Hierarchical Diagonal Blocking and Precision Reduction Applied to Combinatorial Multigrid*

Guy E. Blelloch Ioannis Koutis Gary L. Miller Kanat Tangwongsan

Computer Science Department
Carnegie Mellon University
{guyb, jkoutis, glmiller, ktangwon}@cs.cmu.edu

***Abstract*—Memory bandwidth is a major limiting factor in the scalability of parallel iterative algorithms that rely on sparse matrix-vector multiplication (SpMV). This paper introduces Hierarchical Diagonal Blocking (HDB), an approach which we believe captures many of the existing optimization techniques for SpMV in a common representation. Using this representation in conjuction with precision-reduction techniques, we develop and evaluate high-performance SpMV kernels. We also study the implications of using our SpMV kernels in a complete iterative solver. Our method of choice is a Combinatorial Multigrid solver that can fully utilize our fastest reduced-precision SpMV kernel without sacrificing the quality of the solution. We provide extensive empirical evaluation of the effectiveness of the approach on a variety of benchmark matrices, demonstrating substantial speedups on all matrices considered.**

I. Introduction

Iterative algorithms are often the method of choice for large sparse problems. For example, specialized multigrid linear solvers have been developed to solve large classes of symmetric positive definite matrices [14, 44]. Many of these solvers run in near linear time and are being applied to very large systems. These algorithms heavily rely on the sparse matrix-vector multiplication (SpMV) kernel, which dominates the running time. As noted by many, the performance of SpMV on large matrices, however, is almost always limited by memory bandwidth. This is even more pronounced on modern multicore hardware where the aggregate memory bandwidth can be particularly limiting [46] when all the cores are busy.

Many approaches have been suggested to reduce the memory bandwidth requirements in SpMV: row/column reordering [38, 37], register blocking [41], compressing row or column indices [45] , cache blocking [25, 46], symmetry [39], using single or mixed precision [16], and reorganizing the SpMV ordering across multiple iterations in a solver [35], among others. Some of these approaches are hard to parallelize. For example, the standard sparse skyline format for symmetric matrices does not parallelize well.

In this paper, we suggest an approach we refer to as *hierarchical diagonal blocking* (HDB) which we believe captures many of the existing optimization techniques in a common representation. It can take advantage of symmetry while still being easy to parallelize. It takes advantage of reordering. It also allows for simple compression of column indices. In conjunction with precision reduction (storing single-precision numbers in place of doubles), it can reduce the overall bandwidth requirements by more than a factor of three. It is particularly well-suited for the type of problems that CMG is designed for, symmetric matrices for which the corresponding graphs have reasonably small graph separators, and for which the effects of reduced precision arithmetic are well-understood. Our approach does not use register blocking although this could be added.

We prove various theoretical bounds for matrices for which the adjacency structure has edge separators of size $O(n^\alpha)$ for $\alpha < 1$. Prior work has shown a wide variety of sparse matrices have a graph structure with good separators [8]. We study the algorithm in the cache-oblivious framework [19], where algorithms are analyzed assuming a two-level memory hierarchy with an unbounded main memory and a cache of size M and line size B. As long as the algorithm does not make use of any cache parameters, the bounds are simultaneously valid across all cache levels in a hierarchical cache. For an $n \times n$ matrix with m nonzeros, we show that the number of misses is at most $m/B + O(1 + n/(Bw) + n/M^{1-\alpha})$, where w is the number of bits in a word.

We complement the theoretical results with a number of experiments, evaluating the performance of various SpMV schemes on recent multicore architectures. Our results show that a simple double-precision parallel SpMV algorithm saturates the multicore bandwidth, but by reducing the bandwidth requirements—using a combination of hierarchical diagonal blocking and precision reduction—we are able to obtain, on average, a factor of 2.5x speedup on an 8-core Nehalem machine. We also examine the implications of using the improved SpMV routine in CMG and preconditioned conjugate gradient (PCG) solvers. In addition, we explore heuristics for finding good separator-orderings and study the effects of separator quality on SpMV performance.

Reducing SpMV Bandwidth Requirements. Prior work has proposed several approaches for reducing the memory bandwidth requirements of SpMV. Reordering of rows and columns of the matrix can reduce the cache misses on the input and output vectors x and y by bringing references to these vectors closer to each other in time [37]. Many heuristic reordering approaches have been used, including graph separators such as Chaco [24] or METIS [26], Cuthill-McKee

*Partially supported by the National Science Foundation under grant number CCF-0635257 and gifts from Intel, IBM, and Microsoft.

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

reordering [20] or the Dulmage-Mendelsohn permutation [38]. These techniques tend to work well in practice since real-world matrices tend to have high locality. This is especially true with meshes derived from 2- and 3-d embeddings. Recent results have shown various bounds for meshes with good separators [6, 10, 11]. The graph structure of a wide variety of sparse matrices has been to shown to have good separators, including graphs such as the Google link graph. Reordering can be used with cache blocking [25], which blocks the matrix into sparse rectangular blocks and processes each block separately so that the same rows and columns are reused.

Index compression reduces the size of the column and row indices used to represent the matrix. The indices are normally represented as integers, but there are various ways to reduce their size. Willcok and Lumsdaine [45] apply graph compression techniques to reduce the size, showing speedups of up to 33% (although much more modest numbers on average). Williams et al. point out that by using cache blocking, it is possible to reduce the number of bits for the column indices since the number of columns in the block is typically small [46]. Register blocking [41] represents the matrix as a set of dense blocks. This can reduce the index information needed, but for very sparse or unstructured matrices, it can cause significant fill due to the insertion of zero entries to fill the dense blocks.

Data compression is a natural extension of index compression that attempts to reduce the size of the actual data contained in the matrix. For symmetric matrices, one can store the lower-triangular entries and use them twice. When stored in the sparse skyline format [39], (the compressed sparse row format with only elements strictly below the diagonal stored) a simple loop of the following form can be used:

```
// loop over rows.
for (i = 0;i < n;i++) {
  float sum = diagonal[i]*x[i];
  // loop over nonzeros below diagonal in row
  for (j = start[i];j < start[i+1];j++) {
    sum += x[cols[j]] * vals[j]; // as row
    y[cols[j]] += x[i] * vals[j]; // as column
  }
  y[i] += sum;
}
```

Fig. 1: Simple sequential code for sparse matrix vector multiply (SpMV).

Unfortunately, this loop does not parallelize well because of the unstructured addition to an element in the result vector in the statement `y[cols[j]] += x[i] * vals[j];`. Buluç et al. study how to parallelize this by recursively blocking the matrix [15], but this does not take advantage of any locality in the matrix.

Another approach to data compression is to reduce the number of bits used by the nonzero entries. Buttari et al. [16] suggest the implementation of mixed-precision inner-outer iterative algorithms, i.e. a nesting of iterative algorithms where the outer iterative method is implemented in double precision, and the inner one—formally viewed as a preconditioner to the outer one—is implemented in single precision. While often positive, the effects of reduced precision are in general unpredictable. One main advantage of the CMG solver comparing to other iterative methods is that it can be used as a preconditioner to Conjugate Gradient, and the effects of using single precision are well-understood.

Finally, recent work by Mohiyuddin et al. [35] suggests reorganizing a sequence of SpMV operations on the same matrix structure across iterations so that the same part of the vector can be reused. Although this works well when using the same matrix over multiple iterations, it does not directly help in algorithms such as multigrid, where only a single iteration on a matrix is applied before moving to another matrix of quite different form.

II. Preliminaries

Separators. Informally, a graph has $n^\alpha, \alpha < 1$ edge separators if there is a cut that partitions the graph into two almost equal sized parts such that the number of edges between the two parts is no more than n^α, within a constant. To properly deal with asymptotics and what it means to be "within a constant," separators are typically defined with respect to a infinite class of graphs. Formally, let $\mathcal{S}$ be a class of graphs that is closed under the subgraph relation (i.e., for $G \in \mathcal{S}$, every subgraph of G is also in $\mathcal{S}$). We say that $\mathcal{S}$ satisfies an *$f(n)$-edge separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that every graph $G = (V, E)$ in $\mathcal{S}$ with n vertices can be partitioned into two sets of vertices V_a, V_b such that

$$\mathsf{cutSize}(V_a, V_b) \stackrel{def}{=} |E \cap (V_a \times V_b)| \quad \leq \quad \beta f(n)$$

where $|V_a|, |V_b| \leq \alpha n$ [32]. It is well-known that bounded-degree planar graphs and graphs with bounded genus satisfy an $n^{1/2}$ edge separator theorem. It is also known that certain well-shaped meshes in d dimensions satisfy a $n^{(d-1)/d}$ edge separator theorem [34]. We note that such meshes allow for features that vary in size by large factors (e.g. small near a singularity and large where nothing is happening), but require a smooth transition from small features to large features. In addition, many other types of real-world graphs have good separators, including, for example, a link graph from Google [8].

Edge separators are often applied recursively to generate a separator tree with the vertices at the leaves and the cuts at internal nodes. Such a separator tree can be used to reorder the vertices based on an in- or post- order traversal of the tree. It is not hard to show that for graphs satisfying an n^α separator theorem, the tree can be fully balanced while maintaining the $O(n^\alpha)$ separator sizes at each node.

Separators have been used for many applications. The seminal work of Lipton and Tarjan showed how separators can be used in nested dissection to generate efficient direct solvers [32]. Another common application is to partition data structures across parallel machines to minimize communication. It has also been used to compress graphs [7] down to a linear

number of bits. The idea is that if the graph is reordered using separators, then most of the edges are 'short' and can thus be encoded using fewer bits than other edges. In this paper, we extend this to show that HDB also compresses graphs down to a linear number of bits.

Cache Oblivious Algorithms. The goal of the cache oblivious approach for analyzing algorithms is to analyze the cache cost on a simple single-level cache and then use the results to imply good performance bounds on a variety of hierarchical caches [19]. The *ideal-cache model* is used for analyzing cache costs. It is a two-level model of computation consisting of an unbounded memory and a cache of size M. Data are transferred between the two levels using cache lines of size B; all computation occurs on data in the cache. The model can run any standard computation designed for a random access machine on words of memory, and the cost is measured in terms of the number of misses incurred by the computation. This cost, often denoted by $Q(C; B, M)$, is referred to as the cache complexity for a computation C.

An algorithm is *cache oblivious* in the ideal-cache model if it does not take into account the size of M or B (or any other features of the cache). If a cache oblivious algorithm A has cache complexity $Q(A; B, M)$ on a machine with block size B and cache size M, then on a hierarchical cache with cache parameters (M_i, B_i) at level i, the algorithm will suffer at most $Q(A; M_i, B_i)$ misses at each level i. Therefore, if $Q(A; M_i, B_i)$ is asymptotically optimal for B and M, it is optimal for all levels of the cache.

Parallel Cache Oblivious Algorithms. The cache oblivious model was designed for analyzing sequential algorithms, but it has recently been extended to analyze parallel algorithms [11]. For nested parallel computations (ones with nested parallel loops and fork joins), one can analyze the algorithm using a sequential ordering and then use general results to bound cache misses on parallel machines with either shared or private hierarchical caches. In particular, for a shared-memory parallel machine with private caches (each processor has its own cache) using a work-stealing scheduler, $Q_p(A; M, B) < Q(A; M, B) + O(pMD/B)$ with probability $1 - \delta$ [3],[1] and for a shared cache using a parallel-depth-first (PDF) scheduler, $Q_p(A; M+pBD, B) \leq Q(A; M, B)$ [9], where D is the depth of the computation and p the number of processors. In a nested parallel computation, the *depth* (also known as critical path, or span) is defined inductively by taking the maximum over the depth of parallel calls and summing across sequential calls.

Therefore, the overall paradigm is to design nested parallel algorithms with reasonably low depth and for which the cache complexity is low in the ideal cache model. Controlling the depth is important as it appears in the bounds. In the context of sparse-matrix vector matrix multiply, the following has been shown for the Compressed Sparse Row (CSR) SpMV algorithm.

[1]In this paper, δ is an arbitrarily small positive constant.

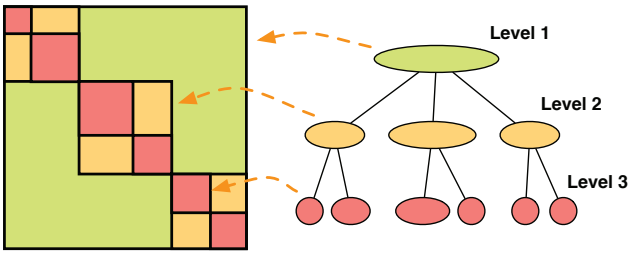

Fig. 2: Hierarchical diagonal blocking: decomposing a matrix into a tree of submatrices.

Theorem 1 (Blelloch et al. [10])**.** *Let $\mathcal{M}$ be a class of matrices for which the adjacency graphs satisfy an n^α-edge separator theorem with $\alpha < 1$. Any $n \times n$ matrix $A \in \mathcal{M}$ with $m \geq n$ nonzeros can be reordered so the CSR SpMV algorithm has $O(\log n)$ depth and $O(1 + m/B + n/M^{1-\alpha})$ sequential cache complexity.*

For $B \leq M^{1-\alpha}$ (likely in practice), the m/B term dominates so the number of cache misses is asymptotically optimal (no more than needed to scan the array entries in order).

III. Hierarchical Diagonal Blocking SpMV

In this section, we describe the *hierarchical diagonal blocking* (HDB) representation for sparse square matrices and an SpMV routine for the representation. We assume that we have already computed a fully balanced tree of edge-separators for the graph of the matrix, with the vertices as leaves. In the following discussion, we assume the rows of the square matrix are ordered by left-to-right pass over the leaves (the separator ordering), and since the matrix is square, we will use row to refer to both the row and corresponding column.

The *HDB representation* is a partitioning of the matrix into a tree of submatrices (see Figure 2). Each leaf represents a range of rows (possibly a single row), and each internal node of the tree represents a continuous range of rows it covers. Nonzero entries of the matrix are stored at the least common ancestor of the leaves containing its two indices (row and column). If both indices are in the same leaf, then the element will be stored at that leaf (all diagonal entries are at a leaf). The representation stores with each internal node the range of rows it covers, and we refer to the number of rows in the range as the node's size.

The separator tree can be used directly as the structure of the HDB tree. This, however, creates many levels which help neither in theory nor in practice. Instead, we coalesce the nodes of the separator tree so that sizes square at each level: $2, 4, 16, 256, 65536, \ldots, 2^{2^i}$. We maintain the separator ordering among the children of a node. This is important for the cache analysis. We note that for matrices with good separators most of the entries will be near the leaves.

The SpMV routine on the HDB representation works as shown in Algorithm 1. The recursive algorithm takes as arguments the input vector x, the output vector y, and a subtree/internal node T. The algorithm requires that the ordering of the x and y vectors coincide with the separator (matrix) ordering. We denote by $[\ell, u]$ the range of rows the

Algorithm 1 Sparse Matrix Vector Multiply for HDB

HDB_SpMV(x, y, T):

```
A = T.M // the nonzero entries in this node of T
[ℓ, u] = T.range
if isLeaf(T) then
    y[ℓ, u] = A · x[ℓ, u]
else
    for all t ∈ T.children, in parallel, do
        HDB_SpMV(x, y, t)
    end for
    y[ℓ, u] = y[ℓ, u] + A · x[ℓ, u]
end if
```

subtree covers. The algorithm computes the contribution to y for-all nonzero entries in the subtree. In the base case, it directly calculates the contribution. In the inductive case, each recursive call in the **for all** loop computes the contribution for the entries in its subtree. Since each of these is on a distinct range of rows, all the calls can be made in parallel without interference. After returning from the recursive calls, the algorithm adds in the contribution for the entries in the current node, hence accounting for the contribution of all entries in the subtree.

An important feature of HDB is that it gives freedom in the selection of the matrix representation A and corresponding SpMV algorithm used for each node of the tree. In particular, depending on the level, different representations can be used. If A in some node has many empty rows in its range, we need store only the non-empty rows. This can easily be done using an additional index vector of non-empty rows as is often done in cache-blocked algorithms [46]. If the matrix is symmetric, then we can keep just the lower triangular part and store it in Compressed Sparse Row (CSR) format. For a submatrix stored in this form, we can use the skyline algorithm given in Figure 1 and for internal nodes, we need not even worry about diagonals. Since the skyline algorithm is difficult to parallelize, it can be used sequentially at lower levels of the tree where there is plenty of parallelism from the recursive calls, and the CSR representation with redundant entries can be used at the higher levels. This works both in theory (proof of Theorem 2) and in practice (Section V). Another important feature of HDB is that space can be saved in storing the indices by only storing an offset relative to the beginning of the range. Again, this is used both in theory (Theorem 2) and practice (Section V).

We now bound space, cache complexity, and depth for HDB_SpMV for matrices with good separators. We assume that each nonzero value takes one word of memory. Therefore, B nonzeros fit in a cache line (this is just the values and not any indices). We assume a word has w bits in it.

Theorem 2. *Let $\mathcal{M}$ be a class of matrices for which the adjacency graphs satisfy an n^α-edge separator theorem, $\alpha < 1$, and $A \in \mathcal{M}$ be an $n \times n$ matrix with $m \geq n$ nonzeros, or $m \geq n$ lower triangular nonzeros for a symmetric matrix. If A is stored in the HDB representation T then:*

1) *T can be implemented to use $m + O(n/w)$ words.*
2) *Algorithm HDB_SpMV(x, y, T) is cache oblivious and runs with $m/B + O(1 + n/(Bw) + n/M^{1-\alpha})$ misses in the ideal cache model.*
3) *Algorithm HDB_SpMV(x, y, T) runs in $O(\log^c n)$ depth (span) for some constant c.*

Proof: We will use a modified CSR representation for all matrices stored in the tree. For symmetric matrices, we only store the lower triangular entries and diagonals for nodes of size $r < \log^{1/(1-\alpha)} n$, and all entries for larger nodes. The idea is that the number of entries in the larger matrices is small enough that we can store them twice or use a pointer to the second copy without significantly affecting space or cache complexity. As mentioned above, we modify CSR so it does not store any information for empty rows.

Consider a matrix at a node of T with size r and with e entries assigned to it. Because the range of columns is bounded by r, all column and row indices can be stored relative to the lower bound of the range using $O(\log r)$ bits. This means the matrix can be stored in ew bits for the values and $O(e \log r)$ bits for the indices. We also need to store the pointers to the children of the node. For this, we assume that the memory for nodes are allocated one after another in a postorder traversal of the tree. This means to point to a child the structure only has to point within the memory used by this subtree. This is certainly bounded by $O(wr^2)$ bits and therefore we can use a $O(\log r)$ bit pointer for each child. We can also use $O(\log r)$ bits to specify the range limits of each child, which we charge to the parent even though stored with the child. Therefore, the total space required by the node with c children is $ew + O((e + c) \log r)$. Now if we organize the tree so the nodes grow doubly exponentially $r_i = 2^{2^i}$, $(2, 4, 16, 256, 65536, \ldots)$, a node at level i captures all edges that were cut in the binary separator tree above size $2^{2^{i-1}}$ and up to 2^{2^i}. Using the separator bounds, and counting per pairwise split, we have for a node at level i, $e_i = \sum_{j=2^{i-1}+1}^{2^i} \eta(j) \times O(2^{\alpha j})$, where $\eta(j) = 2^{2^i - j}$ is the number of splits at the binary tree level j. This sum is bounded by $O(2^{2^{\alpha(i-1)}} 2^{2^{i-1}}) = O(2^{2^{\alpha(i-1)} + 2^{i-1}})$ since the terms of the sum geometrically decrease with increasing j. We also have $c_i = 2^{2^{i-1}}$ for the number of children at level i. There are n/r_i nodes at level i and therefore the total space in bits for pointers is bounded by:

$$
\begin{aligned}
S(n) &= \sum_{i=0}^{\log\log n} O\left(\frac{n}{r_i}(e_i + c_i) \log r_i\right) \\
&= \sum_{i=0}^{\log\log n} O\left(\frac{n}{2^{2^i}}(2^{(2^{\alpha(i-1)} + 2^{i-1})} + 2^{2^{i-1}})2^i\right)
\end{aligned}
$$

For $\alpha < 1$, this sum geometrically decreases, so for asymptotic analysis, we need only consider $i = 0$ and therefore $S(n) = O(n)$. When we include the space for the matrix values and convert from bits to words, the total space is $m + O(n/w)$. We note that we can store matrices with size $r \geq \log^{1/(1-\alpha)} n$ using two nonzeros per symmetric entry without affecting the asymptotic bounds. This is because there

are at most $O(n/\log n)$ nonzeros in matrices of that size so we can use a pointer of size $O(\log n)$ bits to point to the other copy, or if $w = O(\log n)$, we can store the duplicates directly.

We now consider bounds on the sequential cache complexity. The argument is similar to the argument for the CSR format [10]. We separate the misses into the accesses to the matrix entries and to the input and output vectors x and y. Recall that all tree nodes are stored in post-order with respect to the tree traversal, and at the nodes, the elements within each matrix are stored in CSR format. Since the CSR algorithm visits the matrix in the order it is stored, the algorithm visits all elements in the order they are laid out. When including the $O(n/w)$ words for indices in the structure, which are also visited in order, visiting the matrix causes a total of $m/B + O(n/(Bw))$ misses. For larger nodes in the tree where $r \geq \log^{1/(1-\alpha)} n$, we store duplicate entries, but for the same reason, this is a lower order term in the space and also a lower order term in cache misses. This leaves us to consider the number of misses from accessing x and y. For the sake of analysis, we can partition the leaves into blocks that fit into the cache, where each such block is executed in order by the algorithm. We therefore only have to consider edges that go between blocks. By the same argument as in [10], the number of such edges (entries) is bounded by $O(n/M^{1-\alpha})$ each potentially causing a miss. The total number of misses is therefore bounded by $m/B + O(1 + n/(Bw) + n/M^{1-\alpha})$.

Finally, we consider the depth of the algorithm. We assume that the SpMV for all nodes of size $r \geq \log^{1/(1-\alpha)} n$ run in parallel since they are stored with both symmetric entries. Such a SpMV runs in $O(\log n)$ depth. For $r < \log^{1/(1-\alpha)} n$, we run the SpMV on the skyline format sequentially. The total time is bounded asymptotically by the size, and all these small multiplies can run in parallel. This is the dominating term giving a total depth of $O(\log^{1/(1-\alpha)} n)$. ■

IV. Combinatorial Multigrid

To study how the improvements in SpMV performance benefit an actual iterative method, we consider Combinatorial Multigrid (CMG), a recently introduced variant of Algebraic Multigrid (AMG) [28, 29, 30] providing strong convergence guarantees for symmetric diagonally dominate linear systems [27, 40, 28, 30, 31]. Our choice is motivated by the potential for immediate impact on the design of industrial strength code for important applications. In contrast to AMG, CMG offers strong convergence guarantees for the class of symmetric diagonally dominant (SDD) matrices [23, 13, 4], and under certain conditions for the even more general class of symmetric M-matrices [17]. The convergence guarantees are based on recent progress in spectral graph theory and combinatorial preconditioning (see for example [12], [27]). At the same time, linear systems from these classes play an increasingly important role in a wave of new applications in computer vision [21, 42, 30], and medical imaging in particular [43]. Multigrid algorithms are commonly used as preconditioners to other iterative methods. The idea of implementing the preconditioner in single precision has been explored before, but the effects on convergence are in general unpredictable [16]. However, in the case of CMG, switching to single precision has provably no adverse effects. In summary, CMG can benefit from our fastest SpMV primitive, which exploits both symmetry and precision reduction, in applications that are well suited for the diagonal hierarchical blocking approach.

A thorough discussion of multigrid algorithms is out of the scope of this paper. There are many excellent survey papers and monographs on various aspects of the topic and among them [14, 44]. The purpose of this section is to discuss aspects of the parallel implementation that are specific to CMG, but at the same time, convince the reader that the performance improvements we see for CMG are expected to carry over to other flavors of multigrid.

A. CMG Description and Parallel Implementation Details

Similarly to AMG, the CMG algorithm consists of the setup phase which computes a multigrid hierarchy, and the solve phase. The CMG setup phase constructs a hierarchy of SDD matrices $A = A_0, \ldots, A_i$. As with most variants of AMG, CMG uses the *Galerkin condition* to construct the matrix A_{i+1} from A_i. This amounts to the computation of a restriction operator $R_i \in \mathbb{R}^{\dim(A_i)\times\dim(A_{i+1})}$, and the construction of A_{i+1} via the relation $A_{i+1} = R_i^T A_i R_i$. CMG constructs the restriction operator R_i by grouping the variables/nodes of A_i into $\dim(A_{i+1})$ disjoint clusters and letting $R(i,j) = 1$ if node i is in cluster j, and $R(i,j) = 0$ otherwise. This simple approach is known as *aggregate-based* coarsening, and it has recently attracted significant interest due to its simplicity and advantages for parallel implementations [22, 36]. Classic AMG constructs more complicated restriction operators that can be viewed as (partially) overlapping clusters. The main difference between CMG and other AMG variants is the algorithm for clustering, which in the CMG case is combinatorially rather than algebraically driven. The running time of the CMG setup phase is negligible comparing to the actual MG iteration, so we do not further discuss it in this paper. The reader can find more details in [30].

Algorithm 2 The CMG Solve Phase

function $x_i = \mathsf{CMG}(A_i, b_i)$

1: $D = \mathrm{diag}(A)$
2: $r_i = b_i - A_i(D^{-1}b)$
3: $b_{i+1} = Rr_i$
4: $z = \mathsf{CMG}(A_{i+1}, b_{i+1})$
5: **for** $i = 1$ **to** $t_i - 1$ **do**
6: $\quad r_{i+1} = b_{i+1} - A_{i+1}z$
7: $\quad z = z + \mathsf{CMG}(A_{i+1}, r_{i+1})$
8: **end for**
9: $x = R^T z$
10: $x = r_i - D^{-1}(A_i x - b)$

The solve phase of CMG, which is dominated by SpMV operations, is quite similar to the AMG solve phase; the pseudocode is given in Figure 2. When $t_i = 1$, the algorithm is known

in the MG literature as the V-cycle, while when $t_i = 2$, it is known as the W-cycle. It has been known that the aggregate-based AMG does not exhibit good convergence for the V-cycle. The theory in [27] essentially proves that more complicated cycles are expected to converge fast, without blowing up the total work performed by the algorithm. This is validated by our experiments with CMG where we pick

$$t_i = \max\left\{\left\lceil \frac{nnz(A_i)}{nnz(A_{i+1})} - 1\right\rceil, 1\right\}.$$

Here $nnz(A)$ denotes the number of nonzero entries of A. This choice for the number of recursive calls, combined with the fast geometric decrease of the matrix sizes, targets a geometric decrease in the total work per level.

In our parallel implementation, we optimized the CMG solve phase by using different SpMV implementations for different matrix sizes. When the matrix size is larger than 1K, we use the blocked version of SpMV, and when it is smaller than that, we resort to the plain parallel implementation, where the matrix is stored in full and we compute each row in parallel. The reason is that the blocked version of SpMV has higher overhead than the simple implementation for smaller matrices.

In our experiments, we found that a choice of $t'_i = t_i + 1$ improves (in some examples) the sequential running time required for convergence by as much as 5%. However, it redistributes work to lower levels of the hierarchy where, as noted above, the SpMV speedups are smaller. As a result, the overall performance gains for CMG are less significant with this choice.

B. Single vs. Double Precision CMG

The CMG solve phase is the implicit inverse of a symmetric positive operator B. The condition number $\kappa(A, B)$ can therefore be defined, and it is well-understood that it characterizes the rate of convergence of the preconditioned CG iteration [5].

Recall that the CMG core works with the assumption that the system matrix A is SDD. We form a single precision matrix $\hat{A}$ from the double precision matrix A as follows; we decompose A into $A = D + L$, where L has zero (in double precision) row sums and D is a diagonal matrix with non-negative entries. We form $\tilde{D}$ by casting the positive entries of D into single precision. We form $\tilde{L}$ by casting the off-diagonal entries of L into single precision, adding them in the order they appear using single precision, and then negating the sum and setting it to the corresponding diagonal entry of $\tilde{L}$. Finally, we let $\tilde{A} = \tilde{D} + \tilde{L}$. This construction guarantees that $\tilde{A}$ is numerically diagonally dominant and thus positive definite.

Substituting a double-precision hierarchy $A_0, \ldots, A_d$ by its single-precision counterpart $\tilde{A}_0, \ldots, \tilde{A}_d$ in effect changes the symmetric operator B to a new operator $\hat{B}$, which is also symmetric. By an inductive (on the number of levels) argument, it can be shown that

$$\kappa(B, \tilde{B}) \leq \max_i \kappa(A_i, \tilde{A}_i).$$

Using the Splitting Lemma for condition numbers [12], it is easy to show that

$$\kappa(A, \tilde{A}) \leq \left(\max_i \left\{\frac{D_{i,i}}{\tilde{D}_{i,i}}, \frac{\tilde{D}_{i,i}}{D_{i,i}}, \max_{j\neq i}\left\{\frac{|L_{i,j}|}{|\tilde{L}_{i,j}|}, \frac{|\tilde{L}_{i,j}|}{|L_{i,j}|}\right\}\right\}\right)^2.$$

Under reasonable assumptions for the range of numbers used in A, we get $\kappa(B, \tilde{B}) < 1 + 10^{-7}$. Using the transitivity of condition numbers, we get

$$\kappa(A, \tilde{B}) \leq \kappa(A, B)\kappa(B, \tilde{B}) \leq \kappa(A, B)(1 + 10^{-7}).$$

It is known that the condition number of a pair (A, B) is the ratio of the largest to the smallest generalized eigenvalue of (A, B). The above inequality can in fact be extended to show that *each* generalized eigenvalue of the pair (A, B) is within a $(1 + 10^{-7})$ factor of the corresponding generalized eigenvalue of $(A, \tilde{B})$. Thus, the preconditioned CG is expected to have an almost identical convergence, independent of whether B or $\tilde{B}$ is the preconditioner.

V. Implementation and Evaluation

This section describes an implementation of an SpMV based on hierarchical diagonal blocking and a study of its performance compared to other related variants.

A. Implementation of SpMV

We implemented SpMV routines for symmetric matrices using the descriptions from Section III. The implementation stores a matrix as groups of on-diagonal entries, diagonal-block entries, and off-block entries (similar to Figure 2 with only 2 inner-node levels and a level of leaf nodes). The diagonal blocks in the first level are $\sim$ 32K in size (to take advantage of caching) and the leaves correspond the singletons along the matrix's diagonal. This representation allows for a simple implementation which delivers good performance in practice.

The two main ideas from previous sections are precision reduction and diagonal blocking. To understand the benefits of these ideas individually, we study the following variants: the sequential program using double-precision numbers "seq. (double)" is our baseline implementation (more details below). The simple parallel program for double-precision numbers "simple par. (double)" computes the rows in parallel. The corresponding version for single-precision numbers is known as "simple par. (single)." We have two variants of the hierarchical diagonal blocking routines, one for double-precision numbers "blocked par. (double)" and one for single-precision numbers "blocked par. (single)". *The names inside quotation marks are abbreviated names used in all the figures.*

The baseline implementation is a simple sequential program similar to what is shown in Figure 1. We optimized the code slightly by applying one level of loop-unrolling to the inner loop. Note that although the code is simple, its performance matches, within 1%, that of highly optimized kernels for SpMV, such as Intel Math Kernel Library [2]. We decided to work with our own implementation because of the flexibility in changing and instrumenting the code (e.g., for collecting statistics).

All versions of our parallel programs were written in Cilk++, a language similar to C++ with keywords that allow users to specify what should be run in parallel [1]. Cilk++'s runtime system relies on a work-stealing scheduler, a dynamic scheduler that allows tasks to be rescheduled dynamically at low overhead cost. Our benchmark programs were compiled with Intel Cilk++ build 8503 using the optimization flag `-O2`. To avoid overhead in the Cilk++'s runtime system, we compiled the baseline sequential programs with GNU `g++` version 4.4.1 using the optimization flag `-O2`.[2]

B. Experimental Setup

Testbed. We are interested in understanding the performance characteristics of SpMV and CMG solvers on three recent machine architectures: the Nehalem-based Xeon, the Intel Harpertown, and the AMD Opteron Shanghai. A brief summary of our test machines is presented in Table I. Our measurements were taken with hyperthreading turned off. Even though hyperthreading gives a slight boost in performance (though less than 5%), the timing numbers were much more reliable with it turned off.

Machine Model	Speed	Layout	Agg. Bandwidth	
	(Ghz)	(#chips×#cores)	1 core	8 cores
Intel Nehalem X5550	2.66	2 × 4	10.5	27.9
Intel Harpertown E5440	2.83	2 × 4	2.8	6.4
AMD Shanghai 2384	2.70	2 × 4	4.9	10.7

TABLE I: Characteristics of the architectures used in our study, where clock speeds are reported in Ghz and 1- and 8-core aggregate bandwidth numbers in GBytes/sec. For the aggregate bandwidth, we report the performance of the *triad* test in the University of Virginia's STREAM benchmark [33], compiled with `gcc -O2` and using `gcc`'s OpenMP.

Among these architectures, the Intel Nehalem is the current flagship, which shows significant improvements in bandwidth over prior architectures. For this reason, this work focuses on our performance on the Nehalem machine; we include results for other architectures for comparisons as our techniques benefit other architectures as well.

Datasets. Our study involves a diverse collection of large sparse matrices, gathered from the University of Florida Matrix Collection [18] and a collection of mesh matrices generated by applications in vision and medical imaging. We present a summary of these matrices in Table II. These matrices are chosen so that for the majority of them, neither the vectors nor the whole matrix can fit entirely in cache; smaller matrices are also included for comparison.

For the CMG experiments, since the CMG solver requires the input matrix to be SDD, we replace each off-diagonal entry with a negative number of the same magnitude, and we adjust the diagonals to get zero row-sums. The perturbation does not affect the SpMV performance, as the matrix structure remains unchanged, but it allows us to study the performance of CMG on various sparse patterns.

[2]We have also experimented with the Intel compiler and found similar results.

Matrix	#rows/cols	#nonzero
2d-A	999,999	4,995,995
3d-A	999,999	6,939,993
af_shell10	1,508,065	52,672,325
audikw_1	943,695	77,651,847
bone010	986,703	71,666,325
ecology2	999,999	4,995,991
nd24k	72,000	28,715,634
nlpkkt120	3,542,400	96,845,792
pwtk	217,918	11,634,424

TABLE II: Summary of matrices used in the experiments.

All matrices in the study are ordered in the best possible ordering we are able to find. Each matrix is reordered using a number of heuristics and we keep the ordering that yields the best baseline performance. For each matrix, we use the same ordering when comparing SpMV schemes. We discuss the effects of separator quality in Section V-E.

C. Performance of SpMV

The first set of experiments concerns the performance of SpMV. In these experiments, we are especially interested in understanding how the ideas outlined in previous sections perform on a variety of sparse matrices.

Throughput. Figure 3 and Table III show the performance (in GFlops) and the speedup achieved by various SpMV routines on the matrices in our collection. Several things are clear. First, on all these matrices, a simple parallel algorithm speeds up SpMV by 3.4x–4.5x. In fact, without any data reduction, we cannot hope to improve the performance much further, because as will be apparent in the next discussion, the simple parallel algorithm operates near the peak bandwidth.

Matrix	Speedup simple par. (double)	Speedup blocked par. (single)
2d-A	3.9x	7.1x
3d-A	3.7x	7.6x
af_shell10	4.3x	11.3x
audikw_1	4.0x	11.0x
bone010	3.7x	9.7x
ecology2	3.4x	6.2x
nd24k	3.9x	9.6x
nlpkkt120	3.8x	8.4x
pwtk	3.7x	10.1x
thermal2	4.5x	7.3x

TABLE III: Speedup numbers of parallel SpMV on an 8-core Nehalem machine as compared to the sequential baseline code.

Second, but more importantly, both hierarchical diagonal blocking and precision reduction can help enhance the speed of SpMV, but *neither idea alone yields as much performance improvement as their combination.* By replacing double-precision numbers with single-precision numbers, we use 4 bytes per matrix entry instead of 8. Furthermore, by using the hierarchical diagonal blocking with the top-level block size $\sim$ 32K, we can represent the indices of the entries in the diagonal blocks using 16-bit words, a saving from 32-bit

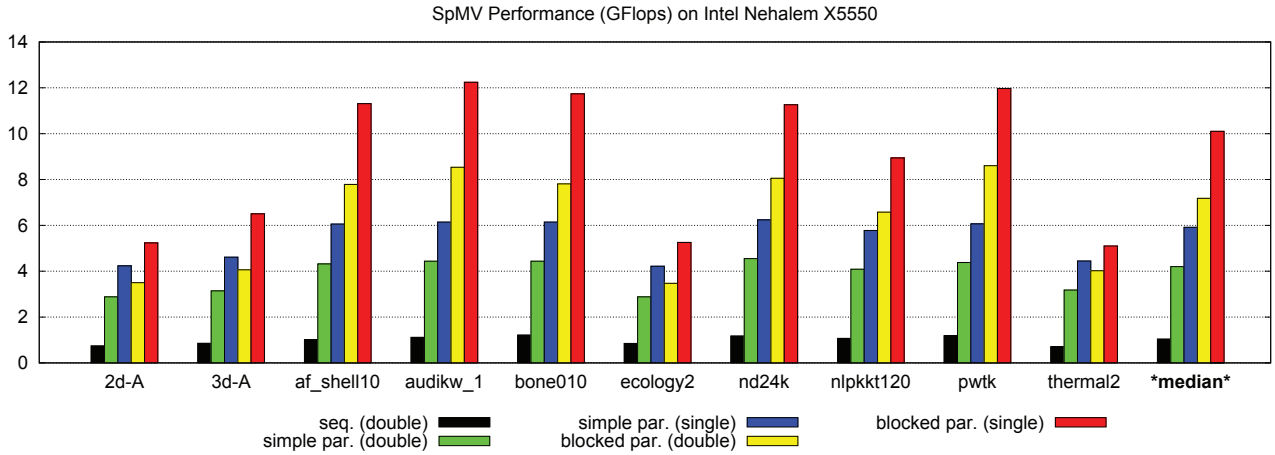

Fig. 3: Performance of different SpMV routines (**in GFlops**) on a variety of matrices.

words used to represent matrix indices in a normal CSR format. Diagonal blocking can also take advantage of symmetry: each digonal blocks can be stored in the skyline format, which halves the number of entries (both indicies and values) we have to store. Combining these ideas, we not only further reduce the bandwidth but also improve the cache locality due to blocking. Shown in Table IV is the memory footprint of the different representations. By applying the blocking on these matrices, the footprint can be reduced by more than 1.5x and can be further reduced by precision reduction. This is reflected in the additional speedup of more than 2x in the speedup of the single-precision blocked parallel version over the speedup of the simple double-precision parallel code.

Matrix	Memory Access (MBytes)		
	CSR/double	blocked/double	blocked/single
2d-A	80	56	36
3d-A	103	67	43
af_shell10	657	313	193
audikw_1	951	426	261
bone010	880	404	251
ecology2	80	56	36
nd24k	346	164	106
nlpkkt120	1212	589	367
pwtk	143	65	40
thermal2	128	85	55

TABLE IV: Total memory accesses (**in MBytes**) to perform one SpMV operation using different representations.

Scalability. Presented in Figures 4 and 5 are speedup and bandwidth numbers for different SpMV routines. The speedup on i cores is how much faster a program is on i cores than on 1 core running the same program. First and most importantly, *blocked parallel single precision scales the best on all three machines.* On the Nehalem, it achieves a factor of almost 7x compared to approximately 4x for the simple double-precision parallel SpMV. Furthermore, the trend is similar between Nehalem and Shanghai, which both have more memory channels and higher bandwidth than the Harpertown. On the Harpertown, all the benchmarks saturate at 4 cores, potentially due to the limited bandwidth.

Second, reducing the memory footprint (hence the bandwidth requirement) is key to improving the scalability. As Figure 5 shows, the simple parallel SpMV seems to be compute bound on 1 core but runs near peak bandwidth on 8 cores, suggesting that further performance improvement is unlikely without reducing the bandwidth requirements. But, as noted earlier, the blocked schemes have substantially smaller memory footprint than the simple scheme. For this reason, the blocked schemes are able to achieve better FLOPS counts and scalability even though they do not operate near the peak bandwidth.

D. Performance and Convergence of CMG

Figure 6 shows the performance of *one call* to three CMG programs, differing in the SpMV kernel used. The precision of scalars and vectors used by CMG match that of its SpMV kernel. In the parallel implementations, vector-vector operations in the CMG programs are also parallelized, when possible, in a straightforward manner.

From the figure, two things are clear. First, the speedup—the ratio between the performance of the baseline sequential program and the parallel one—varies with the linear system being solved; however, on all datasets we consider here, the speedup is more than 3x, with the best case reaching beyond 6x. Second, the speedup of the CMG solver seems to be proportional to the speedup of SpMV, but not as good. This finding is consistent with the fact that the largest fraction of the work is spent in SpMV, while part of the work is spent on operations with more modest speedups (e.g., vector-vector operators and SpMV operations on smaller matrices).

The CMG is used as a preconditioner in a Preconditioned Conjugate Gradients (PCG) iteration. In Table V, we report the number of PCG iterations required to compute a solution x such that the relative residual error satisfies $\|Ax - b\|/\|b\| < 10^{-8}$, for various matrices and three different b-sides. The first column corresponds to a random vector b, the second to Ab and the third to an approximate solution of $Ax = b$, for the same random b. We note that the reported convergence rates are preliminary. Improvements may be possible as long as the hierarchy construction abides by the sufficient and necessary

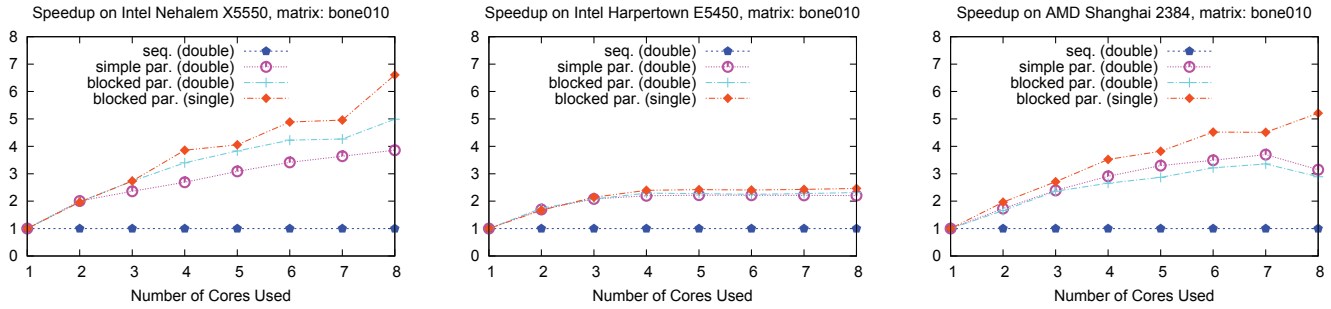

Fig. 4: Speedup factors of SpMV on Intel Nehalem X5550, AMD Shanghai 2384, and Intel Harpertown E5440 as the number of cores used is varied.

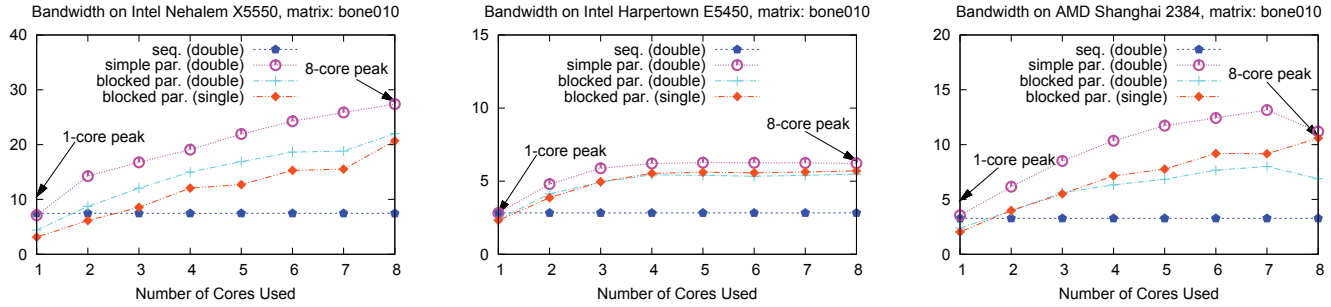

Fig. 5: Bandwidth consumption of SpMV (**in GBytes/sec**) on Intel Nehalem X5550, Intel Harpertown E5440, and AMD Shanghai 2384 as the number of cores used is varied. The peak 1- and 8-core bandwidth numbers are from Table I.

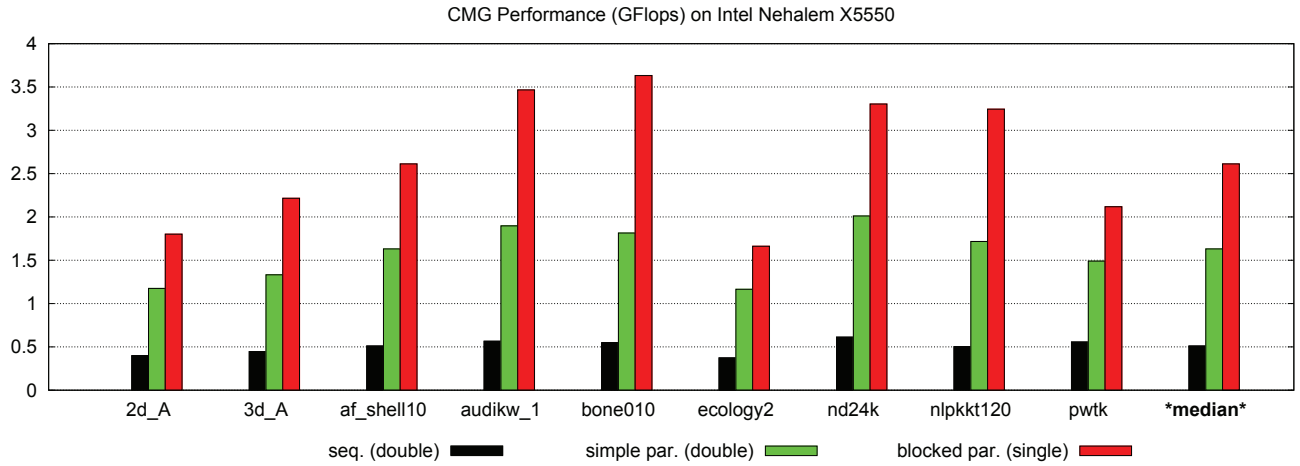

Fig. 6: Performance of a CMG solve iteration (**in GFlops**) on different linear systems.

conditions reported in [29]. One call to CMG is on average 5–6 times slower than one call to SpMV. Most of the matrices have a particularly bad condition number and standard CG without preconditioning would require thousands of iterations to achieve the same residual error.

As predicted by the theory in Section IV-B, CG preconditioned with double-precision CMG is virtually indistinguishable from CG preconditioned with single-precision CMG; the number of iterations for convergence differs by at most 1 in all our experiments. We have also found that further improvements can be found by using a single-precision implementation of CG to drive the error down to 10^{-6} and then switching to the double-precision mode. In Table V, we report the running times of *one call* to PCG, with CG implemented in single

Matrix	#iterations			PCG run time per call	
	random b	Ab	A^+b	P-single-CG	P-double-CG
2d-A	42	34	48	24.15	31.1
3d-A	37	32	37	24.3	31.5
af_shell10	26	23	30	195.3	231.3
audikw_1	19	15	17	205.0	245.8
ecology2	49	37	55	25.5	32.2
nlpkkt120	26	20	28	203.2	256.8

TABLE V: PCG: number of iterations required for convergence of error to 10^{-8} and running time per call **in milliseconds**.

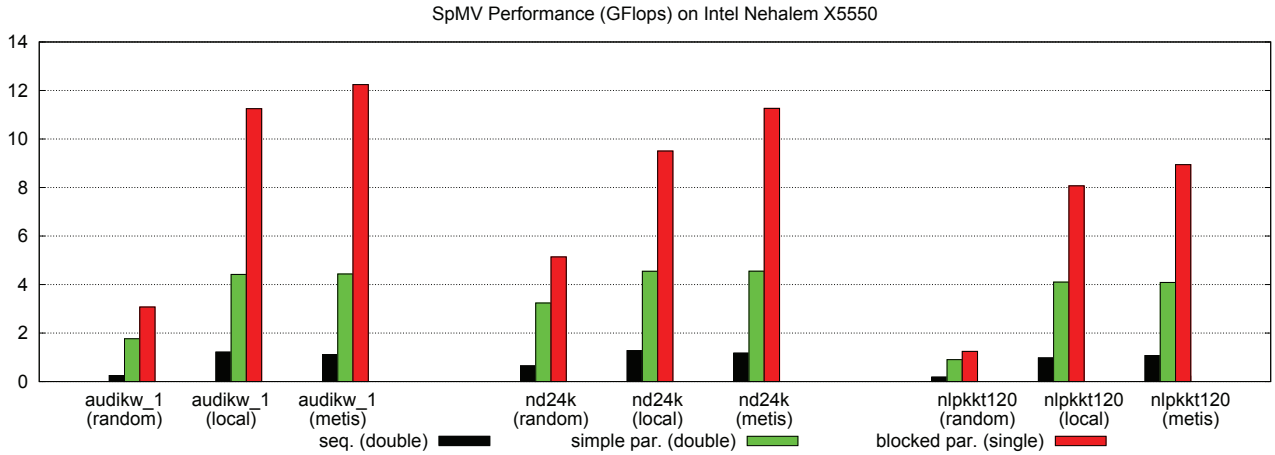

Fig. 7: Performance of SpMV routines (**in GFlops**) with different ordering heuristics.

precision and double precision—the preconditioner CMG is implemented consistently in single precision.

E. Effects of Separator Quality

Our results thus far rely on the assumption that the input matrices are given in a good separator-ordering. Often, however, the matrices have good separators but are not prearranged in such an ordering. In this section, we explore various heuristics for computing a good separator-ordering and compare their relative performance with respect to SpMV.

We begin by defining two abstract measures of the quality of an ordering. The first measure, called the ℓ-*distance*, is inspired by previous work on graph compression using separator trees [7]. The ℓ-distance is an information-theoretic lower bound on the average number of bits needed to represent the index of an entry. This measure therefore indicates how well the ordering compresses. Formally, for a matrix M,

$$\ell(M) := \frac{1}{\#nnz} \sum_{(i,j)\in M} \log_2 |i-j+1|.$$

Simpler than the first, the second measure—denoted by "off"—is simply the percentage of the nonzero entries that fall off the first-level blocks. This measure tells us what fraction of the nonzero elements has to resort to the simple parallel scheme and cannot benefit from the blocks.

As we already discussed in Section III, at the heart of a separator ordering is a separator tree—a fully balanced tree of edge-separators for the graph of the matrix. For the study, we consider the following graph-partitioning and reordering heuristics: (1) "local," a bottom-up contraction heuristic (known in the original paper as bu) [7]; (2) METIS, an algorithm which recursively applies the METIS partitioning algorithm [26]; and (3) a random ordering of the vertices.

Table VI shows statistics for these heuristics on three of the matrices used in previous sections. On both the ℓ-distance and off-block measures, it is clear that METIS produces superior orderings than the local heuristic does on all of the matrices considered; however, the local heuristic is significantly faster than METIS, both running sequentially—and as we will see next, both schemes yield comparable SpMV performance. In terms of parallelization potential, we were unable to run parMETIS on our Nehalem machine. Yet, the local heuristic shows good speedup running on 8 cores, finishing in under 3 seconds on the largest matrix with almost 100 million entries and exhibiting more than 5x speedup over 1 core.

Matrix	nnz/row	Random		Local				METIS		
	(avg.)	ℓ	off	ℓ	off	T_1	T_8	ℓ	off	T_1
audikw_1	82.3	17.5	92.6%	7.6	9.0%	11.1	1.9	6.8	3.6%	76.1
nd24k	399.0	13.9	93.5%	9.5	36.1%	5.0	0.8	8.5	21.4%	12.0
nlpkkt120	26.9	19.2	96.6%	7.5	11.9%	15.3	2.6	6.3	5.3%	230.5

TABLE VI: Statistics about different ordering heuristics: ℓ is the ℓ-distance defined in Section V-E and off is the percentage of the entries that fall off the diagonal blocks. The timing numbers (**in seconds**, T_1 for the sequential code and T_8 for the parallel code on 8 cores) on the Nehalem are reported.

We show in Figure 7 how the different ordering heuristics compare in terms of SpMV performance. First but unsurprisingly, the random ordering, which we expect to have almost no locality, performs the worst on all three SpMV algorithms. Second, as can be seen from the stark difference between the random ordering and the other two schemes, a good separator-ordering benefits *all* algorithms, not just the HBD scheme. Third but most importantly, the SpMV algorithms are "robust" against small differences in the separator's quality: on all algorithms, there is no significant performance loss when switching from METIS to a slightly worse, but faster to compute, ordering produced by the local heuristic.

VI. Conclusions

This paper described a sparse matrix representation which in conjunction with precision reduction, forms the basis for high-performance SpMV kernels. We evaluated their performance both as stand-alone kernels and on CMG, showing substantial speedsup on a diverse collection of matrices.

References

[1] Intel + Cilk , 2010. URL http://software.intel.com/en-us/blogs/2009/07/31/cilk-intel/.

[2] Intel Math Kernel Library , 2010. URL http://software.intel.com/en-us/intel-mkl/.

[3] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3), 2002.

[4] H. Avron, D. Chen, G. Shklarski, and S. Toledo. Combinatorial preconditioners for scalar elliptic finite-element problems. *SIAM J. Matrix Anal. Appl.*, 31(2):694–720, 2009.

[5] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, New York, NY, 1994.

[6] M. A. Bender, B. C. Kuszmaul, S.-H. Teng, and K. Wang. Optimal cache-oblivious mesh layout. Computing Research Repository (CoRR) abs/0705.1033, 2007.

[7] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *SODA*, pages 679–688, 2003.

[8] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis of a compact graph representation. In *ALENEX/ANALC*, pages 49–61, 2004.

[9] G. E. Blelloch and P. B. Gibbons. Effectively sharing a cache among threads. In *ACM SPAA '04*, 2004. doi: http://doi.acm.org/10.1145/1007912.1007948.

[10] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA*, pages 501–510, 2008.

[11] G. E. Blelloch, P. B. Gibbons, and H. V. Simhadri. Low-depth cache oblivious algorithms. In *SPAA*, 2010.

[12] E. G. Boman and B. Hendrickson. Support theory for preconditioning. *SIAM J. Matrix Anal. Appl.*, 25(3):694–717, 2003.

[13] E. G. Boman, B. Hendrickson, and S. A. Vavasis. Solving elliptic finite element systems in near-linear time with support preconditioners. *CoRR*, cs.NA/0407022, 2004.

[14] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial: second edition*. Society for Industrial and Applied Mathematics, 2000.

[15] A. Buluc, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *SPAA*, pages 233–244, 2009.

[16] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34(4):1–22, 2008. ISSN 0098-3500.

[17] S. I. Daitch and D. A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 451–460, 2008.

[18] T. A. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 92, 1994. http://www.cise.ufl.edu/research/sparse/matrices/.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298, 1999.

[20] A. George and J. W. Liu. *Computer Solutions of Large SParse Positive Definite Systems*. Prentice-Hall, 1981.

[21] L. Grady. Random walks for image segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 2(11):1768–1783, 2006.

[22] L. Grady. A lattice-preserving multigrid method for solving the inhomogeneous poisson equations used in image analysis. In D. A. Forsyth, P. H. S. Torr, and A. Zisserman, editors, *ECCV (2)*, volume 5303 of *Lecture Notes in Computer Science*, pages 252–264. Springer, 2008.

[23] K. Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonally Dominant Linear Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, October 1996. CMU CS Tech Report CMU-CS-96-123.

[24] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing '95: Proceedings ACM/IEEE conference on Supercomputing*, page 28, 1995.

[25] E.-J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[26] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[27] I. Koutis. *Combinatorial and algebraic algorithms for optimal multilevel algorithms*. PhD thesis, Carnegie Mellon University, Pittsburgh, May 2007.

[28] I. Koutis and G. Miller. The Combinatorial Multigrid Solver. In *The 14th Copper Mountain Conference on Multigrid Methods*, March 2009. Conference Talk.

[29] I. Koutis and G. L. Miller. Graph partitioning into isolated, high conductance clusters: Theory, computation and applications to preconditioning. In *Symposiun on Parallel Algorithms and Architectures (SPAA)*, 2008.

[30] I. Koutis, G. L. Miller, and D. Tolliver. Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing. In *International Symposium of Visual Computing*, pages 1067–1078, 2009.

[31] I. Koutis, G. L. Miller, and R. Peng. Approaching optimality for solving SDD systems. In *FOCS*, 2010.

[32] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979. doi: 10.1137/0136016.

[33] J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. URL http://www.cs.virginia.edu/stream/. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[34] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *FOCS*, pages 538–547, 1991.

[35] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. A. Yelick. Minimizing communication in sparse matrix solvers. In *SC*. ACM, 2009.

[36] A. C. Muresan and Y. Notay. Analysis of aggregation-based multigrid. *SIAM J. Scientific Computing*, 30(2):1082–1103, 2008.

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

[38] A. Pothen and C.-J. Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw.*, 16(4):303–324, 1990.

[39] Y. Saad. Sparskit: A basic took kit for sparse matrix computations. Technical Report 90-20, RIACS, NASA Ames Research Center, 1990.

[40] D. A. Spielman and S.-H. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear

systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*, pages 81–90, June 2004.

[41] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *IBM Journal of Research and Development*, 1997.

[42] D. Tolliver and G. L. Miller. Graph partitioning by spectral rounding: Applications in image segmentation and clustering. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2006)*, pages 1053–1060, 2006.

[43] D. A. Tolliver, I. Koutis, H. Ishikawa, J. S. Schuman, and G. L. Miller. Automatic multiple retinal layer segmentation in spectral domain oct scans via spectral rounding. In *ARVO Annual Meeting*, May 2008.

[44] U. Trottenberg, A. Schuller, and C. Oosterlee. *Multigrid*. Academic Press, 1st edition, 2000.

[45] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *ICS*, pages 307–316, 2006.

[46] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, 2007.