The Combinatorial BLAS: Design, Implementation, and Applications [*][†][‡]

Aydın Buluç [§¶]
High Performance Computing Research
Lawrence Berkeley National Laboratory
1 Cyclotron Road, Berkeley, CA 94720
abuluc@lbl.gov

John R. Gilbert
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106-5110
gilbert@cs.ucsb.edu

[*]Proposed running head: The Combinatorial BLAS

[†]This work was supported in part by the National Science Foundation under grant number CRI-IAD0709385 and through TeraGrid resources provided by TACC under grant number TG-CCR090036

[‡]Technical Report UCSB-CS-2010-18

[§]Part of this work is performed while the author was with the University of California, Santa Barbara

[¶]Corresponding author. Tel: 510-368-4326

Abstract

This paper 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, which can be uncovered by using the right primitives.

We describe the Parallel Combinatorial BLAS, which consists of a small but powerful set of linear algebra primitives specifically targeting graph and data mining applications. We provide an extendible library interface and some guiding principles for future development. The library is evaluated using two important graph algorithms, in terms of both performance and ease-of-use. The scalability and raw performance of the example applications, using the combinatorial BLAS, are unprecedented on distributed memory clusters.

Keywords: Mathematical Software, Graph Analysis, Software Framework, Sparse Matrices, Combinatorial Scientific Computing.

1 Introduction

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 (Asanovic et al. 2006; Brodman et al. 2009). Primitives both allow algorithm designers to think on a higher level of abstraction, and help to avoid duplication of implementation efforts.

Primitives have been successfully used in the past to enable many computing applications. The *Basic Linear Algebra Subroutines* (BLAS) for numerical linear algebra (Lawson et al. 1979) are probably the canonical example of a successful primitives package. The BLAS became widely popular following the success of LAPACK (Anderson et al. 1992). LINPACK's use of the BLAS encouraged experts (preferably the hardware vendors themselves) to implement its vector operations for optimal performance. In addition to efficiency benefits, BLAS offered portability by providing a common interface. It also indirectly encouraged structured programming. Most of the reasons for developing the BLAS package about four decades ago are generally valid for primitives today.

In contrast to numerical computing, a scalable software stack that eases the application programmer's job does not exist for computations on graphs. Some existing primitives can be used to implement a number of graph algorithms. Scan primitives (Blelloch 1990) 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 (Dean and Ghemawat 2008), but the approaches are quite unintuitive and the performance is unknown (Cohen 2009). Our work fills a crucial gap by providing primitives that can be used for traversing graphs.

The goal of having a BLAS-like library for graph computation is to support rapid implementation of graph algorithms using a small yet important subset of linear algebra operations. The library should also be parallel and scale well due to the massive size of graphs in many modern applications.

The Matlab reference implementation of the HPCS Scalable Synthetic Compact Applications graph analysis (SSCA#2) benchmark (Bader et al.) was an important step towards using linear algebra operations for implementing graph algorithms. Although this implementation 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. The parallel scaling was also limited on most parallel Matlab implementations.

In this paper, we introduce a scalable high-performance software library, the Combinatorial BLAS, to be used for graph computations on distributed memory clusters. The Combinatorial BLAS 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, which is partially achieved through ideas borrowed from the domain of parallel numerical computation.

Our library is especially useful for tightly-coupled, traversal-based computations on graphs.

The remainder of this paper is organized as follows. Section 2 summarizes existing frameworks for parallel graph and sparse matrix computations. Section 3 describes the design and the guiding principles of the Combinatorial BLAS library. Section 4 gives an overview of the software engineering techniques used in the implementation. Section 5 presents performance results from two important graph applications implemented using the Combinatorial BLAS primitives. Section 6 offers some concluding remarks as well as future directions.

2 Related Work

2.1 Frameworks for Parallel Graph Computation

This section surveys working implementations of graph computations, rather than research on parallel graph algorithms. We focus on frameworks and libraries instead of parallelization of standalone applications. The current landscape of software for graph computations is summarized in Table 1.

Table 1: High-performance libraries and toolkits for parallel graph analysis

Library/Toolkit	Parallelism	Abstraction	Offering	Scalability
PBGL (Gregor and Lumsdaine 2005)	Distributed	Visitor	Algorithms	Limited
GAPDT (Gilbert et al. 2008)	Distributed	Sparse Matrix	Both	Limited
MTGL (Berry et al. 2007)	Shared	Visitor	Algorithms	Unknown
SNAP (Bader and Madduri 2008)	Shared	Various	Both	High
Combinatorial BLAS	Distributed	Sparse Matrix	Kernels	High

The Parallel Boost Graph Library (PBGL) by Gregor and Lumsdaine (2005) 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 the sequential Boost Graph Library (Siek et al. 2001), it has a dual focus on efficiency and flexibility. It relies heavily on generic programming through C++ templates.

Lumsdaine et al. (2007) observed poor scaling of PBGL for some large graph problems. We believe that the scalability of PBGL is limited due to two main factors. The graph is distributed by vertices instead of edges, which corresponds to a one-dimensional partitioning in the sparse matrix world. We have shown (Buluç and Gilbert 2008a) that this approach does not scale to large numbers of cores. We also believe that the visitor paradigm is sometimes too low-level for scalability, because it makes the computation data driven and obstructs opportunities for optimization.

The MultiThreaded Graph Library (MTGL) (Berry et al. 2007) was originally designed for development of graph applications on massively multithreaded machines, namely Cray MTA-2 and XMT. It was later extended to run on mainstream shared-memory architectures (Barrett et al. 2009). MTGL is a significant step towards an extendible and generic parallel graph library. As of now, only preliminary performance results are published for MTGL.

The Graph Algorithm and Pattern Discovery Toolbox (GAPDT, later renamed KDT) (Gilbert et al. 2008) provides both combinatorial and numerical tools to manipulate large graphs interactively. KDT runs sequentially on Matlab or in parallel on Star-P (Shah and Gilbert 2004), a parallel dialect of Matlab. Although KDT focuses on algorithms, the underlying sparse matrix

infrastructure also exposes linear algebraic kernels. KDT, like PBGL, targets distributed-memory machines. Differently from PBGL, it uses operations on distributed sparse matrices for parallelism. KDT provides an interactive environment instead of compiled code, which makes it unique among the frameworks surveyed here. Like PBGL, KDT's main weakness is limited scalability due to its one-dimensional distribution of sparse arrays.

The Small-world Network Analysis and Partitioning (SNAP) framework (Bader and Madduri 2008) contains algorithms and kernels for exploring large-scale graphs. SNAP 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. SNAP's performance and scalability are high for the reported algorithms, but a head-to-head performance comparison with PBGL and KDT is not available.

Both MTGL and SNAP are powerful 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 commodity architectures without either distributed memory or out-of-core support. Experimental studies show that an out-of-core approach (Ajwani et al. 2007) is two orders of magnitude slower than an MTA-2 implementation for parallel breadth-first search (Bader and Madduri 2006b). 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 nor SNAP seems easily extendible to distributed memory.

2.2 Frameworks for Parallel Sparse Matrix Computation

We briefly mention some other work on parallel sparse arrays, much of which is directed at numerical sparse matrix computation rather than graph computation. Many libraries exist for solving sparse linear system and eigenvalue problems; some, like Trilinos (Heroux et al. 2005), include significant combinatorial capabilities. The Sparse BLAS (Duff et al. 2002) is a standard API for numerical matrix- and vector-level primitives; its focus is infrastructure for iterative linear system solvers, and therefore it does not include such primitives as sparse matrix-matrix multiplication (SpGEMM) and sparse matrix indexing (SpRef). Global Arrays (Nieplocha et al. 2006) is a parallel dense and sparse array library that uses a one-sided communication infrastructure portable to message-passing, NUMA, and shared-memory machines. Star-P (Shah and Gilbert 2004) and pMatlab (Kepner 2009) are parallel dialects of Matlab that run on distributed-memory message-passing machines; both include parallel sparse distributed array infrastructures.

3 Design Philosophy

3.1 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, sparse vectors. Concrete data structures for these objects are explained in detail in Section 4.

We follow some design principles of the successful PETSc package (Balay et al. 1997). We define 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 as well as to avoid inter-library and intra-library collisions, matrices and vectors can be distributed over only a subset of processors by passing restricted MPI communicators to constructors. We do not attempt to create the illusion of a flat address space; communication is handled internally 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 versions that operate on parallel objects manage communication and call the sequential versions for computation. This symmetry of function prototypes has a nice effect on interoper-

ability. 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.

3.2 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 onto 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 elementwise operations on sparse matrices, although it is tempting to define a single function prototype that accepts a *binop* parameter, the most-efficient data access pattern depends 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. 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. As a result, the sparse matrix-matrix multiplication routine (SpGEMM) and the sparse matrix-vector multiplication routine (SpMV) each have a single function prototype that accepts a parameter representing the semiring.

(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 kth frontier, the set of vertices that are discovered during the kth iteration. After the frontier expansion $\mathbf{A}^\mathsf{T}\mathbf{X}^k$, previously discovered vertices can be pruned by performing an elementwise multiplication with a matrix $\mathbf{Y}$ that includes a zero for every vertex that has been discovered before, and nonzeros elsewhere. However, this approach might not be work-efficient as $\mathbf{Y}$ 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

Table 2: 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	Dense Vector
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

doing fast lookups on **B**. Not all elementwise operations can be efficiently implemented in-place (for example elementwise addition of a sparse matrix and a dense matrix will produce a dense matrix), so we declare them as members of the dense matrix class or 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 are dictated by its the class object and its operand. Called on a sparse matrix **A** with a vector **v**, it independently scales nonzero columns (or rows) of the sparse matrix. For a row vector **v**, SCALE replaces every nonzero $\mathbf{A}(i,j)$ with $\mathbf{v}(j) \cdot \mathbf{A}(i,j)$. The parameter **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 **A**. The SCALE operation never deletes columns from **A**; deletion of columns is handled by the more expensive SPASGN function described below. Alternatively, zeroing out columns during scaling can be accomplished by performing $\mathbf{A} \cdot \text{DIAG}(\mathbf{v})$ with a sparse **v**. Here, DIAG(**v**) creates a sparse matrix with diagonal populated from the elements of **v**. Note that this alternative approach is still more expensive than SCALE, as the multiplication returns a

new matrix.

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.

4 A Reference Implementation

4.1 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 1. Although the inheritance relationships are shown in the traditional way (via inclusion polymorphism as described by Cardelli and Wegner 1985), the class

hierarchies are static, obtained by the parameterizing the base class with its subclasses as explained below.

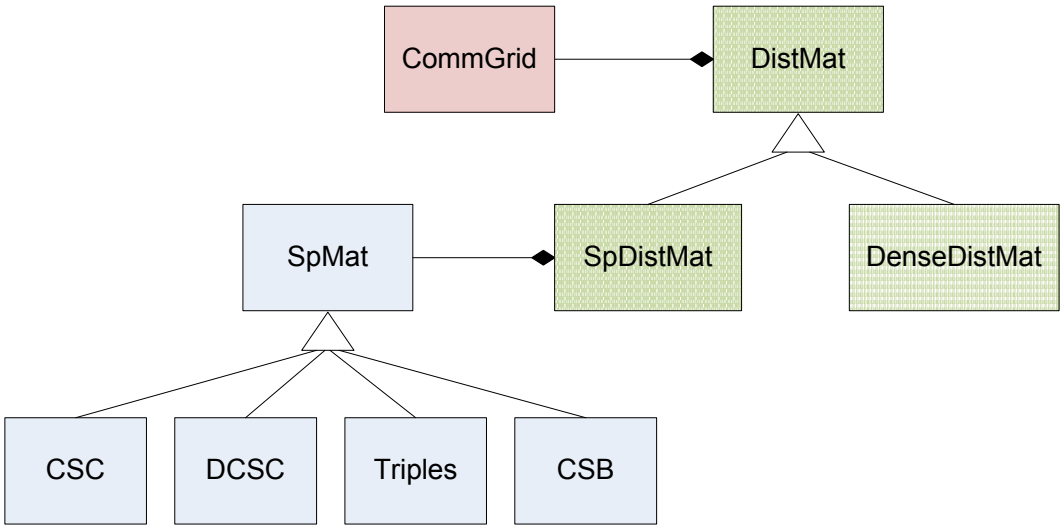

Figure 1: Software architecture for matrix classes

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 (Burrus et al. 2003) rather than expensive dynamic dispatch. A trick known as the *Curiously Recurring Template Pattern* (CRTP), a term coined by Coplien (1995), 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 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 to call the right subroutines. This requires run-time type information queries and *dynamic_cast()* operations. Also, relying on run-time operations is unsafe, as any unconforming set of parameters will lead to a run-time error or an exception. Static OOP catches any such incompatibilities at 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. The distinguishing feature of SpMat is contiguous storage of its sparse matrix, 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.

Almost all popular sparse matrix storage formats are internally composed of a number of arrays (Dongarra 2000; Saad 2003; Buluç et al. 2009), since arrays are cache friendlier than pointer-based data structures. Following this observation, 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 nonze-

```
// Abstract base class for all derived serial sparse matrix classes
// Has no data members, copy constructor or 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 constructor 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 2: Partial C++ interface of the base SpMat class

ros, starting addresses of its internal arrays and the sizes of those arrays.

The use of Essentials allows any SpDistMat object to have any SpMat object internally. 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 (Bonachea 2002) or ARMCI (Nieplocha et al. 2005), 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 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 (Barton and Nackman 1994). 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.

4.2 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 (Grama et al. 2003). Figure 3 shows this for the sparse case.

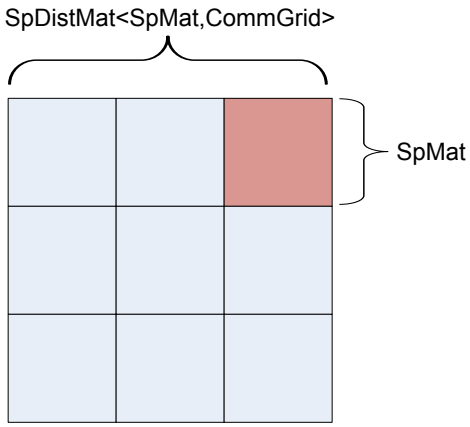

Figure 3: Distributed sparse matrix class and storage

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. We (Buluç and Gilbert 2008b; Buluç and Gilbert 2010) previously reported the problems associated with using the popular compressed sparse rows (CSR) or compressed sparse columns (CSC) representations in a 2D block decomposition. 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, explained in detail by Buluç and Gilbert (2008b). As previously mentioned, this choice is by no means exclusive and one 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 symmetric performance for 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.

5 Applications and Performance Analysis

This section 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 (Freeman 1977), 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. The betweenness centrality (BC) of a vertex is 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 1, where σ_{st} denotes the number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of such paths passing through vertex v.

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

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 (Bader et al.)) and various implementations on different platforms exist (Bader and Madduri 2006a; Madduri et al. 2009; Tan et al. 2009) for comparison.

5.2 BC Algorithm and Experimental Setup

We compute betweenness centrality using the algorithm of Brandes (2001). 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 make the exact $O(nm)$ calculation too expensive, so we resort to efficient approximations. Bader et al. (2007) propose an unbiased estimator of betweenness centrality that is based on sampling nodes from which to compute single-source shortest paths. The resulting scores approximate a uniformly scaled version of the actual betweenness centrality. We focus on unweighted graphs in this performance study.

Following the specification of the graph analysis benchmark (Bader et al.), we use R-MAT matrices as inputs. We report the performance of the approximate algorithm with 8192 starting vertices. We measure performance using the Traversed Edges Per Second (TEPS) rate, which is an algorithmic performance count that is independent of the particular implementation. 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 isolated vertices whose selection would have boosted the TEPS scores artificially.

We implemented an array-based formulation of the Brandes' algorithm due to Robinson and Kepner (Kepner and Gilbert). A reference MATLAB implementation is publicly available from the Graph Analysis webpage (Bader et al.). 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. Initially, each column of $\mathbf{X}$ has only one nonzero that represents the starting vertex of the breadth-first search. 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 (Buluç and Gilbert 2008a; Buluç 2010), because it is the most portable SpGEMM implementation and 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 pairs. The experiments are run on TACC's Lonestar cluster (lon), 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 MPI.

5.2.1 Parallel Strong Scaling

Figure 4 shows how our implementation scale for graphs of smaller size. Figure 5 shows the same code on larger graphs, with larger numbers of processors. Both results show good scaling for this challenging tightly coupled algorithm. To the best of our knowledge, ours are the first distributed

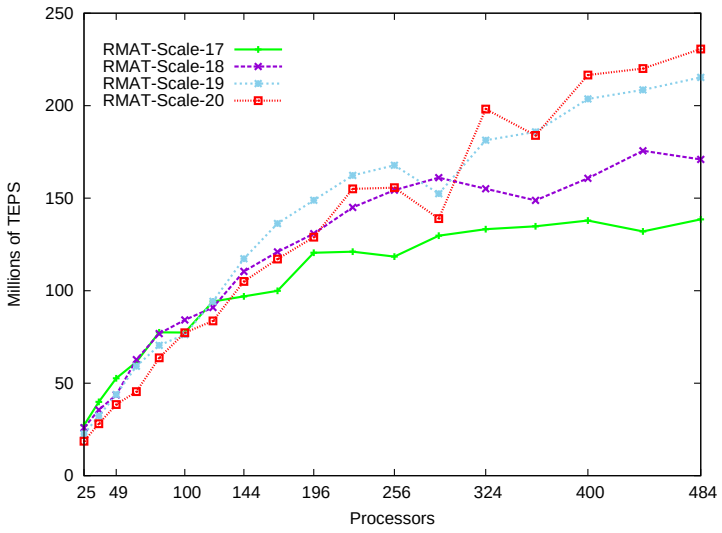

Figure 4: Parallel strong scaling of the distributed-memory betweenness centrality implementation (smaller input sizes)

memory performance results for betweenness centrality. The performance results on more than 500 processors are not smooth, but the overall upward trend is clear. Run time variability of large-scale parallel codes, which can be due to various factors such as the OS jitter (Petrini et al. 2003), is widely reported in the literature (Van Straalen et al. 2009). The expensive computation prohibited us to run more experiments, which would have smoothed out the results by averaging.

The best reported performance results for this problem are due to Madduri et al. (2009), who used an optimized implementation 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

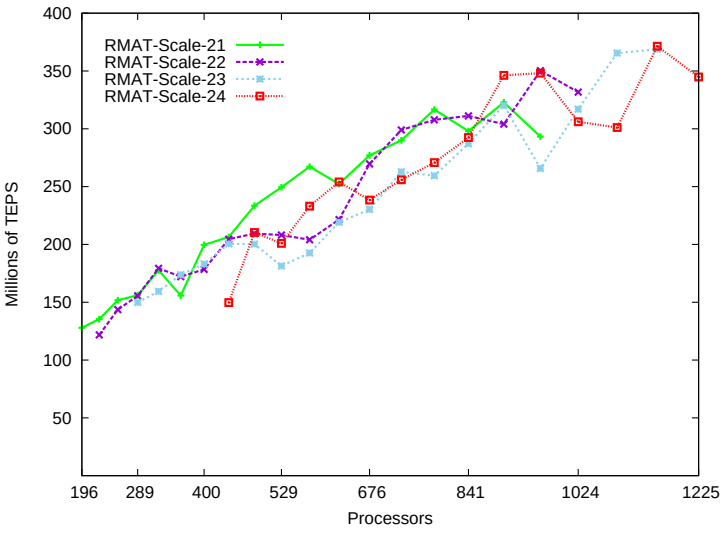

Figure 5: Parallel strong scaling of the distributed-memory betweenness centrality implementation (bigger input sizes)

no problem or machine specific optimizations. We did not 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 were equivalent to 40 processors of MTA-2.

5.2.2 Sensitivity to Batch Processing

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 can trade off space usage and potential parallelism. Space increases linearly with increasing batchsize. As we show experimentally, performance also increases substantially, especially for large numbers of processors. In Figure 6, we show the strong scaling

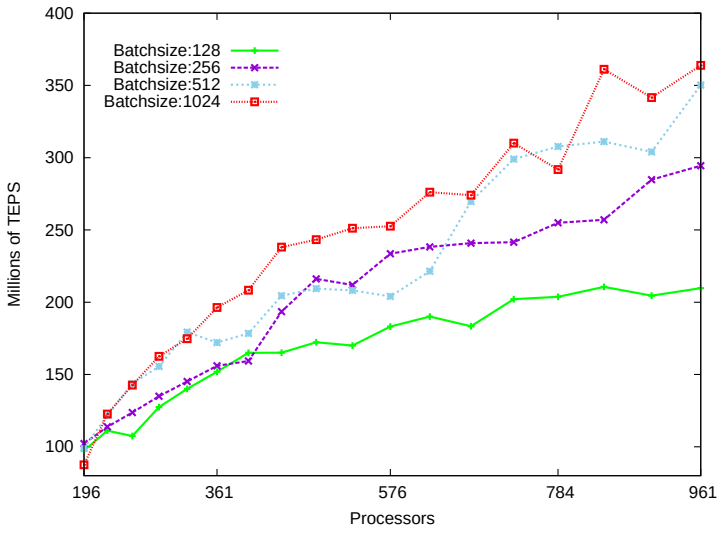

Figure 6: The effect of batch processing on the performance of the distributed-memory betweenness centrality implementation

of our betweenness 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 numbers of processors, the performance gain of using a large batchsize is 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.

```
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 7: Inflation code using the Combinatorial BLAS primitives

5.3 Markov Clustering

Markov clustering (MCL) (Van Dongen 2008) is a flow based graph clustering algorithm that has been popular in computational biology, among other fields. It simulates a Markov process to the point where clusters can be identified by a simple interpretation of the modified adjacency matrix of the graph. Computationally, it alternates between an expansion step in which the adjacency matrix is raised to its nth power (typically $n = 2$), and an inflation step in which the scalar entries are raised to the dth power ($d > 1$) and then renormalized within each column. The inflation operation boosts the larger entries and sends the smaller entries closer to zero. MCL maintains sparsity of the matrix by pruning small entries after the inflation step.

Implementing the MCL algorithm using the Combinatorial BLAS primitives generates a naturally concise code. The full MCL code, except for the cluster interpretation, is shown in Figure 8; the inflation subroutine is shown in Figure 7.

```
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 8: MCL code using the Combinatorial BLAS primitives

Van Dongen provides a fast sequential implementation of the MCL algorithm. We do not attempt an apples-to-apples comparison with the original implementation, as that software has many options, which we do not replicate in our 10-15 line prototype. Van Dongen's sequential `mcl` code is twice as fast as our parallel implementation 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 and column. However, serial performance is not a bottleneck, as our code achieves superlinear speedup until $p = 1024$.

We have been able to cluster gigascale graphs with our implementation of MCL using the Combinatorial BLAS. Here, we report on a smaller instance in order to provide a complete strong scaling result. Figure 9 shows the speedup of the three most expensive iterations, which 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. 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.

6 Conclusions and Future Work

Linear algebra has played a crucial role as the middleware between continuous physical models and their computer implementation. We have introduced the Combinatorial BLAS library as the middleware between discrete structures and their computer implementation. To accomodate

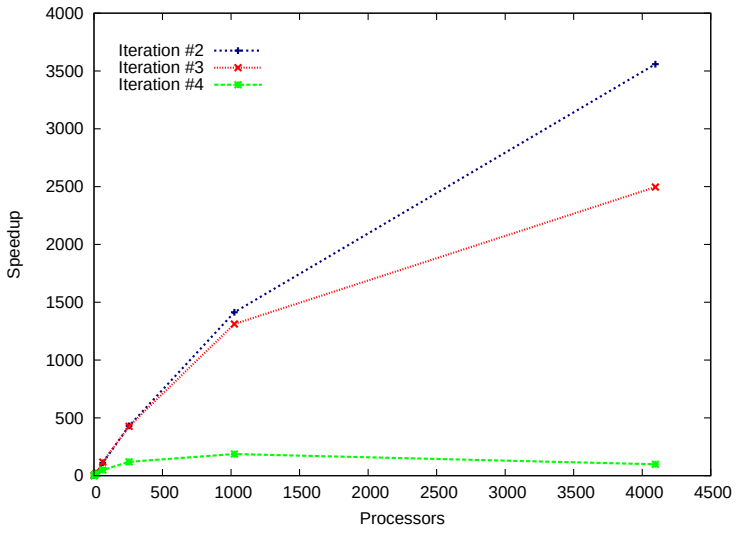

Figure 9: Strong scaling of the three most expensive iterations while clustering an R-MAT graph of scale 14 using the MCL algorithm implemented using the Combinatorial BLAS

future extensions and developments, we have avoided explicit specifications and focused on guiding principles instead.

The Combinatorial BLAS aggregates elementary operations to a level that allows optimization and load-balancing within its algebraic primitives. Our efficient parallel sparse array infrastructure, which uses a 2D compressed sparse block data representation, provides efficiency and scalability, as demonstrated by large scale experiments on two important graph applications.

One limitation of our work is that the user must cast elementary operations as semiring operators, which can sometimes be unintuitive. We plan to generalize our work to incorporate visitor/-traversal based primitives in a coherent framework. Our MPI infrastucture, albeit portable due to the widespread adoption of MPI, cannot take advantage of flexible shared-memory operations. Part

of our future work will be to leverage the hierarchical parallelism that is characteristic of current and future supercomputers.

References

Lonestar User Guide. `http://services.tacc.utexas.edu/index.php/lonestar-user-guide`.

Ajwani, D., U. Meyer, and V. Osipov (2007). Improved external memory BFS implementation. In *ALENEX*. SIAM.

Anderson, E., Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen (1992). *LAPACK's User's Guide*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Asanovic, K., R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick (2006, December). The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`.

Bader, D., 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 `http://www.highproductivity.org/SSCABmks.htm`.

Bader, D., J. Gilbert, J. Kepner, and K. Madduri. HPC graph analysis benchmark. `http://www.graphanalysis.org/benchmark`.

Bader, D. and K. Madduri (2006a). Parallel algorithms for evaluating centrality indices in real-world networks. In *Proc. 35th Int'l. Conf. on Parallel Processing (ICPP 2006)*, pp. 539–550. IEEE Computer Society.

Bader, D. A., S. Kintali, K. Madduri, and M. Mihail (2007). Approximating betweenness centrality. In A. Bonato and F. R. K. Chung (Eds.), *WAW*, Volume 4863 of *Lecture Notes in Computer Science*, pp. 124–137. Springer.

Bader, D. A. and K. Madduri (2006b, August). 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)*, Washington, DC, USA, pp. 523–530. IEEE Computer Society.

Bader, D. A. and K. Madduri (2008). SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. In *IPDPS'08: Proceedings of the 2008 IEEE International Symposium on Parallel&Distributed Processing*, pp. 1–12. IEEE Computer Society.

Balay, S., W. D. Gropp, L. C. McInnes, and B. F. Smith (1997). Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen (Eds.), *Modern Software Tools in Scientific Computing*, pp. 163–202. Birkhäuser Press.

Barrett, B. W., J. W. Berry, R. C. Murphy, and K. B. Wheeler (2009). Implementing a portable multi-threaded graph library: The MTGL on Qthreads. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pp. 1–8.

Barton, J. J. and L. R. Nackman (1994). *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Berry, J. W., B. Hendrickson, S. Kahan, and P. Konecny (2007). Software and algorithms for graph queries on multithreaded architectures. In *IPDPS*, pp. 1–14. IEEE Computer Society.

Blelloch, G. E. (1990). *Vector models for data-parallel computing*. Cambridge, MA, USA: MIT Press.

Bonachea, D. (2002). GASNet specification, v1.1. Technical Report CSD-02-1207, Computer Science Division, University of California, Berkeley.

Brandes, U. (2001). A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology 25*, 163–177.

Brodman, J. C., B. B. Fraguela, M. J. Garzarn, and D. Padua (2009, mar). New abstractions for data parallel programming. In *HotPar '09: Proc. 1st USENIX Workshop on Hot Topics in Parallelism*.

Buluç, A. (2010). *Linear Algebraic Primitives for Parallel Computing on Large Graphs*. Ph. D. thesis, University of California, Santa Barbara.

Buluç, A., J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson (2009). Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In F. M. auf der Heide and M. A. Bender (Eds.), *SPAA*, pp. 233–244. ACM.

Buluç, A. and J. R. Gilbert (2008a). Challenges and advances in parallel sparse matrix-matrix multiplication. In *ICPP'08: Proc. of the Intl. Conf. on Parallel Processing*, Portland, Oregon, USA, pp. 503–510. IEEE Computer Society.

Buluç, A. and J. R. Gilbert (2008b). On the representation and multiplication of hypersparse matrices. In *IPDPS'08: Proceedings of the 2008 IEEE International Symposium on Parallel&Distributed Processing*, pp. 1–11. IEEE Computer Society.

Buluç, A. and J. R. Gilbert (2010). Highly parallel sparse matrix-matrix multiplication. Technical Report UCSB-CS-2010-10, Computer Science Department, University of California, Santa Barbara. `http://arxiv.org/abs/1006.2183`.

Burrus, N., A. Duret-Lutz, R. Duret-Lutz, T. Geraud, D. Lesage, and R. Poss (2003). A static C++ object-oriented programming (SCOOP) paradigm mixing benefits of traditional OOP and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL)*.

Cardelli, L. and P. Wegner (1985). On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv. 17*(4), 471–523.

Cohen, J. (2009). Graph twiddling in a mapreduce world. *Computing in Science and Engg. 11*(4), 29–41.

Coplien, J. O. (1995). Curiously recurring template patterns. *C++ Rep. 7*(2), 24–27.

Dean, J. and S. Ghemawat (2008). Mapreduce: simplified data processing on large clusters. *Commun. ACM 51*(1), 107–113.

Dongarra, J. (2000). Sparse matrix storage formats. In Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst (Eds.), *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM.

Duff, I. S., M. A. Heroux, and R. Pozo (2002). An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software 28*(2), 239–267.

Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry 40*(1), 35–41.

Gilbert, J. R., S. Reinhardt, and V. B. Shah (2008). A unified framework for numerical and combinatorial computing. *Computing in Science and Engineering 10*(2), 20–25.

Grama, A., G. Karypis, A. Gupta, and V. Kumar (2003). *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Addison-Wesley.

Gregor, D. and A. Lumsdaine (2005). The Parallel BGL: A generic library for distributed graph computations. In *Workshop on Parallel Object-Oriented Scientific Computing (POOSC)*.

Heroux, M. A., R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley (2005). An overview of the Trilinos project. *ACM Trans. Math. Softw. 31*(3), 397–423.

Kepner, J. (2009). *Parallel MATLAB for Multicore and Multinode Computers*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics.

Kepner, J. and J. Gilbert (Eds.). *Graph Algorithms in the Language of Linear Algebra*. Philadelphia: SIAM. in press.

Lawson, C. L., R. J. Hanson, D. R. Kincaid, and F. T. Krogh (1979). Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software 5*(3), 308–323.

Lumsdaine, A., D. Gregor, B. Hendrickson, and J. Berry (2007, 2007). Challenges in parallel graph processing. *Parallel Processing Letters 17*(1), 5–20.

Madduri, K., D. Ediger, K. Jiang, D. Bader, and D. Chavarria-Miranda (2009, May). 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.

Nieplocha, J., B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà (2006). Advances, applications and performance of the global arrays shared memory programming toolkit. *International Journal of High Performance Computing Applications 20*(2), 203–231.

Nieplocha, J., V. Tipparaju, M. Krishnan, and D. K. Panda (2005). High performance remote memory access communication: The ARMCI approach. *International Journal of High Performance Computing Applications 20*, 2006.

Petrini, F., D. J. Kerbyson, and S. Pakin (2003). The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, pp. 55. IEEE Computer Society.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems* (Second ed.). Philadelpha, PA: SIAM.

Shah, V. and J. R. Gilbert (2004). Sparse matrices in Matlab*P: Design and implementation. In L. Bougé and V. K. Prasanna (Eds.), *HiPC*, Volume 3296 of *Lecture Notes in Computer Science*, pp. 144–155. Springer.

Siek, J. G., L.-Q. Lee, and A. Lumsdaine (2001). *The Boost Graph Library User Guide and Reference Manual (With CD-ROM)*. Addison-Wesley Professional.

Tan, G., V. Sreedhar, and G. Gao (2009). Analysis and performance results of computing betweenness centrality on IBM Cyclops64. *The Journal of Supercomputing*, 1–24.

Van Dongen, S. MCL - a cluster algorithm for graphs. `http://www.micans.org/mcl/index.html`.

Van Dongen, S. (2008). Graph clustering via a discrete uncoupling process. *SIAM Journal on Matrix Analysis and Applications 30*(1), 121–141.

Van Straalen, B., J. Shalf, T. Ligocki, N. Keen, and W.-S. Yang (2009). Scalability challenges for massively parallel AMR applications. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, pp. 1–12. IEEE Computer Society.