock	Time (ms)	GFlops/s	Slowdown (%)
Default values	648	1152	1028.3	124.4	-
Memory clock fixed at 1152 Mhz	567	1152	1190.8	107.5	13.6
	486	1152	1362.9	93.9	24.5
	405	1152	1673.1	76.5	38.5
	324	1152	2093.7	61.1	50.8
GPU core clock fixed at 648 Mhz	648	1008	1036.2	123.5	0.7
	648	864	1047.3	122.2	1.8
	648	720	1096	116.8	6.1
	648	576	1124.9	113.8	8.5

Table 7.7: Scalability of our iterative GPU implementation. We tweaked core and memory clock rates using Coolbits.

$\|V\| = 4096$	GPU Clock	Memory Clock	Time (ms)	Slowdown (%)
Default values	648	1152	417611.4	-
Core clock halved	324	1152	418845.7	0.3
Memory clock halved	648	576	856689.7	51.2

The peak rate observed was 130 GFlops/s for $|V| = 8192$, compared to the theoretical peak of 384 GFlops. However, the theoretical peak counts 2 Flops for each fused multiply-add operation, which is not available on the tropical semiring our algorithm operates on. Therefore, the actual theoretical peak in the absence of fused multiply-add operations is 192 GFlops. Our implementation achieves more than 67% of that arithmetic peak rate for APSP.

The iterative implementation, on the other hand, is observed to be completely bandwidth bound. Even when the GPU cores are underclocked to half, no slowdown was observed. Underclocking the memory to half, however, slowed down the computation by exactly a factor of two. Exact timings can be seen in Figure 7.7. We conclude that the iterative formulation is putting too much stress on GPU memory bandwidth, consequently not harnessing the available computation power of the GPU. This is indeed expected, because the iterative formulation accesses $O(n^2)$ data and does $O(n^2)$ work in every iteration. The recursive algorithm, on the other hand, does almost all of its work in matrix multiplications, which access $O(n^2)$ data for doing $O(n^3)$ work. Therefore, it clearly has better locality of reference.

As it was not possible to disable a subset of GPU cores in the NVIDIA 8800, we do not report any scalability results with increasing number of processors.

7.4.6 Power and Economic Efficiency

Power efficiency is becoming an important consideration when comparing different architectures [90]. The Green500 list ranks supercomputers according to their Flops/Watts×sec (or Flops/Joule) ratio. In this section, we compare the power efficiency of different architectures for the APSP problem, using power specs of the manufacturer's equipment (in Thermal Watts)

Nvidia reports a peak power consumption of 175 Watts for its GeForce 8800 Ultra video card. Our dual-core Opteron (model number 8214) is reported to consume a peak power of 95 Watts, but we are using only a single core of it during serial computation. The machines used in the reported timings of automatically tuned CPU implementations are Pentium 4 (model number 560) and Athlon 64 (model 4000+). They consume 115 and 89 Watts, respectively. The Intel Core Duo T2400, the most power efficient CPU in this comparison, has a maximum power consumption of only 31 Watts even when both cores are active.

This comparative study should be considered very preliminary, because we are not running the same code in every architecture. The GPU code is assumed to use $175 + 95/2 = 222.5$ Watts as it also uses one of the CPU cores to assist the computation. This is also a rough estimate as it is likely that when one core is idle, the whole processor's power consumption is more than half of its maximum. However, our rationale is that it is possible to use the other core to perform the same computation on a different input.

The results, outlined in Table 7.8, show that the Nvidia Cuda implementation is not only powerful, but also efficient. The closest competitor is the auto generated Spiral [115] code that runs on Pentium 4. Note that Pentium 4 is not a particularly power efficient processor. Therefore, it is plausible that an auto generated code on more power efficient hardware would get closer to the efficiency

Table 7.8: Efficiency comparison of different architectures (running various codes), values in MFlops/Watts×sec (or equivalently MFlops/Joule)

$\|V\|$	Nvidia GPU	Athlon	Pentium 4	Core 2 Duo	Neumann (Opteron)
	Best Cuda code	Spiral Code		Reference FW	Cilk++ (p=16)
512	173	35.6	44.1	19.1	2.9
1024	368	31.1	43.7	17.4	3.7
2048	510	30.6	41.5	17.3	4.1
4096	569	33.2	38.8	17.2	4.2

of the GPU. A couple of factors contribute to the inefficiency of Neumann. The most important one being that the Opterons we use are not high-efficiency (HE) versions, but rather high-performance Opterons. A single Opteron core in Neumann consumes more than three times the power that is consumed by Core 2 Duo, while still giving worse performance in this particular problem.

Looking at the timings are listed in Table 7.4, the economic efficiency of the GPU is also clear. At the time of writing, the processors of our 8-way Opteron server is priced about 7x the price of Nvidia GPUs we have been using. Given that the GPU implementation runs about 17-45x faster, we see Flops/Dollar ratio of the GPU is up to 119-315x better than an 8-way server. These statements are by no means conclusive as they are based on APSP performance only.

7.5 Conclusions and Future Work

We have considered the efficient implementation of Gaussian elimination based algorithms on the GPU. Choosing the right algorithm that efficiently maps to the underlying hardware has always been important in high-performance computing. Our work shows that it is even more important when the hardware in question is a GPU. Our proof-of-concept implementation runs more than two orders of magnitude faster than a simple porting of the most popular algorithm to the GPU. The key to performance was to choose an algorithm that has good locality of reference and makes the most use of optimized kernels.

We made extensive comparisons with our reference implementations on single processor and shared memory multiprocessor systems, as well as with previously reported results obtained on various CPUs and GPUs. Future work includes identifying and implementing crucial kernels that are likely to speed up a large class of applications. Specifically, we are working on implementing an efficient sparse matrix-matrix multiplication algorithm on the GPU, which is to be used as a building block for many graph algorithms [48, 49].

Acknowledgments

We acknowledge the kind permission of Charles Leiserson and CilkArts to use an alpha release of the Cilk++ language. We also thank P.J.Narayanan and Pawan Harish for providing us the exact timings from their experiments. Sivan Toledo helped us improve the presentation of the paper with various comments. Thanks to Fenglin Liao and Arda Atali for their help during the initial implementation on Cuda. Finally, we would like to thank three anonymous reviewers for their constructive comments that improved the presentation.

Chapter 8

Conclusions and Future Directions

Our Sultan told us to come, here we came.

Mercan Dede

This thesis aims to provide a solution to the problem of graph analysis and data mining, especially for tightly-coupled computations. It proposes a scalable high-performance library along with a clear direction on the discovery and refinement of novel algorithms on this subject.

Graph computations are pervasive in sciences and it is our view that they will become more so in the future. We showed that carefully chosen and implemented primitive operations are key to high performance. This thesis specifically focused on linear algebraic primitives. This is not to claim that linear algebraic primitives are the only primitives needed to perform graph analysis and data mining; they are, however, general enough to be widely useful and compact enough to be implemented in a reasonable time frame.

This work will hopefully become a significant addition to the diversity of combinatorial scientific computing, with its unique emphasis on solving combinatorial problems using matrix methods. The contents of this thesis fall into the areas of graph algorithms, network science, parallel computing, sparse matrix algorithms, software engineering, and performance evaluation. Its main goal is, however, enabling scientific applications by facilitating large scale parallel graph analysis. The primary applications of this thesis are in computational domain sciences such as biology, ecology, chemistry, and cosmology. I hope this work will modestly contribute to the increasing interaction between domain sciences and computer sciences. As tool builders for domain sciences, computer scientists face a challenging task imposed by increasingly complex computer architectures.

This work can be extended in four main directions:

(1) *Support for higher dimensional data in the combinatorial BLAS.*

The first class citizens of the combinatorial BLAS are (sparse) matrices. Some emerging applications need a higher dimensional representation of data, where matrices are often replaced by tensors. Similarly, a more expressive way of modeling the data is to use hypergraphs, which are generalizations of graphs where the interactions among vertices is not restricted to be pairwise. Although a rectangular sparse matrix is a natural representation of an hypergraph, the right set

of primitives for computations on hypergraphs is not clear at the moment. I will be exploring these two extensions to provide a more complete library.

(2) *Hybrid MPI and shared-memory support in the combinatorial BLAS.*

Looking at the recently deployed systems targeting text and data mining, such as the Petascale Data Analysis Facility of SDSC's Triton resource, we see that a small number of shared memory nodes having large numbers of cores are connected to provide the fastest exploration of large data sets. Therefore, intra-node performance is at least as important as inter-node communication. Even though the current design of the algorithms in the combinatorial BLAS is motivated by distributed-memory systems, it would perform well in shared memory too, as it avoids hot spots and load imbalances by ensuring proper work distribution among processors. Still, algorithms specifically designed for shared-memory systems and CMPs often outperform distributed-memory algorithms running on shared memory. I plan to extend our work on compressed sparse blocks beyond sparse matrix-vector products and incorporate it into the combinatorial BLAS to achieve better performance on those hybrid architectures that require both inter-node and intra-node parallelism.

(3) *Novel algorithm development for graph and data mining.*

Many popular algorithms for graph analysis and data mining are either inexact or they have not been proven optimal. For example, it is debatable if betweenness centrality is really the right measure of importance in social networks or if the higher-order SVD and related decompositions are the right methods for revealing the hidden properties of higher order data sets. Furthermore, even well-defined applications frequently need alternative algorithms due to scaling problems that occur with ever growing data. For example, Chen and Saad [57] found that the Lanczos algorithm can be a more efficient alternative to the truncated SVD for dimensionality reduction. Since this algorithm requires only sparse matrix-vector and matrix-transpose vector products, a highly-parallel version is readily available by extending our work on compressed sparse blocks.

(4) *Usage of different metrics, such as energy and cost efficiency.*

So far, high performance computing has solely focused on the performance aspect. This work, except for the preliminary cost and energy analysis for the all-pairs shortest-paths problem in Chapter 7, is no exception. Our primary metric of success was time to solution. However, recent trends suggest that power and monetary costs of systems are exceeding what we will be able to afford [24]. Almost all aspects of computing, including software, architectures, and algorithms, need to be rethought with energy and cost efficiency in mind. Comparisons of different

graph libraries should be performed with these metrics in mind. These comparisons will also guide the healthy evolution of architectures for graph problems, since different libraries run on different architectures.

Bibliography

[1] AMD Stream Computing. `http://ati.amd.com/technology/streamcomputing`.

[2] NVIDIA CUDA. `http://www.nvidia.com/cuda`.

[3] *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. IEEE, 2009.

[4] M. Adams and J. W. Demmel. Parallel multigrid solver for 3d unstructured finite element problems. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 27, New York, NY, USA, 1999. ACM.

[5] M. D. Adams and D. S. Wise. Seven at one stroke: results from a cache-oblivious paradigm for scalable matrix algorithms. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 41–50, New York, NY, USA, 2006. ACM.

[6] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[7] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman, Boston, MA, USA, 1974.

[8] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory bfs implementation, 2007.

[9] Allinea Software Ltd., Warwick, UK. *DDT Software and Users Guide*, 2009. Available from `http://www.allinea.com/`.

[10] G. Almási, C. Caşcaval, J. G. Casta nos, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren, Jr. Dissecting cyclops: a detailed analysis of a multithreaded architecture. *SIGARCH Comput. Archit. News*, 31(1):26–38, 2003.

[11] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. Technical report, Ithaca, NY, USA, 1993.

[12] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[13] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006.

[14] M. Aspnäs, A. Signell, and J. Westerholm. Efficient assembly of sparse matrices using hashing. In B. Kågström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *PARA*, volume 4699 of *Lecture Notes in Computer Science*, pages 900–907. Springer, 2006.

[15] B. W. Bader and T. G. Kolda. Efficient Matlab computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[16] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse. HPCS scalable synthetic compact applications #2. version 1.1.

[17] D. Bader, J. Gilbert, J. Kepner, and K. Madduri. Hpc graph analysis benchmark). `http://www.graphanalysis.org/benchmark`.

[18] D. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proc. 35th Int'l. Conf. on Parallel Processing (ICPP 2006)*, pages 539–550. IEEE Computer Society, Aug. 2006.

[19] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In A. Bonato and F. R. K. Chung, editors, *WAW*, volume 4863 of *Lecture Notes in Computer Science*, pages 124–137. Springer, 2007.

[20] D. A. Bader and K. Madduri. Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2. In *Proc. 35th Int'l. Conf. on Parallel Processing (ICPP 2006)*, pages 523–530, Washington, DC, USA, Aug. 2006. IEEE Computer Society.

[21] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[22] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[23] B. W. Barrett, J. W. Berry, R. C. Murphy, and K. B. Wheeler. Implementing a portable multi-threaded graph library: The mtgl on qthreads. In *IPDPS* [3], pages 1–8.

[24] L. A. Barroso and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009.

[25] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[26] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[27] P. Becker. *The C++ Standard Library Extensions: A Tutorial and Reference*. Addison-Wesley Professional, 2006.

[28] M. A. Bender, G. S. Brodal, R. Fagerberg, R. Jacob, and E. Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 61–70, New York, NY, USA, 2007. ACM.

[29] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[30] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *IPDPS*, pages 1–14. IEEE, 2007.

[31] J. R. Black, C. U. Martel, and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In *Algorithm Engineering*, pages 37–48, 1998.

[32] G. E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[33] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[34] G. E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3), Mar. 1996.

[35] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[36] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[37] J. Bolz, I. Farmer, E. Grinspun, and P. Schrder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics*, 22(3):917–924, 2003.

[38] D. Bonachea. The inadequacy of the mpi 2.0 one-sided communication api for implementing parallel global address-space languages.

[39] D. Bonachea. Gasnet specification, v1.1. Technical Report CSD-02-1207, University of California at Berkeley, Berkeley, CA, USA, 2002.

[40] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[41] E. A. Brewer and B. C. Kuszmaul. How to get good performance from the cm-5 data network. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 858–867, Washington, DC, USA, 1994. IEEE Computer Society.

[42] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial: second edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[43] G. S. Brodal. Finger search trees. In D. Mehta and S. Sahni, editors, *Handbook of Data Structures and Applications*, chapter 11. CRC Press, 2005.

[44] J. C. Brodman, B. B. Fraguela, M. J. Garzarn, and D. Padua. New abstractions for data parallel programming. In *HotPar '09: Proc. 1st USENIX Workshop on Hot Topics in Parallelism*, mar 2009.

[45] E. D. Brooks III and K. H. Warren. The 1991 MPCI yearly report: The attack of the killer micros. Technical Report UCRL-ID-107022, Lawrence Livermore National Laboratory, March 1991.

[46] M. R. Brown and R. E. Tarjan. A fast merging algorithm. *Journal of the ACM*, 26(2):211–226, 1979.

[47] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In F. M. auf der Heide and M. A. Bender, editors, *SPAA*, pages 233–244. ACM, 2009.

[48] A. Buluç and J. R. Gilbert. Challenges and advances in parallel sparse matrix-matrix multiplication. In *ICPP '08: Proc. of the Intl. Conf. on Parallel Processing*, pages 503–510, Portland, Oregon, USA, September 2008.

[49] A. Buluç and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *IPDPS*, pages 1–11. IEEE, 2008.

[50] A. Buluç, J. R. Gilbert, and C. Budak. Solving path problems on the GPU. *Parallel Computing*, In Press, Corrected Proof, 2009.

[51] N. Burrus, A. Duret-Lutz, R. Duret-lutz, T. Geraud, D. Lesage, and R. Poss. A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *In Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL*, 2003.

[52] L. E. Cannon. *A cellular computer to implement the kalman filter algorithm.* PhD thesis, Montana State University, 1969.

[53] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[54] U. Catalyurek and C. Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*, page 118, Washington, DC, USA, 2001. IEEE Computer Society.

[55] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Computing*, 10:673–693.

[56] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. In M. W. Berry, U. Dayal, C. Kamath, and D. B. Skillicorn, editors, *SDM*. SIAM, 2004.

[57] J. Chen and Y. Saad. Lanczos vectors versus singular vectors for effective dimension reduction. *IEEE Trans. Knowl. Data Eng.*, 21(8):1091–1103, 2009.

[58] C. Chevalier and I. Safro. Comparison of coarsening schemes for multilevel graph partitioning. In *Learning and Intelligent Optimization: Third International Conference, LION 3. Selected Papers*, pages 191–205, Berlin, Heidelberg, 2009. Springer-Verlag.

[59] J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan, and G. Wesley. Stanford webbase components and applications. *ACM Transactions on Internet Technology*, 6(2):153–186, 2006.

[60] R. A. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *SODA '06: Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm*, pages 591–600, New York, NY, USA, 2006. ACM.

[61] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. A. van de Geijn. Parallel implementation of BLAS: General techniques for Level 3 BLAS. *Concurrency: Practice and Experience*, 9(9):837–857, 1997.

[62] Cilk Arts, Inc., Burlington, MA. *Cilk++ Programmer's Guide*, 2009. Available from `http://www.cilk.com/`.

[63] E. Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2(4):307–332, 1998.

[64] J. Cohen. Graph twiddling in a mapreduce world. *Computing in Science and Engg.*, 11(4):29–41, 2009.

[65] T. F. Coleman, A. Edenbrandt, and J. R. Gilbert. Predicting fill for sparse orthogonal factorization. *J. ACM*, 33(3):517–532, 1986.

[66] J. O. Coplien. Curiously recurring template patterns. *C++ Rep.*, 7(2):24–27, 1995.

[67] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing*, pages 1–6, New York, NY, USA, 1987. ACM Press.

[68] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*, pages 168–170. The MIT Press, second edition, 2001.

[69] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[70] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference*, pages 157–172, New York, NY, USA, 1969. ACM.

[71] P. D'Alberto and A. Nicolau. R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. *Algorithmica*, 47(2):203–213, 2007.

[72] W. J. Dally. Keynote address: "Stream programming : Parallel processing made simple". In *ICPP '08: Proc. of the Intl. Conf. on Parallel Processing*. IEEE Computer Society, September 2008.

[73] T. A. Davis. University of Florida sparse matrix collection. *NA Digest*, 92, 1994.

[74] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[75] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[76] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, and F. M. Der. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal of Computing*, 23(4):738–761, 1994.

[77] J. Dongarra. Sparse matrix storage formats. In Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors, *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM, 2000.

[78] J. Dongarra and T. Haigh. Biographies. *IEEE Annals of the History of Computing*, 30:74–81, 2008.

[79] J. Dongarra, P. Koev, and X. Li. Matrix-vector and matrix-matrix multiplication. In Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors, *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM, 2000.

[80] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[81] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[82] S. V. Dongen. Graph clustering via a discrete uncoupling process. *SIAM Journal on Matrix Analysis and Applications*, 30(1):121–141, 2008.

[83] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, New York, 1986.

[84] I. S. Duff and J. K. Reid. Some design features of a sparse matrix code. *ACM Transactions on Mathematical Software*, 5(1):18–35, 1979.

[85] R. Duncan. A survey of parallel computer architectures. *Computer*, 23:5–16, 1990.

[86] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package I: The symmetric codes. *International Journal for Numerical Methods in Engineering*, 18:1145–1151, 1982.

[87] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[88] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6(1):290–297, 1959.

[89] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 133–137, New York, 2004. ACM.

[90] W. Feng, X. Feng, and R. Ge. Green supercomputing comes of age. *IT Professional*, 10(1):17–23, 2008.

[91] J. Feo, D. Harper, S. Kahan, and P. Konecny. Eldorado. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 28–34, New York, NY, USA, 2005. ACM.

[92] J. Fineman. Fundamental graph algorithms. In J. Kepner and J. Gilbert, editors, *Graph Algorithms in the Language of Linear Algebra*, Fundamentals of Algorithms. SIAM, Philadelphia. in press.

[93] J. G. Fletcher. A more general algorithm for computing closed semiring costs between vertices of a directed graph. *Communications of the ACM*, 23(6):350–351, 1980.

[94] T. R. P. for Statistical Computing. `http://www.r-project.org/`.

[95] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[96] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, March 1977.

[97] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2009)*, Calgary, Canada, Aug. 2009.

[98] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *SIGPLAN*, pages 212–223, Montreal, Quebec, Canada, June 1998.

[99] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[100] R. A. V. D. Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[101] A. George, J. R. Gilbert, and J. W. Liu, editors. *Graph Theory and Sparse Matrix Computation*. Number 56 in The IMA Volumes in Mathematics and its Applications. Springer-Verlag, Germany, 1993.

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

[103] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[104] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in Matlab: Design and implementation. *SIAM Journal of Matrix Analysis and Applications*, 13(1):333–356, 1992.

[105] J. R. Gilbert, S. Reinhardt, and V. B. Shah. A unified framework for numerical and combinatorial computing. *Computing in Science and Engineering*, 10(2):20–25, 2008.

[106] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1996.

[107] A. Grama, G. Karypis, A. Gupta, and V. Kumar. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison-Wesley, 2003.

[108] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

[109] W. Gropp. Personal communication, 2010.

[110] F. G. Gustavson. Some basic techniques for solving sparse systems of linear equations. In D. J. Rose and R. A. Willoughby, editors, *Sparse Matrices and Their Applications*, pages 41–52, New York, 1972. Plenum Press.

[111] F. G. Gustavson. Finding the block lower triangular form of a matrix. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 275–289. Academic Press, New York, 1976.

[112] F. G. Gustavson. Efficient algorithm to perform sparse matrix multiplication. *IBM Technical Disclosure Bulletin*, 20(3):1262–1264, August 1977.

[113] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978.

[114] F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM J. Res. Dev.*, 41(6):737–756, 1997.

[115] S.-C. Han, F. Franchetti, and M. Püschel. Program generation for the all-pairs shortest path problem. In *PACT '06: Proceedings of the 15th International Conference on Parallel Architectures and Compilation techniques*, pages 222–232, New York, 2006. ACM.

[116] P. Harish and P.J.Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *International Conference on High Performance Computing (HiPC 2007)*, 2007.

[117] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with CUDA. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, 2007.

[118] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing*, page 28, New York, NY, USA, 1995. ACM.

[119] B. Hendrickson and A. Pothen. Combinatorial scientific computing: The enabling power of discrete algorithms in computational science. In M. J. Daydé, J. M. L. M. Palma, A. L. G. A. Coutinho, E. Pacitti, and J. C. Lopes, editors, *VECPAR*, volume 4395 of *Lecture Notes in Computer Science*, pages 260–280. Springer, 2006.

[120] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[121] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.

[122] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[123] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing*, 64(9):1017–1026, 2004.

[124] J. Irwin, J.-M. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE '97: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments*, pages 249–256, London, UK, 1997. Springer-Verlag.

[125] J. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. In *ICPP '87: Proc. of the Intl. Conf. on Parallel Processing*, pages 713–716, 1987.

[126] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[127] J. Kepner and J. Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra*. Fundamentals of Algorithms. SIAM, Philadelphia. in press.

[128] D. E. Knuth. *The art of computer programming, volume 1 (3rd ed.): Fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[129] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25:21–29, 2005.

[130] V. Kotlyar, K. Pingali, and P. Stodghill. A relational approach to the compilation of sparse matrix programs. In *European Conference on Parallel Processing*, pages 318–327, 1997.

[131] K. Kourtis, G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *CF '08: Proceedings of the 2008 conference on Computing frontiers*, pages 87–96, New York, NY, USA, 2008. ACM.

[132] M. Kowarschik and C. Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for Memory Hierarchies*, pages 213–232, 2002.

[133] M. Krishnan and J. Nieplocha. Srumma: A matrix multiplication algorithm suitable for clusters and scalable shared memory systems. In *IPDPS*, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[134] J. Laudon. Performance/watt: the new server focus. *SIGARCH Comput. Archit. News*, 33(4):5–13, 2005.

[135] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *In Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, 1994.

[136] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.

[137] C. E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[138] C. E. Leiserson. The cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.

[139] C. E. Leiserson, Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S.-W. Yang, and R. Zak. The network architecture of the connection machine cm-5 (extended abstract). In *SPAA '92: Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 272–285, New York, NY, USA, 1992. ACM.

[140] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *PKDD*, pages 133–145, 2005.

[141] E. Lindholm, J. Nickolls, S. F. Oberman, and J. Montrym. Nvidia Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, 2008.

[142] K. P. Lorton and D. S. Wise. Analyzing block locality in Morton-order and Morton-hybrid matrices. *SIGARCH Computer Architecture News*, 35(4):6–12, 2007.

[143] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007 2007.

[144] K. Madduri, D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proc. 3rd Workshop on Multithreaded Architectures and Applications (MTAAP 2009)*. IEEE Computer Society, May 2009.

[145] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarría-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *IPDPS* [3], pages 1–8.

[146] B. M. Maggs and S. A. Poltkin. Minimum-cost spanning tree as a path-finding problem. *Information Processing Letters*, 26(6):291–293, 1988.

[147] U. Manber. *Introduction to Algorithms: A Creative Approach*, page 123. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[148] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[149] K. Mehlhorn and S. Näher. *LEDA: A platform for combinatorial and geometric computing*. Cambridge University Press, New York, NY, USA, 1999.

[150] U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies, Advanced Lectures [Dagstuhl Research Seminar, March 10-14, 2002]*, volume 2625 of *Lecture Notes in Computer Science*. Springer, 2003.

[151] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, Mar. 1966.

[152] MPI Forum. MPI: A Message-Passing Interface Standard. Version 2.2, September 4th 2009. available at: `http://www.mpi-forum.org` (Dec. 2009).

[153] J. Nieplocha, J. Nieplocha, M. Krishnan, and M. Krishnan. High performance remote memory access comunications: The ARMCI approach. *International Journal of High Performance Computing and Applications*, 20:2006, 2005.

[154] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, 2007.

[155] NVIDIA. CUDA Programming Guide 1.1, 2007. `http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf`.

[156] K. Olukotun and L. Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[157] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[158] J.-S. Park, M. Penner, and V. K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):769–782, 2004.

[159] S. C. Park, J. P. Draayer, and S.-Q. Zheng. Fast sparse matrix multiplication. *Computer Physics Communications*, 70:557–568, July 1992.

[160] S. Parter. The use of linear graphs in gauss elimination. *SIAM Review*, 3(2):119–130, 1961.

[161] G. Penn. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science*, 354(1):72–81, 2006.

[162] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 30, New York, NY, USA, 1999. ACM.

[163] S. Pissanetsky. *Sparse Matrix Technology*. Academic Press, London, USA, 1984.

[164] M. O. Rabin and V. V. Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10(4):557–567, 1989.

[165] N. Rahman and R. Raman. Analysing cache effects in distribution sorting. *Journal of Experimental Algorithmics*, 5:14, 2000.

[166] R. Raman and D. S. Wise. Converting to and from dilated integers. *IEEE Transactions on Computers*, 57(4):567–573, 2008.

[167] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[168] E. Robinson and C. Kahn. Complex graph algorithms. In J. Kepner and J. Gilbert, editors, *Graph Algorithms in the Language of Linear Algebra*, Fundamentals of Algorithms. SIAM, Philadelphia. in press.

[169] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301. IEEE, 2007.

[170] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-M. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82, New York, NY, USA, 2008. ACM.

[171] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelpha, PA, second edition, 2003.

[172] T. Saif and M. Parashar. Understanding the behavior and performance of non-blocking communications in MPI. In M. Danelutto, M. Vanneschi, and D. Laforenza, editors, *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 173–182. Springer, 2004.

[173] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[174] P. Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithmics*, 5:7, 2000.

[175] N. Sato and W. F. Tinney. Techniques for exploiting the sparsity of the network admittance matrix. *IEEE Trans. Power Apparatus and Systems*, 82(69):944–950, Dec. 1963.

[176] L. R. Scott, T. Clark, and B. Bagheri. *Scientific Parallel Computing*. Princeton University Press, Princeton, NJ, USA, 2005.

[177] R. Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400–403, 1995.

[178] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. In *Graphics Hardware 2007*, pages 97–106. ACM, 2007.

[179] V. Shah and J. R. Gilbert. Sparse matrices in Matlab*P: Design and implementation. In *International Conference on High Performance Computing*, pages 144–155, 2004.

[180] V. B. Shah. *An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity*. PhD thesis, University of California, Santa Barbara, June 2007.

[181] J. Shalf, M. Wehner, L. Oliker, , and J. Hules. The challenge of energy-efficient HPC. *SciDAC Review*, (14):50–57, 2009.

[182] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles. Millisecond-scale molecular dynamics simulations on anton. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[183] G. Shipman, T. Woodall, R. Graham, A. Maccabe, and P. Bridges. Infiniband scalability in open mpi. *Parallel and Distributed Processing Symposium, International*, 0:78, 2006.

[184] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library User Guide and Reference Manual (With CD-ROM)*. Addison-Wesley Professional, 2001.

[185] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.

[186] D. A. Spielman and S.-H. Teng. Smoothed analysis: An attempt to explain the behavior of algorithms in practice. *Communications of the ACM*, 52(10):76–84, 2009.

[187] V. Strassen. Gaussian elimination is not optimal. *Numerical Math.*, 13:354–356, 1969.

[188] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.

[189] P. Sulatycke and K. Ghose. Caching-efficient multithreaded fast multiplication of sparse matrices. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, page 117, Washington, DC, USA, 1998. IEEE Computer Society.

[190] G. Tan, V. Sreedhar, and G. Rao. Analysis and performance results of computing betweenness centrality on IBM Cyclops64. *The Journal of Supercomputing*, 2009.

[191] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[192] R. E. Tarjan. A unified approach to path problems. *Journal of the ACM*, 28(3):577–593, 1981.

[193] T. B. Team. An overview of the bluegene/l supercomputer. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–22, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[194] S.-H. Teng. Coarsening, sampling, and smoothing: Elements of the multilevel method. In *Parallel Processing*, number 105 in The IMA Volumes in Mathematics and its Applications, pages 247–276, Germany, 1999. Springer-Verlag.

[195] W. Tinney and J. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, Nov. 1967.

[196] A. Tiskin. Synchronisation-efficient parallel all-pairs shortest paths computation (work in progress), 2004. `http://www.dcs.warwick.ac.uk/~tiskin/pub/2004/apsp.ps`.

[197] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41(6):711–726, 1997.

[198] S. Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal of Matrix Analysis and Applications*, 18(4):1065–1081, 1997.

[199] Totalview Technologies, Natick, MA. *Totalview User Guide*, 2009. Available from `http://www.totalviewtech.com/`.

[200] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence*, 3(2-4):331–360, 1991.

[201] T. Ungerer, B. Robic, and J. Silc. Multithreaded processors. *The Computer Journal*, 45(3):320–348, 2002.

[202] S. van Dongen. MCL - a cluster algorithm for graphs. `http://www.micans.org/mcl/index.html`.

[203] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, 2005.

[204] V. Volkov and J. Demmel. LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[205] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

[206] R. W. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, 2003.

[207] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 307–316, New York, NY, USA, 2006. ACM.

[208] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[209] D. S. Wise and J. V. Franco. Costs of quadtree representation of nondense matrices. *Journal of Parallel and Distributed Computing*, 9(3):282–296, 1990.

[210] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/L. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2005. IEEE Computer Society.

[211] R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 254–260, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[212] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Transactions on Algorithms*, 1(1):2–13, 2005.

[213] U. Zwick. Exact and approximate distances in graphs - a survey. In *ESA '01: Proceedings of the 9th Annual European Symposium on Algorithms*, pages 33–48. Springer-Verlag, 2001.

Appendices

Appendix A

Alternative One-Sided Communication Strategies for implementing Sparse GEMM

Truely one-sided operations are the key for hiding communication as much as possible by overlapping it with computation. In our first attempt, we used GASNet, which is a language-independent, low-level networking layer [39]. GASNet is primarily designed as a compilation target for PGAS languages, not as a separate library. We were able to get it running and we observed that our preliminary results were outperforming MPI on clusters with interconnects supporting remote direct memory access (RDMA), such as Infiniband and Myrinet. Due to difficulties we faced while installing the library and starting jobs on major supercomputers, this branch of the Combinatorial BLAS is currently dormant.

The widespread support for MPI has forced us to implement our asynchronous SpGEMM algorithm using MPI's one-sided communication primitives, even though its limitations were well-known to the HPC community [38]. Note that in particular, MPI library does not support asynchronous progress directly. The control should somehow be passed back to the MPI library in order to achieve progress. Several MPI implementation use a progress thread to achieve asynchronous progress at the cost of degrading the performance [183].

Passive target is truely one-sided because it does not require involvement of the target processor. The standard [152] allows implementers to restrict passive-target remote memory access to memory that is allocated with `MPI_Alloc_mem` only. During our tests on `OpenMPI 1.3b` and `MVAPICH2 1.2`, we have not encountered any problems with exposing memory allocated with regular `malloc`.

For portability, we also provide a simple memory pool implementation to manage memory allocated with `MPI_Alloc_mem` although we have not used it in our tests.

While evaluating different MPI implementation strategies for our parallel Sparse GEMM, we also experimented with general active target synchronization (post/wait-/start/complete) method of MPI in addition to the synchronous broadcast based and the asynchronous passive target implementations reported in Chapter 3.

We were not able to run our active target implementation on larger than 121 processors because the MPI implementations started to hang inside `WIN::COMPLETE` while waiting for implicit sychronization messages. Our algorithm achieves asynchronous process in the following way:

1. In the beginning of multiplication, each processor creates windows for the input matrices **A** and **B** and start exposure epochs for them by calling `WIN::POST`.

2. During the block outer-product multiplication loop, each processor issues a remote fetch to the submatrix it needs without synchronizing with all the other processors. It starts an access epoch by calling `WIN::START` with only one target processor (the processor that owns the required submatrix) and calls `WIN::COMPLETE` to ensure the completing of the remote `GET` operation.

This methodology is slightly different than the examples given in the MPI-2 standard, which included the cases where there are no multiple `WIN::START` and `WIN::COMPLETE` calls issued by a processors. However, our interpretation of the standard and our communication with William Gropp [109] convinces us that our approach is legal. The MPI-2 standard says that for each call to `WIN::POST(Group group)`, each process in `group` must issue a matching call to `WIN::START`. It does not, however, enforce that all outstanding `WIN::POST` calls should be matched by a single call `WIN::START`. Think about the case where processors 2 and 3 both expose their windows to processor 1 by calling `WIN::POST(1)`. Processor 1 has to match these calls, and it has two options:

We think both should be valid according to the standard [152] as it contains an explicit text in the advice to users, which says "A call is a noop, and can be skipped, if the group argument is empty" in the context of `WIN::POST` and `WIN::START` functions.

```
Start(2)
Get(2)
Complete()

Start(3)
Get(3)
Complete()
```

Figure A.1: Strategy 1

```
Start(2,3)
Get(2)
Get(3)
Complete()
```

Figure A.2: Strategy 2

Figure A.3: Strategies for matching the Post calls issued by multiple processors

Appendix B

Additional Timing Results on the APSP Problem

Table B.1 shows the timings obtained on Intel Core 2 Duo, using MS Visual Studio 2003's C++ compiler. For small inputs ($|V| \leq 1024$), the recursive implementation performs better due to its cache friendliness. For larger inputs, however, the overhead of recursion starts to dominate the running time. We have also experimented with the Boost Graph Library's Floyd-Warshall implementation [184] but found it to be consistently slower than our implementations. This might be due to the overheads coming from the genericity of Boost. Therefore, we excluded its running times from our plots in the main text.

Table B.1: Serial timings on Intel Core 2 Duo (in milliseconds)

Num. of Vertices	Iterative	Recursive	Boost
512	8.43×10^2	7.81×10^2	1.37×10^3
1024	7.40×10^3	7.35×10^3	1.16×10^4
2048	5.94×10^4	7.98×10^4	9.19×10^4
4096	4.79×10^5	7.20×10^5	7.27×10^5
8192	3.77×10^6	5.82×10^6	N.A.

In Table B.2, we list the performance of our reference implementations, compiled both with GCC and Intel C/C++ compiler version 9.1 (ICC). Although Intel's compiler consistently outperformed GCC, its performance still lags behind the performance achieved by MS Visual Studio on Intel.

Table B.2: Serial timings on Opteron (in milliseconds)

Num. of Vertices	Iterative		Recursive	
	GCC	ICC	GCC	ICC
512	1.30×10^3	9.90×10^2	1.60×10^3	1.14×10^3
1024	1.07×10^4	8.31×10^3	1.34×10^4	9.74×10^3
2048	8.41×10^4	6.41×10^4	1.32×10^5	1.03×10^5
4096	6.66×10^5	5.03×10^5	1.24×10^6	1.00×10^6
8192	N.A.	3.94×10^6	N.A.	1.58×10^7