Factorization-based Sparse Solvers and Preconditioners

(4th Gene Golub SIAM Summer School, 2013)

Xiaoye Sherry Li
Lawrence Berkeley National Laboratory, USA
xsli@lbl.gov

Abstract

Efficient solution of large-scale, ill-conditioned and highly-indefinite algebraic equations often relies on high quality preconditioners together with iterative solvers. Because of their robustness, factorization-based algorithms play a significant role in developing scalable solvers. We discuss the state-of-the-art, high performance sparse factorization techniques which are used to build sparse direct solvers, domain-decomposition type direct/iterative hybrid solvers, and approximate factorization preconditioners. In addition to algorithmic principles, we also address the key parallelism issues and practical aspects that need to be taken under consideration in order to deliver high speed and robustness to the users of todays sophisticated high performance computers.

Contents

1 Fundamentals of parallel computing

Parallel computing has become an increasingly indispensable tool in various computing disciplines, such as modeling physical phenomena in science and engineering simulations as well as technical computing in industry. Here, we give a brief overview of the parallel architectures, programming, applications, and performance. The more thorough treatment of the subject can be found in [7, 62] and many references therein.

1.1 Parallel architectures and programming

Parallelism is ubiquitous and occurs at many levels of hierarchy in morden processor architectures. There are several different forms of parallel computing with varying granualities: bit level, instruction level, data, and task parallelism. *Pipelining* is the most fundamental form of parallelism commonly used to increase throughput. In a pipeline, the computation for an input is divided into stages with each stage running on its own spatial division of the processors. The output of one stage is the input of the next one. The different stages of the pipeline are often executed in parallel or in time-sliced fashion. The basic usages of pipeline are instruction execution, arithmetic computation and memory access. For example, the instruction circuitry can be divided into five stages: instruction fetch, instruction decode, register fetch, arithmetic, and register write back stages, wherein each stage processes one instruction at a time. This allows overlapping execution of five instructions at different stages.

Based on the pipeline principle, the *vector machines* with SIMD (Single Instruction, Multiple Data) instructions were introduced to provide high-level operations on arrays of elements. SIMD instructions perform exactly the same operations on multiple data objects, thus produce multiple results at the same time. Each instruction pipelines the operations on the individual elements of a vector. The pipeline includes the arithmetic operations (e.g., multiplication, addition, etc.) and memory access. For example, in 1999, Intel introduced the SSE (Streaming SIMD Extensions) (and SSE2 extension for double precision) in the x86 architectures. The hardware is augmented with a set of vector registers, each of length 128 bits. Each floating-point vector instruction can deliver four results of four pairs of single-precision numbers or two results of two pairs of double-precision numbers. Intel's newer SIMD instruction set is called AVX which supports 512-bit wide vectors [9]. The other examples of SIMD instruction sets include VIS (Sun Microsystem's SPARC) and

AltiVec (Apple/IBM/Freescale Semiconductor).

In recent years, GPU computing has become commonplace in scientific and enginnering applications. Here, a GPU (graphics processing unit) is used together with a CPU to accelerate computations. CPU + GPU is a powerful combination because CPUs consist of a few cores optimized for serial processing, while GPUs consist of thousands of smaller, more efficient cores designed for applications with abundance of *data parallelism.* This provides unprecedented application performance by offloading compute-intensive portions of the application to the GPU, while the remainder of the code still runs on the CPU.

The state-of-the-art massively parallel architectures usually consist of clusters of distributed memory, manycore nodes. For example, in the list of the world's top 500 fastest supercomputers (`http://www.top500.org/`), the first on the list is **Tianhe-2**, which contains 16,000 compute nodes, each comprising two Intel Ivy Bridge Xeon processors and three Xeon Phi chips (with wide SIMD paralleism). The second on the list is **Titan**, which contains 18,688 compute nodes, each comprising a 16-core AMD processor and an Nvidia Kelper GPU. Utilizing such machines requires exploiting both the coarse-level task parallelism and fine-grained data parallelism, wherein the dataset is divided into pieces each of which is stored on one compute-node. Data-parallel computations can be performed locally on each node using the locally stored data. When needed, nodes may send data to the other nodes through the interconnect fabric for cooperatively performing the same task.

The level of sophistication of programming the parallel machines above varies with different forms of parallelism. For the small degree parallelism provided by the pipeline and vector forms, the compilers usually can generate efficient codes to make full use of the hardware features. The users do not need to write explicit parallel programs. For the moderate parallelism provided by the shared memory machines, a commonly used standard programming is OpenMP [85, 86]. OpenMP is an implementation of multithreading, whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime system allocating threads to different processors.

For the massive parallelism provided by the distributed memory machines, the commonly used standard programming is MPI (Message Passing Interface) [80, 81]. MPI primarily addresses the message-passing parallel programming model: data is partitioned and distributed among the address spaces of different processes. Through cooperative operations, data is transferred from the address space of one process to that of another process.

For the cluster of distributed memory with heterogeneous nodes with multicores and/or GPUs, it is insufficient to use MPI alone. We need to use a hybrid programming model such as MPI+X, where MPI is used to generate multiple processes across multiple nodes, and X is used within the address space of each MPI process. Here, X can be OpenMP for the classic cache-based multicore processors or OpenCL [84] (or Nvidia's CUDA [25]) for the GPUs on the node.

1.2 Parallel algorithms and applications

Vast amounts of applications can benefit from different levels of parallelisms. In the early days of parallel computing, different parallel algorithms were designed and tailored for different applications. This limits the reusability of the algorithms and codes. A sustainable approach is to identify a number of patterns of communication and computation each of which is *common* to a class of applications. Colella first proposed a high level of abstraction categorizing seven such patterns (*Seven Dwarves*) for the numerical methods commonly used in scientific and engineering applications [24]: Structured grids, Unstructured grids, Dense linear algebra, Sparse linear algebra, FFT, N-body methods, and Monte Carlo. Later, the researchers at Berkeley extended the list to *Thirteen Dwarves* to capture the parallel computing patterns in broader applications [7]. The additional six dwarves are: Combinatorial logic, Graph traversal, Dynamic programming, Backtrack and branch-and-bound, Construct graphical models, and Finite state machine. The parallel hardware, software and algorithms can be designed to optimize performance for each dwarf.

1.3 Performance models and upper bounds

Given the complexity of the modern architectures, the actual runtime of the parallel algorithms and codes are extremely difficult to estimate. Despite this, the performance upper bounds can be predicted reasonably well using the following perofrmance models: the roofline model, Amdahl's law, and the latency-bandwidth model. We briefly present them below.

In numerical computing the traditional metric of analyzing an algorithm efficiency is flop count. This is far from an accurate performance predictor for the modern high-performance machines. Now performance is more dominated by memory access and inter-node communication, particularly for the algorithms involving graphs and sparse matrices. *Arithmetic Intensity* (AI) is a measure to capture both floating-point operations and the memory/network traffic; it is calculated as the ratio of floating-point operations to DRAM traffic in bytes, i.e., flops:bytes ratio. For example, the AI for Level 3 BLAS is $O(n)$ and for Levels

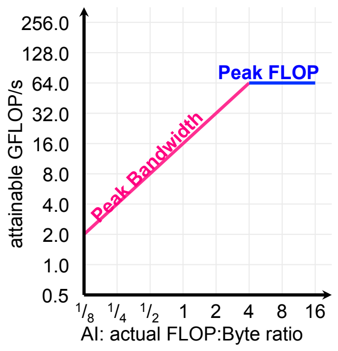


Figure 1.1: The roofline performance bound of a hyperthetocal machine (source: S. Williams)

1 and 2 BLAS is $O(1)$, where n is the matrix dimension. The AI for FFT is $O(\log n)$, where n is the number of points. A higher AI indicates more potential for data reuse in cache, and the kernel is more amenable to various code optimizations to achieve a higher percent of machine's peak performance. The *roofline model* gives a more realistic performance upper bound depending on both the computer architecture and the algorithm/code to be executed [109]. Simply put, the performance is bound to

$$\text{Attainable Performance} = \min \begin{cases} \text{Peak FLOP performance} \\ \text{Peak Bandwidth} \times AI \end{cases}$$

Fig. 1.1 depicts the peak roofline ceiling as a function of AI. Depending on how extensive the various optimization techniques are used, the actual roofline may be lower than the peak. For example, if the SSE2 instruction (128-bit SIMD) is not used on the Intel Opterons, the peak FLOP ceiling would be halved. Similarly for memory performance, if the code exhibits many random memory access patterns, the peak bandwidth ceiling would be lower.

For parallel computations *speedup* is commonly used to measure the performance gains achieved by using multiple processors, which is defined as the ratio of the sequential runtime over the parallel runtime. *Amdahl's law* gives an upper bound of the attainable speedup of a given parallel algorithm [1], independent of the machine architecture. In a parallel application, let s be the fraction of the work performed sequentially, $1-s$ is the fraction parallelized, and P is the number of cores. Then,

$$\text{Attainable Speedup} = \frac{1}{s + \frac{1-s}{P}} \leq \frac{1}{s}$$

That is, the sequential chunk of work prevents the code from scaling up no matter how many cores are used. Therefore, nearly 100% of the code needs to be parallelized in order to use millions of cores.

On the distributed memory systems, it is necessary to model the cost of transferring data between different nodes, e.g., using MPI. A commonly used cost model for network performance is *α-β model*, where α refers to the latency and β is the inverse of the bandwidth between two processors. The time to send a message of length n is roughly:

$$\text{Time} = \text{latency} + n/\text{bandwidth} = \alpha + n \times \beta \; .$$

This is a simplified and ideal model, without taking into account such practicalities as network congestion etc. For most parallel machines we have $\alpha \gg \beta \gg$ time_per_flop. For example, on a Cray XE6, time_per_flop $= 0.11ns$. Using MPI message transfer, $\alpha = 1.5\mu s \approx 13,636$ flops, and $\beta = 0.17ns \approx 12$ flops_per_double_word. Therefore, the fundamental principle is to organize the parallel algorithm so that it maintains high data locality and sends fewer long messages rather than sending many short messages.

2 Sparse matrix basics

Sparse matrices are ubiquitous in scientific and engineering calculations. A matrix is considered sparse if there are many zeros in it, and it is worth using special algorithms to perform matrix operations on it. A large class of sparse matrices arise from discretizing partial differential equations (PDE) for which the number of nonzeros in the matrix does not grow proportionally to the square of the matrix dimension (n^2), but only grows linearly w.r.t. n. Therefore, when the problem size increases, the sparsity #nonzeros/n^2 becomes smaller and sparse matrix algorithms have asymptotically lower complexity than the dense counterparts.

The first issue to address is the data structure used to store a sparse matrix. The goal is to store only the nonzeros in the matrix and to perform operations only on the nonzeros, and to handle arbitrary sparsity patterns. In the early days, the *coordinate* format (a.k.a. triplets) and the *linked list* were used. Although flexible, they are not efficient for many matrix algorithms on high performance computers—the former requires more storage than necessary and the latter prevents the algorithm from using the BLAS routines directly. Today several other compressed formats are more widely used. The most popular format is *Compressed Row Storage (CRS)* or *Compressed Column Storage (CCS)*. Let n denote the dimension of the matrix and nnz denote the number of nonzeros in the matrix. The CSR format consists of three vectors: one for floating-point numbers and the other two for integer values. The `nzval` vector of size nnz stores the nonzero values row-by-row contiguously. The `colind` vector of size nnz stores the column indices of the entries in `nzval`. The `rowptr` vector of size $n+1$ stores the position in `nzval` that starts a new row. For example, the following 7-by-7 sparse matrix

$$\begin{bmatrix} 1 & & & a & & & \\ & 2 & & & b & & \\ c & d & 3 & & & & \\ & e & & 4 & f & & \\ & & & & 5 & & g \\ & & & h & i & 6 & j \\ & & k & & l & & 7 \end{bmatrix}$$

is represented in CRS as follows:

`nzval`	1	a	2	b	c	d	3	e	4	f	5	g	h	i	6	j	k	l	7
`colind`	1	4	2	5	1	2	3	2	4	5	5	7	4	5	6	7	3	5	7

`rowptr`	1	3	5	8	11	13	17	20

.

The CCS (a.k.a. Harwell-Boeing) format is a symmetric analogue of CRS as follows:

`nzval`	1	c	2	d	e	3	k	a	4	h	b	f	5	i	l	6	g	j	7
`rowind`	1	3	2	3	4	3	7	1	4	6	2	4	5	6	7	6	5	6	7

`colptr`	1	3	6	8	11	16	17	20

.

Both formats require storage of nnz floating-point numbers and $nnz + n + 1$ integers. This is smaller than that required by the coordinate format: nnz floating-point numbers and $2 \times nnz$ integers. With the CRS storage format, the algorithm need to traverse the matrix in a row-wise fashion to ensure sequential access to the `nzval` and `colind` arrays, whereas with the CCS format, the algorithm need to traverse the matrix in a column-wise fashion.

The other sparse data structures include block-entry format, skyline or profile format, ELLPACK format and segmented-sum format, see [11] for details. The latter two are good for machines with wide SIMD instructions.

In sparse iterative solution methods, the most used operation is sparse matrix-vector multiplication (SpMV): $y = Ax$. Using the aforementioned representations for A, it is fairly straightforward to code the algorithms for $y = Ax$. However, the performance of the simple algorithms is usually rather low. The main bottlenecks are due to the low arithmetic intensity (bandwidth bound) and the strided access to the x or y vectors. A number of optimization techniques have been developed to mitigate the problems which can achieve several fold speedup relative to the baseline implementations, see [12] for many papers on this topic.

A unique aspect of sparse matrix computations is the connection to combinatorial algorithms related to graphs. An indispensible tool is the graph manipulation to reason about the nonzero structure and to transform (e.g. via reordering) the matrix to increase the performance of the numerical computation. A graph $G = (V, E)$ consists of a finite set V, called the vertex set and a finite, binary relation E on V, called the edge set. There are three standard graph models commonly used for sparse matrices.

- *Undirected graph:* The edges are unordered pairs of vertices, that is, $\{u, v\} \in E$ for some $u, v \in V$; Undirected graphs can be used to represent symmetric matrices: rows/columns correspond to the vertex set V. For each nonzero $A(i, j)$ there is an edge $\{v_i, v_j\}$.
- *Directed graph:* The edges are ordered pairs of vertices, that is, (u, v) and (v, u) are two different edges; Directed graphs can be used to represent nonsymmetric matrices.
- *Bipartite graph:* $G = (U \cup V; E)$ consists of two disjoint vertex sets U and V such that for each edge $\{u, v\} \in E, u \in U$ and $v \in V$. Bipartite graphs can be used to represent rectangular or nonsymmetric matrices.

The *degree* of a vertex v is the number of neighboring vertices connected to v. An *ordering* or labelling of $G = (V, E)$ having n vertices, i.e., $|V| = n$, is a mapping of V onto $\{1, 2, \ldots, n\}$. Very often, the sparse matrix directly coming from the physical model is not in the best ordering. We can apply various transformations by reordering the rows (equations) and columns (variables) of the matrix (linear system), which serve different purposes in different operations. In SpMV $y = Ax$, a reordering may improve access locality to the x or y vectors, and reduce communication in a parallel algorithm (see e.g. [83, 102, 104, 110].) In sparse LU factorization $A = LU$, a reordering may reduce the number of fill-ins in the L and U factored matrices (see e.g. [3, 5, 42, 45, 65, 78, 23]).

3 Direct methods for sparse linear systems

Direct methods for solving a sparse linear system $Ax = b$ are based on Gaussian elimination (GE). The matrix A is first decomposed (factorized) into a lower triangular matrix L and an upper triangular matrix U, then x is obtained by forward substitution with L followed by back substitution with U. When A is symmetric and positive definite (SPD), the Cholesky factorization $A = LL^T$ can be computed. When A is symmetric and indefinite, LDL^T can be computed. In both cases, saving is obtained by exploiting symmetry. In many situations, we need to

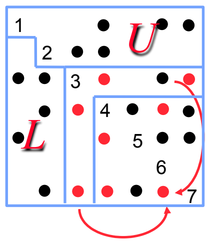


Figure 3.1: Illustration of the fill-ins in sparse GE.

solve a transformed linear system for accuracy and/or performance reasons. When A is dense we often use *partial pivoting* during GE and the resulting factorization is $PA = LU$, where P is a permutation matrix determined such that at each step of elimination the largest-magnitude entry of the column is chosen as the pivot and the corresponding row is swapped to the pivot row. Sometimes *complete pivoting* is used to swap the largest-magnitude entry of the entire trailing submatrix to the pivot position, resulting in a factorization $PAQ = LU$ with both rows and columns being permuted.

Many more complicated issues arise in sparse factorizations because of the *fill-ins*, which are the new nonzeros generated in the factored matrices L and U. Fig. 3.1 shows a 7×7 matrix with the original nonzeros in black dots and the fill-ins in red dots.

A typical sparse solver consists of the following four distinct steps:

1. An *ordering* step that reorders the rows and columns such that the factors suffer little fill, or that the matrix has special structure such as block triangular form [37, 91].
2. An analysis step or *symbolic factorization* that determines the nonzero structures of the factors and create suitable data structures for the factors.
3. A numerical factorization step that computes the L and U factors.
4. A triangular solution step that performs forward and back substitution using the factors.

There are vast varieties of algorithms associated with each step. Usually steps 1 and 2 involve only the graphs of the matrices and integer operations. Steps 3 and 4 involve floating-point operations. Step 3 is often the most time-consuming part, whereas step 4 can be orders of magnitude faster. The algorithm used in step 1 is quite independent of that used in step 3. But the algorithm in step 2 is often closely related to that of step 3. In a solver for SPD systems, the four steps can be well separated. For the most general unsymmetric systems, the solver may combine steps 2 and 3 (e.g. SuperLU) or even combine steps 1, 2 and 3 (e.g. UMFPACK) so that the numerical values also play a role in determining the elimination order.

3.1 Combinatorics

The ordering algorithms and symbolic factorizations are based on various graph models. They are simpler and faster for symmetric factorizations (e.g. Cholesky factorization) than for unsymmetric factorizations. Given certain elimination order $(v_1, v_2, \ldots, v_n)$, the purpose of the symbolic analysis is to determine the fill-in positions for the factors L and U (called the *filled graph*, denoted as $G^+(A)$), and set up the sparse compressed data structures for them. It turns out that the fill-ins can often be discovered solely based on the original graph $G(A)$. The graph tool to aid this task is *reachable set*. A node v_j is reachable from node v_i if there is a path from v_i to v_j in the graph. Let S be a subset of the vertex set, the reachable set of y through S consists of the set of vertices x such that x is reachable from y via a path $(y, v_1, \ldots, v_k, x)$ and all the intermediate vertices v_is are in S; this is denoted as $Reach(y, S)$. The following fill-path theorem by Rose and Tarjan gives precisely the edges in $G^+(A)$.

Theorem 3.1. *[94] Let $G(A) = (V, E)$ be a directed graph of A, then an edge (v, w) exists in the filled graph $G^+(A)$ if and only if*
$w \in Reach(v, \{v_1, \ldots, v_k\})$, where $v_i < \min(v, w), 1 \leq i \leq k$.

For a Cholesky factorization LL^T, the graph reachability can be computed efficiently with the aid of *elimination tree* (or etree) [99, 74]: The vertices of the tree are integers 1 through n, representing the columns of A. The first nonzero $L(j, i), j > i$ in column i defines a child-parent edge (i, j) of the tree. The etree can be computed based solely on $G(A)$ in almost linear time. The symbolic factorization algorithm can simply traverse the etree bottom-up to discover all the reachable vertices (nonzeros) from the nonzeros in the children's columns. The computational complexity is linear w.r.t. the number of nonzeros in the factor L. For unsymmetric LU factorization, the symbolic factorization involves more work. The unsymmetric analogue of the etree is the *column elimination tree*, i.e. the etree of $A^T A$ (or column etree). However, the analysis based on the column etree only gives the nonzero structure of the Cholesky factor of $A^T A$, which is an upper bound on the nonzero structure of L and U^T in $A = LU$. The precise symbolic factorization for LU has to base on the elimination DAGs of both factors L and U [49]. The computational complexity is more than linear w.r.t. the number of nonzeros in L and U, but is much lower than the flop count needed for LU factorization when employing symmetric pruning [39] and supernodes [29].

The sparsity-preserving ordering is an important research area. The goal is to solve an optimization problem of finding the best ordering of equations and variables so that the number of fill-ins in the factors is minimized. Unfortunately, finding the optimal ordering is an NP-complete problem [117]. Therefore, many heuristic algorithms for finding a good ordering have been developed. One class of algorithms is called *minimum degree* heuristic [45] for symmetric matrices. This is based on the following graph changes due to elimination: eliminating a variable/equation acts like eliminating a vertex in the associated undirected graph, and the neighboring vertices of this vertex become fully connected (called a *clique*). An edge in that clique would be a fill-in if it is not in A. Therefore, the minimium degree algorithm chooses a vertex with the smallest degree to eliminate next, which minimizes the upper bound on the fill-ins produced at that step (*local greedy* strategy). Although the basic principle is simple, the straightforward implementation is slow and requires too much memory. The main innovation for efficient implementation is the introduction of the *quotient graph* [40, 47], which represents the collection of cliques compactly throughout the elimination in space bounded by the size of $G(A)$ instead of $G^+(A)$. Another idea is to use *approximate degree* which is faster to compute than the exact degree [3, 2]. There are large numbers of research papers on the algorithmic and implementational variants and performance comparison [8, 3, 34, 47, 73, 101], In particular, George and Liu's article presented an excellent review on this subject [45].

In contrast to minimum degree, the *nested dissection* algorithm is a *global* approach based on divide and conquer paradigm. It recursively partitions the (sub)mesh via *separators*. At each level of dissection, the equations/variables associated with the two (or more) parts are first eliminated before those of the separator. From reachability argument, the vertices are not reachable between the two parts because they are separated by the higher numbered separator vertices, therefore no fill-in is generated between the two parts. At the end of recursion, the final ordering is performed such that the lower level separator nodes are ordered before the upper level ones. The recursive bisection procedure results in a complete binary (amalgamated) elimination tree, a.k.a. *separator tree*. George first introduced this for two-dimensional finite element mesh [42], and proved its *optimality* for fill-reduction: the number of nonzeros in the Cholesky factor is $O(n \log n)$ and the operation count is $O(n^{3/2})$. This optimality condition cannot be proven for the minimum degree ordering algorithm. The generalization of nested dissection to the irregular geometry is the *graph partitioning* method [72, 48]. A good algorithm is to find the separators as small as possible, so that the zero-block between the two (sub)parts are large which better preserves sparsity. There is a large body of research on finding small separators [60, 61, 65, 23, 92, 10, 18].

Several good graph partitioning packages have been developed and are publically available, including Chaco [61], ParMetis [64, 66] and PT-Scotch [89, 88].

For unsymmetric LU factorization, a local greedy strategy analogous to minimum degree was developed by Markowitz [78]: At each step of elimination, row and column permutations are performed so as to minimize the product of the number of off-diagonal nonzeros in the pivot row and pivot column (called Markowitz count). This directly minimizes the arithmetic operations and tends to minimize the number of fill-ins. Although effective, it is difficult to implement the Markowitz algorithm efficiently primarily due to two obstacles: one is the lack of compact representation like the quotient graph used in the minimum degree algorithm, and the other is the need for numerical pivoting in LU factorization which requires the Markowitz ordering algorithm to be interleaved with the numerical factorization (see MA28 [32] and MA48 [35]).

Several alternative strategies have been developed to make the ordering for LU separate from the numerical phase so that the implementation of the ordering is more efficient. When the pivots can be chosen on the diagonal, the Markowitz scheme can be implemented efficiently by using the *bipartite quotient graph* [87] and the *bi-clique* [5]. In addition, *local symmetrization* was introduced to restrict the search path of length bounded by three while searching the reachable set to update the Markowitz count. Thus the algorithm can be implemented in space bounded by the size of $G(A)$, and has the same time complexity as that of the minimum degree algorithm [5, 6].

When partial pivoting is needed, one eficient approach is to symmetrize the matrix first forming A^TA (ignoring numerial cancellation), then to apply *any* symmetric ordering algorithm to $G(A^TA)$ which gives a fill-reducing permutation Q. Then, Q is applied to the columns of A before performing the LU decomposition with row pivoting: $PAQ^T = LU$. The rationale behind this is due to the following result.

Theorem 3.2. *[46] Consider the Cholesky factorization $A^TA = R^TR$ and the LU factorization with partial pivoting $PA = LU$. For any row permutation P, struct$(L) \subset$ struct(R^T) and struct$(U) \subset$ struct(R).*

Therefore, the nonzero structure of Cholesky factor R_q in $(AQ^T)^T(AQ^T) = R_q^TR_q$ upper bounds that of L_q^T and U_q in $P(AQ^T) = L_qU_q$. Since with a good fill-reducing ordering Q, R_q contains less fill than R does, which leads to an indirect effect that L_q and U_q are likely to contain less fill than L and U. In essence, the column ordering Q tends to minimize an upper bound on the actual fill-ins in the LU factors, taking into account all the possible row permutations.

The sequential and shared-memory SuperLU solvers use this principle for sparsity ordering. There are also efficient ordering algorithms that are based on the graph $G(A^T A)$, but without forming the matrix $A^T A$ and base solely on $G(A)$: COLAMD is a minimum degree variant [28] and HUND is a nested dissection variant using the hypergraph model for unsymmetric matrices [51].

In general, the minimum degree algorithms work well for small to medium sized problems, while the nest dissection variants work better for large-scale problems. Researcher have developed the *hybrid* methods that perform nested dissection for a few levels, then uses a minimum degree algorithm for each subgraph at the bottom level of dissection [90]. Apart from the two broad classes of methods above, several other ordering algorithms were also used to permute the matrix into certain special forms, such as the maximum matching algorithm to compute the block triangular form [37, 91], the (reverse) Cuthill-McKee algorithms (CM and RCM) to reduce the bandwidth of the nonzero pattern [26, 44].

3.2 Dataflow organization, task ordering

The Gaussian elimination algorithm can be organized in different ways, such as left-looking (fan-in) or right-looking (fan-out). These variants are mathematically equivalent under the assumption that the floating-point operations are associative (approximately true), but they have very different memory access and communication patterns. The pseudo-code for the left-looking blocking algorithm is given in Algorithm 1.

Algorithm 1. *Left-looking Gaussian elimination*

```
for block $K = 1$ to $N$ do
    (1) Compute $U(1 : K-1, K)$
        (via a sequence of triangular solves)
    (2) Update $A(K : N, K) \leftarrow A(K : N, K) - L(1 : N, 1 : K-1) \cdot U(1 : K-1, K)$
        (via a sequence of calls to GEMM)
    (3) Factorize $A(K : N, K) \rightarrow L(K : N, K)$
        (may involve pivoting)
end for
```

SuperLU and SuperLU_MT use the left-looking algorithm, which has the following advantages:

- In each step, the sparsity changes are restricted within the Kth block column instead of the entire trailing submatrix, which makes it relatively easy to accommodate dynamic compressed data structures due to partial pivoting.
- There are more memory *read* operations than *write* operations in Algorithm 1. This is better for most modern cache-based computer architectures, because write tends to be more expensive in order to maintain cache coherence.

The pseudo-code for the right-looking blocking algorithm is given in Algorithm 2.

Algorithm 2. *Right-looking Gaussian elimination*

```
for block $K = 1$ to $N$ do
    (1) Factorize $A(K:N,K) \rightarrow L(K:N,K)$
        (may involve pivoting)
    (2) Compute $U(K,K+1:N)$
        (via a sequence of triangular solves)
    (3) Update $A(K+1:N,K+1:N) \leftarrow$
                $A(K+1:N,K+1:N) - L(K+1:N,K) \cdot U(K,K+1:N)$
        (via a sequence of calls to GEMM)
end for
```

SuperLU_DIST uses right-looking algorithm mainly for scalability consideration.

- The sparsity pattern and data structure can be determined before numerical factorization because of static pivoting.
- The right-looking algorithm fundamentally has more parallelism: at step (3) of Algorithm 2, all the GEMM updates to the trailing submatrix are independent and so can be done in parallel. On the other hand, each step of the left-looking algorithm involves operations that need to be carefully sequenced, which requires a sophisticated pipelining mechanism to exploit parallelism across multiple loop steps.

The multifrontal method is a variant of right-looking approach [36, 75]. Similar to the right-looking algorithm, at each elimination step, several variables (fully-summed variables) in the supernode (frontal matrix) are eliminated and the Schur complement update matrix is produced. In contrast to the right-looking algorithm where the Schur complement is updated *in place* immediately at each step, the multifrontal method postponed the update until it is time to eliminate the variables that are affected by this update matrix. Operationally, there are a number of such update matrices which are temporarily merged among themselves and stored in memory before they are finally updated into the destination Schur complement location. In other words, the Schur complement needs to be updated by a number of update matrices. The right-looking algorithm performs each update right away, whereas the multifrontal algorithm accumulates the update matrices in the *partial sums* first, and performs the updates to destination at a later stage. The multifrontal method has several advantages: it consistls of a sequence of dense matrix operations, and can use Level 3 BLAS to great extent; the partial sum of the update matrices is simply passed only between a node and its parent in the elimination/assembly tree, which eases parallel algorithm design. The biggest drawback is the large memory demand for storing the intermediate update matrices. This storage is often referred to as stack memory.

3.3 Parallelization

A great deal of effort has been invested on parallelizing the numerical factorization and triangular solution phases, because they often contribute to over 90% of the total runtime. The (column) elimination tree is a valuable tool to help design the parallel algorithms. The eliminations of the (super)nodes corresponding to the different branches of the tree can proceed independently in parallel. For the two (super)nodes situated along the same leaf-to-root path, there is a potential dependency due to the update from the descendant node to the ancestral node. Synchronization is needed among the cores owning these nodes in order to preserve the precedence relation during the computation. Intuitively, the sparse factorization should exhibits more parallelism than the dense counterpart, because both the tree-based parallelism and the fine-grained dense matrix parallelism are available. However, the parallel scaling of a sparse factorization is often hampered by many factors, including high communication-to-computation ratio and irregular communication pattern implicitly encoded in the sparse LU DAG.

On a shared memory machine there is no need to partition the matrices. Usually the parallel elimination can use an asychronous and barrier-free algorithm to schedule different types of tasks to achieve a high degree of concurrency, such as panel or frontal matrix factorization, Schur complement update, or assembly (extend-add) of the update matrices. The scheduler facilitates synchronization among different tasks to preserve task dependency and to maintain dynamic load balance. The example codes include MA41 [4], PARDISO [98], SuperLU_MT [30] and SuiteSparseQR [27]. SuperLU_MT achieved over 10-fold speedups on a number of earlier SMP machines with 16 processors [30]. Recent evaluation shows that SuperLU_MT performs very well on current multithreaded, multicore machines; it achieved over 20-fold speedup on a 16 core, 128 thread Sun VictoriaFalls [69].

The design of a distributed memory algorithm can be drastically different from a shared memory one. Many design choices are made due to the need for scalability on a large number of processors. The input sparse matrix A is usually divided by block rows, with each process having one block row represented in a local row-compressed format. This format is user-friendly and is compatible with the input interface of many other distributed memory sparse matrix software. On the other hand, the factored matrices or the intermediate frontal matrices are often distributed by a two-dimensional block cyclic layout, see e.g., SuperLU_DIST [70] and WSMP [55]. This distribution ensures that most (if not all) processors can participate in the update at each block elimination step, and also ensures that inter-process communication is restricted among the row sets or the column sets of the processes. The right-looking sparse LU factorization in SuperLU_DIST uses elimination DAGs to identify task and data dependencies, and a pipelined look-ahead scheme to overlap communication with computation. SuperLU_DIST has achieved 50- to 100-fold speedups with sufficiently large matrices, and over half a teraflops factorization rate [114].

Although the ordering and the symbolic factorization algorithms require much less time than the numerical algorithms, it is still essential to develop the distributed memory algorithms for memory scalability, because the problem size is becoming too large to fit in the memory of a single node. Parallelizing the minimum degree type of algorithm is very challenging, even for shared memory machines [22]. The nested dissection variants of algorithms exhibit more parallelism, and the notable successes with distributed memory parallel implementations include ParMETIS [66], PT-Scotch [23] and Zoltan [119].

For sparse Cholesky factorization LL^T with SPD matrices, the parallel symbolic factorization algorithm is usually designed based on the elimination tree [54, 43, 68]. The unsymmetric factorization has to base on a more complex graph model—the elimination DAGs [49]. A good parallel algorithm is presented in [52].

3.4 Available software

Researchers had developed a number of sparse direct solver packages throughout many years, which span the spectrum of different factorization methods (e.g., LU, Cholesky, LDLT, QR), and on different parallel machines (e.g., shared memory, distributed memory). The following survey article contains a table of representative codes: `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/SparseDirectSurvey.pdf`. Recently, there have been several research papers on sparse factorizations exploiting the GPU power [67, 76, 118, 97]. We will update our table as new codes become available.

4 Approximate factorizations as preconditioners

When iterative methods are used to solve the linear system, it is often necessary to solve a transformed linear system $M^{-1}Ax = M^{-1}b$, where

M is called the preconditioner. A good preconditioner M would approximate A very well so that the eigenvalue distribution of $M^{-1}A$ is better than that of A. On the other hand, we would like M to be cheap to compute and to invert. Very often the two objectives are contradicting and trade-off is desired in practice. There is a large body of research on preconditioners, see the excellent survey by Benzi [14]. Here, we present two classes of approximate factorization methods that are based on "exact" factorization, but remove some "small" entries in the factors and hence achieve better time and memory efficiency. The approximate factorizations can be used as "black-box" algebraic preconditioners for unstructured systems arising from a wide range of applications.

4.1 Incomplete factorization

A variety of incomplete LU (ILU) techniques have been studied extensively in the past, including different strategies of dropping elements, such as the level-of-fill structure-based approach (i.e., ILU(0), ILU(k)) [96], the numerical threshold-based approach [95], and the numerical inverse-based multilevel approach [15].

A standard design strategy is to start with a complete (sparse) factorization code, modify the code to drop entries by various rules. If numerical pivoting is not a concern, the level-based dropping method can be implemented efficiently by a separate symbolic factorization phase followed by the numerical phase. The symbolic factorization phase determines the nonzero structure of the factors using the incomplete fill-path theorem [63] which is an adaptation of the fill-path theorem (Theorem 3.1) for the complete factorization.

The rationale behind the level-based method is that for some problems, the higher the level of an entry is updated, the smaller it becomes. This may not be true in general. The value-based threshold method is often more robust than the level-based method, but it is harder to implement efficiently. Here, we cannot separate the symbolic and numerical phases; they must be interleaved at each step of factorization. One of the most sophisticated value-based methods is ILUTP proposed by Saad [95, 96], which combines a dual dropping strategy with numerical pivoting ("T" stands for threshold, and "P" stands for pivoting). The dual dropping rule in ILU(τ, p) first removes the elements that are smaller than τ from the current factored row or column. It then keeps only the largest p elements to control the memory requirement.

The original ILUTP algorithm was presented as a row-wise or a column-wise variant. It suffers from the same inefficiency problem as the row-wise or column-wise LU in that it lends to very little reuse of the cached data. Recently, we developed a new variant of ILUTP that exploits *supernode* in the incomplete factors. We modified the high-performance direct solver SuperLU [29] to perform incomplete factorization. To retain supernode, a delayed dropping strategy is used which first computes the entire supernode in L, then drops some rows in the supernode if they are small in certain measure, such as vector norm [53, 71]. Although the average size of the supernodes in an incomplete factor is smaller than that in a complete factor, the supernodal ILUTP can still be twice as fast as the column-wise ILU [71]. It has the combined benefits of retaining numerical robustness of ILUTP as well as achieving fast construction and application of the ILU preconditioner.

For the secondary dropping strategy, the traditional methods examines only the current column (row), and limits the number of nonzeros allowed in this column (row). We proposed an *area-based* fill control method which examines the fill ratio due to all the preceeding columns, and limits the current column size based on this dynamic fill ratio. This is shown to be more flexible and numerically robust than the column-based scheme. Furthermore, we incorporated several heuristics for adaptively modifying various threshold parameters as the factorization proceeds, which improves the robustness of the algorithm [71].

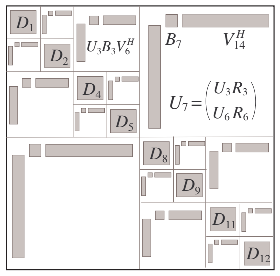
When tested with over 230 unsymmetric matrices, the supernodal, area-based adaptive ILUTP combined with GMRES can solve nearly 70% of the problems. When successful, the preconditioned iterative solver is often faster than the direct solver and uses less memory.

In general, designing an efficient ILU algorithm faces many of the similar issues as that of the complete LU algorithm. Worse yet, the ILU fatorization has even less arithmetic intensity and more sequentiality. It is extremely hard to achieve a scalable implementation. Hysom and Pothen presented a parallel approach using the domain decomposition idea [63], and showed the numerical results with 216 processors. The question remains wide open whether a parallel ILU can be designed which scales to thousands of cores, let alone millions.

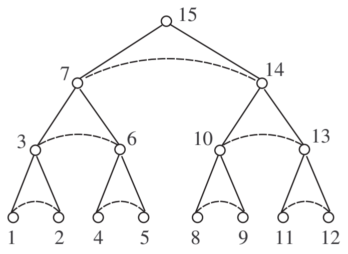
4.2 Low-rank factorization

In the last ten to fifteen years, several rank structured matrix representations have been developed, such as $\mathcal{H}$-matrices [56, 59, 57], $\mathcal{H}^2$-matrices [16, 17, 58], quasiseparable matrices [13, 38], and semiseparable matrices [20, 103]. They have been widely used in the fast solutions of the integral equations and the partial differential equations using the boundary element method.

More recently we have been developing a new class of structured sparse factorization method exploiting low-rank structures using hierarchically semi-separable (HSS) matrices [19, 77, 103, 112]. The novelty of the HSS-sparse solver is to apply the HSS compression techniques to the intermediate, dense submatrices that appear in the standard sparse



(a) 8×8 HSS matrix



(b) 8×8 HSS tree

Figure 4.1: Pictorial illustrations of a block 8×8 HSS form and the corresponding HSS tree $\mathcal{T}$.

factorization methods, such as supernodes or frontal matrices. The resulting HSS-sparse factorization can be used as a direct solver or preconditioner depending on the application's accuracy requirement and the characteristics of the PDEs. If the randomized sampling compression technique [79] is employed in compression, it can be shown that for the 3D model problems, the HSS-sparse factorization costs $O(n)$ flops for discretized matrices from certain PDEs and $O(n^{4/3})$ for broader classes of PDEs [111]. This complexity is much lower than the $O(n^2)$ cost of the traditional, exact sparse factorization method. Moreover, the new class of HSS-sparse factorizations can be applied to much broader classes of discretized PDEs (including non-selfadjoint and indefinite ones) aiming towards optimal complexity preconditioners.

Informally, the HSS representation partitions the off-diagonal blocks of a dense matrix in a hierarchical fashion; these off-diagonal blocks are approximated by compact forms, such as truncated SVD. A key property of HSS is that the orthogonal bases are desired to be *nested* following the hierarchical partitioning. This leads to asymptotically faster construction and factorization algorithms. Figure 4.1 illustrates a block 8×8 HSS representation of A, for which the hierarchical structure and the generators U_i, V_i, R_i, and B_i are succinctly depicted by the HSS tree on the right side. As a special example, its leading block 4×4 part looks like the following, where t_7 is the index set associated with node 7 of the HSS tree:

$$A|_{t_7 \times t_7} \approx \begin{pmatrix} \begin{pmatrix} D_1 & U_1 B_1 V_2^H \\ U_2 B_2 V_1^H & D_2 \end{pmatrix} & \begin{pmatrix} U_1 R_1 \\ U_2 R_2 \end{pmatrix} B_3 \begin{pmatrix} W_4^H V_4^H & W_5^H V_5^H \end{pmatrix} \\ \begin{pmatrix} U_4 R_4 \\ U_5 R_5 \end{pmatrix} B_6 \begin{pmatrix} W_1^H V_1^H & W_2^H V_2^H \end{pmatrix} & \begin{pmatrix} D_4 & U_4 B_4 V_5^H \\ U_5 B_5 V_4^H & D_5 \end{pmatrix} \end{pmatrix}.$$

With this representation, we can use the ULV factorization and the accompanying solution algorithms to solve the linear systems [21].

In [107], we developed a set of novel parallel algorithms for the key HSS operations (rank-revealing QR factorization, HSS construction, ULV factorization and HSS solution) that are used for solving dense linear systems. The parallel algorithms fully exploit both the HSS tree parallelism and dense matrix parallelism. We demonstrated that the new approach is two to 30 times faster than LU factorization; it reduces memory usage by 70- to 100-fold, using up to 8912 processing cores. Later in [106], we developed a parallel geometric HSS-embedded multifrontal sparse solver by employing the above parallel HSS algorithms to the dense frontal matrices corresponding to the separators in nested dissection. Here, we fully exploit three levels of parallelism from coarsest to finest: separator tree, HSS tree and dense matrix kernels. We tested our new parallel HSS-structured multifrontal code using up to 16,384 cores, and demonstrated that the new solver is more than 3.5 times faster than the pure multifrontal solver, and the maximum peak memory footprint is reduced by up to five-fold. Our new fast structured sparse solver has already been used successfully in several geophysics applications [108, 105]. We also show that the low-rank approximate factorization can be used as effective preconditioners for broader classes of problems [82].

5 Hybrid methods

For the three-dimensional, multiphysics, extreme scale problems, there are a number of challenges encountered by direct solvers (e.g., large amount of fill) and by iterative solvers (e.g., slow or no convergence). To mitigate these difficulties, a number of parallel hybrid (direct/iterative) solution methods have been developed [50, 41, 113, 115, 93, 33]. The domain decomposition based hybrid methods are amenable to highly scalable implementations. We present here a non-overlapping domain decomposition method called the Schur complement method (a.k.a. iterative substructuring) [100]. In this method, the original linear system $Ax = b$ is first reordered, using any parallel graph partitioning method [66, 88, 119], into a system of the following block structure:

$$\left(\begin{array}{cccc|c} D_1 & & & & E_1 \\ & D_2 & & & E_2 \\ & & \ddots & & \vdots \\ & & & D_k & E_k \\ \hline F_1 & F_2 & \cdots & F_k & C \end{array}\right) \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_k \\ y \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_k \\ g \end{pmatrix}, \qquad (5.1)$$

where D_ℓ is referred to as the ℓ-th *interior subdomain*, C consists of the *separators*, and E_ℓ and F_ℓ are the *interfaces* between D_ℓ and C. The unknowns in the interior subdomains are first eliminated using techniques from the direct solvers, and the remaining Schur complement system is solved using a preconditioned iterative solver, such as Conjugate Gradient or GMRES, that is:

$$Sy = \widehat{g} \ , \tag{5.2}$$

where the Schur complement S is given by

$$S = C - \sum_{\ell=1}^{k} F_\ell D_\ell^{-1} E_\ell \ , \tag{5.3}$$

and $\widehat{g} = g - \sum_{\ell=1}^{k} F_\ell D_\ell^{-1} f_\ell$. The final solution vector u_ℓ is obtained by solving the ℓ-th subdomain system

$$D_\ell u_\ell = f_\ell - E_\ell y \ . \tag{5.4}$$

For a symmetric positive definite matrix, it can be shown that the Schur complement has a smaller condition number than the original coefficient matrix [100]. Consequently, the preconditioned iterative solver often requires fewer iterations for the Schur complement system than for the original system. Therefore, this method has the potential of balancing the robustness of the direct solver with the efficiency of the iterative solver since the unknowns in each interior subdomain can be eliminated efficiently and in parallel, while the sparsity can be enforced for solving the Schur complement system, where most of fill may occur.

There are a number of challenges for developing a parallel algorithm and implementation that are both scalable and numerically robust. A straightforward parallel algorithm would assign one subdomain to each process, i.e. single-level parallelism. Then the number of subdomains must increase with the increasing number of processes, leading to a larger Schur complement S, an increase in the cost of solving (5.2) and often divergence of the iterative solver. It is thus imperative to exploit the *hierarchical parallelism*: The processes are divided into k subgroups. Each subgroup factorizes one subdomain D_ℓ in parallel using a parallel direct solver, either a shared memory one or a distributed memory one. All or a subset of processes participate in the iterative solution of the Schur complement system. Hence, the numbers of subdomains can be far fewer than the number of processes. In fact, we can keep a constant number of subdomains and the Schur complement size while increasing the number of processes. A good convergence is maintained regardless of the core count, see [113, 93] for details.

The second critical issue is to design the preconditioners for solving the Schur complement system. One method is to compute the inverses of the *local Schur complements* and form additive Schwarz preconditioner [50, MaPHys]. This method is more scalable but may suffer from slow convergence, espscially with a large number of subdomains. The other method involves a *global approximate Schur complement*, in which some entries are dropped while forming an approximate Schur complement $\tilde{S}$. Then, a general algebraic proconditioner can be constructed using $\tilde{S}$, such as an ILU or a low-rank approximate factorization of $\tilde{S}$ (see Section 4), to precondition GMRES for solving (5.2), e.g., [41, HIPS], [113, PDSLin], and [93, ShyLU]. The global method is numerically more robust, but hard to achieve a scalable implementation.

The parallel performance of the algorithm using hierarchical parallelism and global preconditioner depends on both intra-subgroup and inter-subgroup load balance. A number of new combinatorial problems arise in this context, such as multi-constraint graph partitioning and sparse matrix-matrix multiplication. Some progress was made in the area [116], but many open questions remain to be explored. Some new partitioning problems also arise in the Block Cimmino hybrid methods [31, 33].

Acknowledgments

Our special thanks go to Meiyue Shao from EPFL who provided tremendous help in setting up the parallel programming exercises on the local computer clusters and was on-site to help conduct the afternoon hands-on sessions. We thank Francois-Henry Rouet for helping set up the VirtualBox software to host the programming exercises and assignments on everyone's laptop computers. Patrick Amestoy, Alfredo Buttari, James Demmel, John Gilbert, Jean-Yves LExcellent, Artem Napov, Sam Williams, and Bora Uçar have graciously allowed us to use part of their lecture notes on parallel computing and sparse matrix computations.

This work was partially supported by U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract No. DE-AC02-05CH11231.

References

[1] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings (30)*, page 483485, Washington, D.C., April 1967.

[2] P. Amestoy, T. Davis, and I. Duff. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Mathematical Software*, 30(3):381–388, 2004.

[3] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Analysis and*

Applications, 17(4):886–905, 1996. Also University of Florida TR-94-039.

[4] P. R. Amestoy and I. S. Duff. Memory management issues in sparse multifrontal methods on multiprocessors. *Int. J. Supercomputer Appl.*, 7(1):64–82, Spring 1993.

[5] P. R. Amestoy, X. S. Li, and E. G. Ng. Diagonal markowitz scheme with local symmetrization. *SIAM Journal on Matrix Analysis and Applications*, 29(1):228–244, 2007.

[6] P. R. Amestoy, X. S. Li, and S. Pralet. Unsymmetric ordering using a constrained markowitz scheme. *SIAM Journal on Matrix Analysis and Applications*, 29(1):302–327, 2007.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report no. ucb/eecs-2006-183, EECS Department, Univ. of California, Berkeley, December 18 2006.

[8] C. Ashcraft. Compressed graphs and the minimum degree algorithm. *SIAM Journal on Scientific Computing*, 16:1404–1411, 1995.

[9] Intel® Architecture Instruction Set Extensions Programming Reference. `http://download-software.intel.com/sites/default/files/319433-016.pdf`, 2013.

[10] S. T. Barnard and H. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In R. F. Sincovec, D. Keyes, M. Leuze, L. Petzold, and D. Reed, editors, *Sixth SIAM conference on parallel processing for scientific computing*, pages 711–718, 1993.

[11] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: Building blocks for the iterative methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[12] Berkeley Benchmark and Optimization group. `http://bebop.cs.berkeley.edu`.

[13] T. Bella, Y. Eidelman, and V. Gohberg, I. and. Olshevsky. Computations with quasiseparable polynomials and matrices. *Theoret. Comput. Sci.*, 409:158–179, 2008.

[14] M. Benzi. Preconditioning techniques for large linear systems: A survey. *J. Comp. Phys.*, 182:418–477, 2002.

[15] M. Bollhöfer and Y. Saad. Multilevel preconditioners constructed from inverse-based ILUs. *SIAM J. Scientific Computing*, 27(5):1627–1650, 2006.

[16] S. Börm, L. Grasedyck, and W. Hackbusch. Introduction to hierarchical matrices with applications. *Eng. Anal. Bound. Elem*, 27:405–422, 2003.

[17] S. Börm and W. Hackbusch. Data-sparse approximation by adaptive $\mathcal{H}^2$-matrices. *Technical report, Leipzig, Germany: Max Planck Institute for Mathematics*, 86, 2001.

[18] T. Chan, J. Gilbert, and S.-H. Teng. Geometric spectral partitioning. Technical Report CSL-94-15, Palo Alto Research Center, Xerox Corporation, California, 1994.

[19] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals. A fast solver for hss representations via sparse matrices. *SIAM J. Matrix Anal. Appl.*, 29:67–81, 2006.

[20] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. van der Veen, and D. White. Some fast algorithms for sequentially semiseparable representations. *SIAM J. Matrix Anal. Appl.*, 27:341–364, 2005.

[21] S. Chandrasekaran, M. Gu, and T. Pals. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM J. Matrix Anal. Appl.*, 28:603–622, 2006.

[22] T.-Y. Chen, J. Gilbert, and S. Toledo. Toward an efficient column minimum degree code for symmetric multiprocessors. In *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 1999.

[23] C. Chevalier and F. Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6-8):318–331, 2008.

[24] P. Colella. Defining software requirements for scientific computing. presentation.

[25] CUDA Parallel Computing Platform. `http://www.nvidia.com/object/cuda_home_new.html`.

[26] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. ACM Nat. Conf.*, pages 157–172, New York, 1969.

[27] T. A. Davis. Algorithm 915, SuiteSparseQR: Multifrontal multithreaded rank-revealing sparse QR factorization. *ACM Trans. Mathematical Software*, 38(1), 2011. `http://www.cise.ufl.edu/research/sparse/SPQR/`.

[28] T. A. Davis, J. R. Gilbert, S. Larimore, and E. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Mathematical Software*, 30(3):353–376, 2004.

[29] J. W. Demmel, S. C. Eisenstat, J. R. Gilbert, X. S. Li, and J. W. H. Liu. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Analysis and Applications*, 20(3):720–755, 1999.

[30] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[31] L.A. Drummond, I.S. Duff, R. Guivarch, D. Ruiz, and M. Zenadi. Partitioning strategies for the block cimmino algorithm. Technical Report RAL-P-2013-010, RAL, 2013.

[32] I. S. Duff. MA28 - A set of FORTRAN subroutines for sparse unsymmetric linear equations. Technical Report AERE R-8730, Harwell, 1977.

[33] I.S. Duff, R. Guivarch, D. Ruiz, and M. Zenadi. The augmented block cimmino distributed method. Technical Report RAL-P-2013-001, RAL, 2013.

[34] I.S Duff and J. K. Reid. MA47, a Fortran code for direct solution of indefinite sparse symmetric linear systems. Technical Report RAL-95-001, Rutherford Appleton Laboratory, 1995.

[35] I.S Duff and J. K. Reid. The design of MA48, a code for the direct solution of sparse unsymmetric linear systems of equations. *ACM Trans. Mathematical Software*, 22:187–226, 1996.

[36] I.S Duff and J.K Reid. The multifrontal solution of indefinite sparse symmetric linear equations. *ACM Trans. Mathematical Software*, 9(3):302–325, September 1983.

[37] A.L. Dulmage and N.S. Mendelsohn. Two algorithms for bipartite graphs. *J. Sot. Indust. Appl. Math.*, 11:183–194, 1963.

[38] Y. Eidelman and I. Gohberg. On a new class of structured matrices. *Integral Equations Operator Theory*, 34:293–324, 1999.

[39] S. C. Eisenstat and J. W. H. Liu. Exploiting structural symmetry in sparse unsymmetric symbolic factorization. *SIAM J. Matrix Anal. Appl.*, pages 13:202–211, 1992.

[40] S.C. Eisenstat, M.H. Schultz, and A.H. Sherman. Applications of an element model for gaussian elimination. In J.R. Bunch and D.J. Rose, editors, *Sparse Matrix Computations*, pages 85–96. Academic Press, 1976.

[41] J. Gaidamour and P. Henon. HIPS: a parallel hybrid direct/iterative solver based on a schur complement. In *Proc. PMAA*, 2008.

[42] A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numerical Analysis*, 10:345–363, 1973.

[43] A. George, M. T. Heath, E. Ng, and J. W. H. Liu. Symbolic cholesky factorization on a local-memory multiprocessor. *Parallel Comput.*, 5:85–95, 1987.

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

[45] A. George and J. W. H. Liu. The evolution of the minimum degree ordering algorithms. *SIAM Review*, 31(1):1–19, March 1989.

[46] A. George and E. Ng. An implementation of Gaussian elimination with partial pivoting for sparse systems. *SIAM J. Sci. Stat. Comput.*, 6(2):390–409, 1985.

[47] J. A. George and J. W. H. Liu. A fast implementation of the minimum degree algorithm using quotient graphs. *ACM Trans. Mathematical Software*, 6:337–358, 1980.

[48] J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, pages 377–404, 1987.

[49] John R. Gilbert and Esmond G. Ng. Predicting structure in nonsymmetric sparse matrix factorizations. In A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph theory and sparse matrix computation*, pages 107–139. Springer-Verlag, New York, 1993.

[50] L. Giraud, A. Haidar, and S. Pralet. Using multiple levels of parallelism to enhance the performance of domain decomposition solvers. *Parallel Computing*, 36:285–296, 2010.

[51] L. Grigori, E. Boman, S. Donfack, and T. Davis. Hypergraph-based unsymmetric nested dissection ordering for sparse lu factorization. *SIAM J. Scientific Computing*, 32(6), 2010.

[52] L. Grigori, J. W. Demmel, and X. S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM J. Scientific Computing*, 29(3):1289–1314, 2007.

[53] A. Gupta and T. George. Adaptive techniques for improving the performance of incomplete factorization preconditioning. *SIAM J. Sci. Comput.*, 32(1):84–110, 2010.

[54] A. Gupta, F. Gustavson, M. Joshi, G. Karypis, and V. Kumar. Design and implementation of a scalable parallel direct solver for sparse symmetric positive definite systems. In *Proceedings of the*

8th SIAM Conference on Parallel Processing for Scientific Computing, Minneapolis, Minnesota, 1997.

[55] A. Gupta and V. Kumar. Optimally scalable parallel sparse cholesky factorization. In *The 7th SIAM Conference on Parallel Processing for Scientific Computing*, pages 442–447, 1995.

[56] W. Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. Part I: introduction to $\mathcal{H}$-matrices. *Computing*, 62:89–108, 1999.

[57] W. Hackbusch, L. Grasedyck, and S. Börm. An introduction to hierarchical matrices. *Math. Bohem.*, 127:229–241, 2002.

[58] W. Hackbusch, B. Khoromskij, and S. A. Sauter. On $\mathcal{H}^2$-matrices. *Lectures on applied mathematics (Munich, 1999), Springer, Berlin*, pages 9–29, 2000.

[59] W. Hackbusch and B. N. Khoromskij. A sparse $\mathcal{H}$-matrix arithmetic. Part-II: Application to multi-dimensional problems. *Computing*, 64:21–47, 2000.

[60] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing*, San Diego, CA, 1995.

[61] B. Hendrickson and R. Leland. The CHACO's User's Guide. Technical Report SAND95-2344•UC-405, Sandia National Laboratories, Albuquerque, 1995. `http://www.cs.sandia.gov/~bahendr/chaco.html`.

[62] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, Fifth Edition.* Elsevier, 2012.

[63] D. Hysom and A. Pothen. A scalable parallel algorithm for incomplete factor preconditioning. *SIAM J. Scientific Computing*, 22(6):2194–2215, 2001.

[64] G. Karypis and V. Kumar. MeTiS – serial graph partitioning and computing fill-reducing matrix ordering. University of Minnesota. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[65] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific Computing*, 20:359–392, 1998.

[66] G. Karypis, K. Schloegel, and V. Kumar. ParMeTiS: Parallel graph partitioning and sparse matrix ordering library – version 3.1. University of Minnesota, 2003. `http://www-users.cs.umn.edu/~karypis/metis/parmetis/`.

[67] G. Krawezik and G. Poole. Accelerating the ANSYS direct sparse solver with GPUs. In *Proc. Symposium on Application Accelerators in High Performance Computing (SAAHPC)*, Urbana-Champaign, IL, NCSA, 2009. `http://saahpc.ncsa.illinois.edu/09`.

[68] P.S. Kumar, M.K. Kumar, and A. Basu. A parallel algorithm for elimination tree computation and symbolic factorization. *Parallel Comput.*, 18:849–856, 1992.

[69] X. S. Li. Evaluation of sparse factorization and triangular solution on multicore architectures. In *Proceedings of VECPAR08 8th International Meeting High Performance Computing for Computational Science*, Toulouse, France, June 24-27 2008.

[70] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2):110–140, June 2003.

[71] X. S. Li and M. Shao. A supernodal approach to imcomplete LU factorization with partial pivoting. *ACM Trans. Mathematical Software*, 37(4), 2010.

[72] R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM J. Numerical Analysis*, 16:346–358, 1979.

[73] J. W. H. Liu. Modification of the minimum degree algorithm by multiple elimination. *ACM Trans. Mathematical Software*, 11:141–153, 1985.

[74] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Analysis and Applications*, 11:134–172, 1990.

[75] J. W. H. Liu. The multifrontal method for sparse matrix solution: theory and practice. *SIAM Review*, 34(1):82–109, March 1992.

[76] R. Lucas, G. Wagenbreth, D. Davis, and R. Grimes. Multifrontal computations on GPUs and their multi-core hosts. In *VECPAR'10: Proc. 9th Intl. Meeting for High Performance Computing for Computational Science*, Berkeley, CA, 2010. `http://vecpar.fe.up.pt/2010/papers/5.php`.

[77] W. Lyons. *Fast Algorithms with Applications to PDEs.* PhD thesis, University of California, Santa Barbara, 2005.

[78] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Sci.*, 3:255–269, 1957.

[79] P.G. Martinsson. A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. *SIAM J. Matrix Analysis and Applications*, 32(4):1251–1274, 2011.

[80] Message Passing Interface (MPI) forum. http://www.mpi-forum.org/.

[81] MPI tutorial. `https://computing.llnl.gov/tutorials/mpi/`.

[82] A. Napov, X.S. Li, and M. Gu. An algebraic multifrontal preconditioner that exploits the low-rank property. Technical report, Lawrence Berkeley National Laboratory, 2014. In preparation.

[83] L. Oliker, X. Li, P. Husbands, and R. Biswas. Effects of ordering strategies and programming paradigms on sparse matrix computations. *Siam Review*, 44(3):373–393, 2002.

[84] Open Computing Language. `http://www.khronos.org/opencl/`.

[85] OpenMP API Specification for Parallel Programming. `www.openmp.org`.

[86] OpenMP Tutorial. `https://computing.llnl.gov/tutorials/openMP/`.

[87] G. Pagallo and C. Maulino. A bipartite quotient graph model for unsymmetric matrices. In *Lecture Notes in Mathematics 1005, Numerical Method*, pages 227–239, Springer-Verlag, New York, 1983.

[88] F. Pellegrini. PT-Scotch and libScotch 5.1 User's Guide (version 5.1.11). INRIA Bordeaux Sud-Ouest, Université Bordeaux I. November, 2010. `http://www.labri.fr/perso/pelegrin/scotch/`.

[89] F. Pellegrini. Scotch and libScotch 5.1 User's Guide (version 5.1.11). INRIA Bordeaux Sud-Ouest, Université Bordeaux I. November, 2010. `http://www.labri.fr/perso/pelegrin/scotch/`.

[90] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing Nested Dissection and Halo Approximate Minimum Degree for Efficient Sparse Matrix Ordering. *Concurrency: Practice and Experience*, 12:69–84, 2000.

[91] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Mathematical Software*, Vol. 16:303–324, 1990.

[92] A. Pothen, H. D. Simon, L. Wang, and S. Barnard. Towards a fast implementation of spectral nested dissection. In *Supercomputing, ACM Press*, pages 42–51, 1992.

[93] S. Rajamanickam, E.G. Boman, and M.A. Heroux. ShyLU: A hybrid-hybrid solver for multicore platforms. In *Proc. of IPDPS 2012*, Shanghai, China, May 20-24 2012.

[94] D. J. Rose and R. E. Tarjan. Algorithmic aspects of vertex elimination of directed graphs. *SIAM Journal on Applied Math*, Vol. 34(No. 1):176–197, January 1978.

[95] Y. Saad. ILUT: A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4):387–402, 1994.

[96] Y. Saad. *Iterative methods for sparse linear systems.* SIAM, Philadelphia, 2004.

[97] P. Sao, R. Vuduc, and X. Li. A distributed CPU-GPU sparse direct solver. In *Euro-Par 2014*, Porto, Portugal, August 25-29, 2014.

[98] O. Schenk, K. Gärtner, and W. Fichtner. Efficient sparse LU factorization with left–right looking strategy on shared memory multiprocessors. *BIT*, 40(1):158–176, 2000.

[99] R. Schreiber. A new implementation of sparse Gaussian elimination. *ACM Trans. Mathematical Software*, 8:256–276, 1982.

[100] B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition. Parallel Multilevel Methods for Elliptic Partial Differential Equations.* Cambridge University Press, New York, 1996.

[101] W.F. Tinney and J.W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE*, 55:1801–1809, 1967.

[102] B. Uçar and C. Aykanat. Revisiting hypergraph models for sparse matrix partitioning. *SIAM Review*, 49(4):595–603, 2007.

[103] R. Vandebril, M. Van Barel, G. Golub, and N. Mastronardi. A bibliography on semiseparable matrices. *Calcolo*, 42:249–270, 2005.

[104] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[105] S. Wang, M.V. de Hoop, J. Xia, and X.S. Li. Massively parallel structured multifrontal solver for time-harmonic elastic waves in 3d anisotropic media. *Geophysics J. Int.*, 191:346–366, 2012.

[106] S. Wang, X.S. Li, F.-H. Rouet, J. Xia, and M.V. de Hoop. A parallel fast geometric multifrontal solver using hierarchically semiseparable structure. *ACM Trans. Mathematical Software*, 2013. (submitted).

[107] S. Wang, X.S. Li, J. Xia, Y. Situ, and M.V. de Hoop. Efficient parallel algorithms for solving linear systems with hierarchically semiseparable structures. *SIAM J. Scientific Computing*, 35(6):C519–C544, 2013.

[108] S. Wang, J. Xia, M.V. de Hoop, and X.S. Li. Massively parallel structured direct solver for equations describing time-harmonic qp-polarized waves in tti media. *Geophysics*, 77:T69–T82, 2012.

[109] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. *Communications of the ACM (CACM)*, April 2009.

[110] M. M. Wolf, E.G. Boman, and B. Hendrickson. Optimizing parallel sparse matrix-vector multiplication by corner partitioning. In *PARA08*, Trondheim, Norway, May 2008.

[111] J. Xia. Randomized sparse direct solvers. *SIAM J. Matrix Anal. Appl.*, 34(1):197–227, 2013.

[112] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numer. Linear Algebra Appl.*, 2010:953–976, 2010.

[113] I. Yamazaki and X.S. Li. On techniques to improve robustness and scalability of the schur complement method. In *Proc. of VECPAR 2010*, Berkeley, California, 2010.

[114] I. Yamazaki and X.S. Li. New scheduling strategies and hybrid programming for a parallel right-looking sparse LU factorization on multicore cluster systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS 2012)*, Shanghai, China, May 20-24 2012.

[115] I. Yamazaki, X.S. Li, F.-H. Rouet, and B. Uçar. Partitioning, ordering, and load balancing in a hierarchically parallel hybrid linear solver. *Int'l J. of High Performance Comput.*, 2012 (revised).

[116] I. Yamazaki, X.S. Li, F.-H. Rouet, and B. Uçar. On partitioning and reordering problems in a hierarchically parallel hybrid linear solver. In *Proc. of IPDPS 2013 Workshops*, Boston, May 20-24 2013. PDSEC-13 Workshop.

[117] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J . Alg. & Disc. Meth.*, 2:77–79, 1981.

[118] C.D. Yu, W. Wang, and D. Pierce. A CPU-GPU hybrid approach for the unsymmetric multifrontal method. *Parallel Computing*, 37:759–770, 2011.

[119] Zoltan: Parallel partitioning, load balancing and data-management services. `http://www.cs.sandia.gov/zoltan/Zoltan.html`.