Reducing Multicore Bandwidth Requirements for Combinatorial Multigrid

Guy E. Blelloch Ioannis Koutis Gary L. Miller Kanat Tangwongsan
Carnegie Mellon University
{guyb, jkoutis, glmiller, ktangwon}@cs.cmu.edu

***Abstract*—Memory bandwidth is a major limiting factor in the scalability of parallel algorithms. In this paper, we introduce *hierarchical diagonal blocking*, a sparse matrix representation which we believe captures most of optimization techniques in a common representation. It can take advantage of symmetry while still being easy to parallelize. It takes advantage of, or actually requires, reordering. It also allows for simple compression of column indices. As applications, we show how to use this high-performance SpMV kernel, together with precision reduction techniques, in a combinatorial multigrid solver to lower the bandwidth consumption *without* sacrificing the final solution's quality. 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

Specialized multigrid solvers have been developed to solve large classes of symmetric positive definite matrices [14, 46]. In many cases these solvers run in nearly linear time, and they are being applied to very large systems. Multigrid solvers are quite often used as preconditioners in preconditioned iterative methods that enhance their performance as stand-alone solvers. The dominant work performed by these algorithms is sparse matrix vector multiply (SpMV) where the SpMV is applied on a sequence of matrices of different sizes and structure. The dominant cost is typically on the larger matrices. As noted by many, the performance of SpMV on large matrices is almost always limited by memory bandwidth. This is even more so on modern multicore hardware where the aggregate memory bandwidth when all the cores are busy can be particularly limiting [48]. We show how to achieve significant speed-ups on SpMV, and we study the impact on multigrid iterations. Our test vehicle is the Combinatorial Multigrid Solver, a variant of multigrid which provides strong convergence guarantees for symmetric diagonally dominate linear systems, based on recent theoretical progress [28, 42, 29, 31, 32].

Many approaches have been suggested to reduce the memory bandwidth requirements in SpMV, based on row/column reordering [40, 39], register blocking [43], cache blocking [26, 48], symmetry [41], using single or mixed precision [16], and compressing row or column indices [47], and reorganizing ordering across multiple iterations in a solver [37], among others. Some of these approaches are hard to parallelize and other make it easier. For example the standard approach for representing symmetric matrices with the sparse skyline format does not parallelize well and parallelizing while taking advantage of symmetry is complicated.

In this paper we suggest an approach we refer to as *hierarchical diagonal blocking* (HDB) which we believe captures most of optimization techniques in a common representation. It can take advantage of symmetry while still being easy to parallelize. It takes advantage of, or actually requires, reordering. It also allows for simple compression of column indices. In conjunction with reducing the precision of matrix values from double to single this can half the overall bandwidth requirements. It is particularly well suited for the sort of problems that appear in multigrid solvers—symmetric matrices for which the corresponding graphs have reasonably small graph separators, and for which single precision arithmetic is sufficient. Our approach does not use register blocking, although this could be added.

We prove various theoretical bounds on the approach for matrices for which the adjacency structure has edge separators of size $O(n^\alpha), \alpha < 1$. It has been shown that the graph structure of wide variety of sparse matrices have good separators [8]. For caching we consider the cache-oblivious model [19]. In this model algorithms are analyzed assuming a two level memory hierarchy with an unbounded main memory and a cache of size M and line-size B. However as long as the algorithm does not take advantage 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 bound the number of misses in the the cache oblivious model to $m/B + O(1 + n/(Bw) + n/M^\alpha)$ where w is the number of bits in a word.

We also study the approach experimentally. We look both at times of SpMV and also as used in the context of a combinatorial multigrid solver. We show that by reducing bandwidth we not only significantly speed up, but are able to scale much better on multiple cores where the bandwidth become more limiting. On an 8-core Nehalem machine, we are able to achieve 12Gflops, which is better than previous reported numbers we know about on a fully sparse representation (one in which no register blocking is used).

Approaches to reducing main-memory bandwidth. Many approaches have been suggested for reducing the memory bandwidth of SpMV. One recent approach is to reorganize a

Partially supported by the National Science Foundation under grant number CCF-0635257.

sequence of SpMV operations on the same matrix structure across iterations [37] 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. Other approaches are based on register blocking [43], which represents the matrix as a set of dense blocks. This can reduce the index information needed, but for very sparse or unstructured matrices can cause significant fill due to inserting zero entries to fill the dense blocks.

Another method is to reorder the rows and columns of the matrix to help reduce the cache misses on the input and output vector x and y by bringing references to these vectors closer to each other in time [39]. Heuristically many reordering approaches have been used including graph separators such as Chaco [24] or Metis [27], Cuthill-McKee reordering [20] or the Dulmage-Mendelsohn permutation [40]. These techniques tend to work well in practice since most real-world matrices have a lot of locality. This is especially true with meshes that come from embeddings in 2 and 3 dimensions. Recent results have proved various bounds for meshes with good separators [6, 10, 11]. It has been shown that the graph structure of wide variety of sparse matrices have good separators, including graphs such as the Google link graph. Such reorderings can be used in conjunction with cache blocking [26] which blocks the matrix into sparse rectangular blocks and processes each block separately so that the same rows and columns are reused.

For symmetric matrices once can store the lower triangular entries and use them twice. When stored in sparse skyline format [41] (the compressed sparse row format with only elements strictly below the diagonal stored) a simple loop of the following form can then 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. studied how to parallelize this by recursively blocking the matrix [15]. This however does not take advantage of any locality in the matrix.

Another approach to reducing bandwidth is to reduce the number of bits used by the nonzero entries. Buttari et al. [16] study how reducing from double precision to single precision affects results and in particular suggest using mixed precision, where single precision is used for most of the computation and double for certain critical parts. Yet another method of reducing bandwidth requirements is to compress the column or row numbers. These would normally be represented as integers, but there are various ways to reduce the size of the indices. Willcok and Lumsdaine [47] use graph compression techniques to reduce the size. They show speed ups of up to 33%, although much more modest numbers of average. William's et al. point out that by using cache blocking it is easy to reduce the number of bits for the column indices since the number of columns in the block is typically small [48].

Background on Solvers. Absent any special properties of the input system $Ax = b$, *polynomial acceleration*-based methods, like CG and GMRES, are often the method of choice.

Polynomial acceleration methods are typically based on the computation of kernels $[x_0, x_1, \ldots, x_k]$, where x_k is formed by taking a linear combination of $x_0, \ldots, x_{k-1}$ and multiplying by A. A "communication avoiding" CA-GMRES method described in [37] demonstrated that significant speed-ups are possible over plain GMRES. The algorithm exploits the fact that an entry of x_j depends only on a *small subset* of the entries of x_0 and of A, when the graph of A exhibits (as often is the case) mostly 'local' connectivity; this is used to avoid fetching the whole matrix A for the computation of each x_j, thus reducing the cost of communication.

The speed of the SpMV core of an iterative solver is obviously an important factor in its performance, but the crucial factor is its *convergence rate*, captured by the number of iterations required for the computation of a satisfactory solution. The convergence rate is typically poor for matrices whose graph has large diameter, or more generally low connectivity. The intuition behind this phenomenon is that information does not travel fast within the graph via the SpMV operations and even after a large number of iterations the approximate solution is a function of only a part of the first iterate, when every entry of $A^{-1}b$ is obviously a function of every entry of b.

A very common approach to deal with the problem of slow convergence, is the transformation of the system $Ax = b$ into a *preconditioned* system $B^{-1}Ax = B^{-1}b$, where B is a linear operator, the *preconditioner*, which can be thought of as an approximate version of A which is somehow easier to solve. Any good preconditioner has to be designed in a way that addresses the low connectivity of A. In effect, that means that the new matrix $B^{-1}A$ should be highly connected, rendering useless the approach of [37], provided of course that a good preconditioner is indeed available.

Multigrid (MG) solvers, is a widely studied and successful in practice class of iterative linear system solvers. There are literally thousands of articles on many different flavors of MG [14, 46], a fact indicative of its success. Despite the multitude of MG algorithms, several of them share principles and structure, attempting to remedy the low-connectivity problem in the following interesting way. The original matrix A is

progressively coarsened in order to construct a *hierarchy* of progressively smaller matrices $\{A = A_0, A_1, \ldots, A_d\}$, along with *restriction* and *prolongation* operators that are used to map fine-space to coarse-space vectors and vice-versa. The actual solver moves through the hierarchy of matrices; each time it enters level i it performs usually one SpMV operation with A_i, a few vector-vector operations and then a restriction to the coarser level, or a prolongation to the finer level. So, if at some point during the course of algorithm the product $A_i x$ is computed, the next time a product $A_i y$ is computed, the vector y is a very complicated non-local function of x, and though each A_i inherits the low-connectivity of A, it's not possible to exploit it. In this setting one has to be able to speed-up the atomic SpMV operation.

In most cases, when the matrix A is symmetric, all matrices in the multigrid hierarchy are symmetric, and the MG solver is itself a symmetric operator. In such a case its performance as a stand-alone solver can be enhanced if it is used as a preconditioner. Going back to preconditioning, it has been observed that, since the preconditioner is only an approximation of the input matrix, performance gains are possible without sacrificing the double-precision accuracy if the preconditioner is implemented in lower precision [16]. This makes MG a case where the combined effects of index compression, precision reduction, and exploiting symmetry in (multicore) SpMV, can yield significant speed-ups, over the single-processor double-precision implementation. We demonstrate speed-ups for Combinatorial Multigrid (CMG), a recently proposed multigrid method that we discuss in Section IV. We also discuss some details pertinent to its parallel implementation, along with some applications.

II. Preliminaries

Separators. Informally, a graph has $n^\alpha, \alpha < 1$ edge separators if it is possible to find 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 S be a class of graphs that is closed under the subgraph relation (i.e., for $G \in S$, every subgraph of G is also in S). We say that S satisfies a $f(n)$-*edge separator theorem* if there are constants $\alpha < 1$ and $\beta > 0$ such that every graph $G = (V, E)$ in S with n vertices can be partitioned into two sets of vertices V_a, V_b such that $\text{cut-size}(V_a, V_b) \stackrel{def}{=} |\{(u, v) \in E \ : \ (u \in V_a \wedge v \in V_b)\}| \leq \beta f(n)$, and $|V_a|, |V_b| \leq \alpha n$ [33]. 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 [36]. 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 this work, we will also refer to vertex separators in which removing a small set of vertices partitions the remaining vertices into two disconnected and almost balanced components [33]. Separators are often applied recursively to generate a separator tree. For edge separators, all the vertices are at the leaves, and for vertex separators, some of the vertices are internal. In either case, the separator tree can be used to reorder the vertices based on an in- or post- order traversal of the tree.

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 [33]. Another common application is to partition data structures across parallel machines to minimize communication. A special case of this is when it is applied to sparse matrices. 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 the hierarchical diagonal block (HDB) structure we suggest also compresses 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 comprised 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 is referred to as the cache complexity and denoted by the notation $Q(C; B, M)$ for a computation C.

An algorithm is *cache oblivious* in the ideal-cache model if it does not take into account the size of the M or B (or any other features of the cache). One can show that if a cache oblivious algorithm 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 each level i, the algorithm will suffer at most $Q(A; M_i, B_i)$ misses at each level i. Therefore, if $Q(A; M, B)$ 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]. In particular, 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, $Q_p(n; M, B) < Q(n; M, B) + O(pMD/B)$ with probability $1 - \delta$ [3],[1] and for a shared cache using a parallel-depth-first (PDF) scheduler,

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

$Q_p(n; M + pBD, B) \leq Q(n; M, B)$ [9], where D is the depth of the computation and p is 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 under the cache oblivious analysis. The depth is important since it shows up in the bounds. For example recursive matrix multiplication, FFT, Barnes-Hut n-body code, merging, mergesort, quicksort, k-nearest neighbors, direct solvers using nested dissection, are all highly parallel and all are reasonably efficient under the cache oblivious model.

In the context of sparse-matrix vector matrix multiply the following has been shown for the Compressed Sparse Row (CSR) SpMV algorithm.

Theorem 1. *[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 $m \geq n$ non-zeros can be reordered so the CSR SpMV algorithms has $O(\log n)$ depth and $O(\lceil m/B + n/M^\alpha \rceil)$ sequential cache complexity.*

A similar result has also been shown for vertex separators although it requires a somewhat different algorithm [11].

III. Hierarchical Diagonal Blocking SpMV

In this section we describe the *hierarchical diagonal block* (HDB) representation for sparse square matrices and a SpMV routine for the representation. The representation partitions the matrix into a tree of matrices (see Figure 2). Each leaf represents a row/column and all edges (non-zero elements of the matrix) are stored in nodes of the tree. In particular each edge is stored at the least common ancestor of its two endpoints. We refer to the size of a tree as the number of leaves (rows/columns) below it. We could use the separator tree directly as the HDB tree, although this creates many levels which do not help either theoretically or in practice. Instead we coalesce the nodes of the separator tree so that sizes square at each level: $2, 4, 16, 256, 2^{16}$. Although we maintain the separator ordering among the children of a node. This is important for the cache analysis.

We can represent the submatrix at each node in various ways. If the matrix is symmetric then we can keep just the lower triangle (note the diagonal elements of the matrix are always stored at the leaves) and store it in Compressed Sparse Row. We can use skyline algorithm given in Figure 1 since there are no diagonals. Note that the matrix might have many empty rows so it is important to only store nonempty rows. This can easily be done using an additional index vector of non-empty rows as is often done in cache-blocked algorithms [48]. Also note that using the skyline algorithm prevents parallelism within the matrix. In both theory (see below) and in practice (see Section V) this can be solved by using the skyline algorithm for lower in the tree and the compressed sparse row algorithm with duplicates stored higher in the tree.

Algorithm SparseMxV(x,y,T)
$A = T.M$
$(b, t) = T.range$
if $|T| < k$) **then**
  $y[b, t] = Ax[b, t]$
else
  for all $t \in T$.children (in parallel) **do**
    SparseMxV(x,y,t)
  end for
  $y[b, t] = y[b, t] + Ax[b, t]$
end if

Fig. 3: The diagonal sparse block for Sparse-Matrix Vector Multiply

For a diagonal block of size $n \times n$ all column identifiers used in the CSR representation or integers to represent the non-empty rows can be stored in $\log n$ bits. For separable matrices the number of nonzeros is the submatrix are also bounded linearly in n. Therefore all the start locations in the CSR representation can also be represented in $\log n + k$ bits.

The SpMV routine on the HDB representation then works as shown in Figure 3. We assume the input output vectors x and y are ordered in the same order as the left to right ordering of the leafs. Each call to the algorithm recursively calls each child which will multiply all entries that are within that subtree. The pair (b, t) represents the range of x and y that the node is responsible for. Since the row sets are disjoint for each subtree these can all be done in parallel with no interaction. After the children return the algorithm adds in the contribution from the entries in the node itself (the matrix A). As mentioned in the previous paragraph, this can be done either using the skyline or CSR algorithm although it is important to only keep nonempty rows. At the base case of the recursion we can just do a standard sparse vector matrix multiply. If the $k = 1$ then this is simply adding in the contribution of the diagonal element. In practice we stop earlier.

We now to bound space, cache complexity, and depth for the HDB SpMV. In the analysis we keep track of the constant for the highest order term so we need to be specific what we mean. We assume that each non-zero value takes one word of memory. Therefore B nonzeros fit in a cache line (this is just the value 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 $m > n$ nonzero entries, or $m > n$ lower diagonal nonzero entries for a symmetric matrix. If A is stored in the HDB representation then:*

1) *A uses $m + O(n/w)$ bits.*
2) *Algorithm SparseMxV(x, y, A) runs with $m/B + O(1 + n/(Bw) + n/M^\alpha)$ cache complexity.*
3) *Algorithm SparseMxV(x, y, A) runs in $O(\log^k n)$ depth for some constant k.*

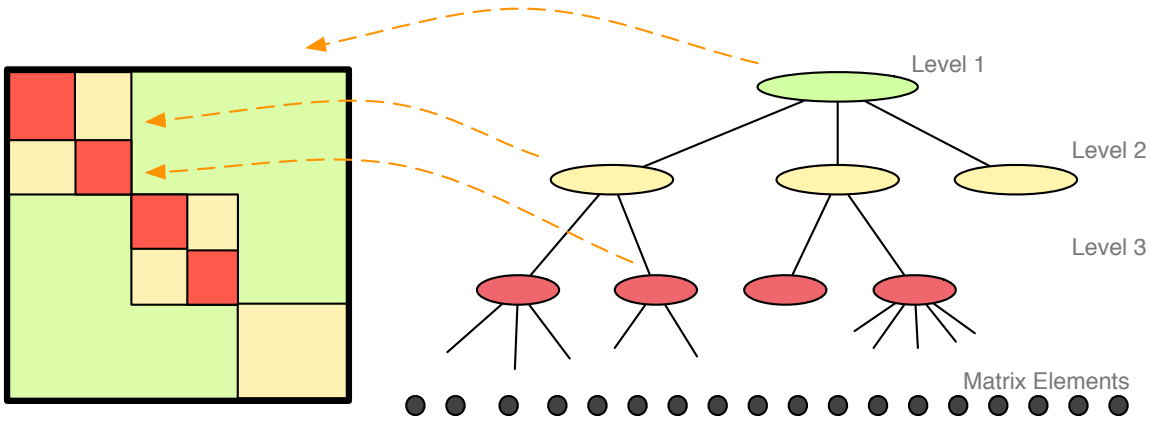

Fig. 2: Hierarchical diagonal blocking: decomposing a matrix into a tree of submatrices.

Proof: (Outline) For the HDB representation we assume that all nodes of size $n < \log^{1/\alpha}$ are stored in symmetric form (only lower triangular elements if symmetric) and all other nodes are stored in asymmetric form (including two copies even if symmetric). 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. Based on this the third item is easy to show since the depth is dominated by the matrices of size $n < \log^{1/\alpha}$ since they have to run sequentially.

We now consider space. Note that we have to consider all space beyond the values including any pointers within the tree. Here we just outline the proof which will be included in the full paper. We use the fact that a separable graph has degree bounded by a constant (the constant can depend on the separator parameters α and β, but not on the size n) [7]. We show that each level i needs a geometrically decreasing number of bits going up the tree, and a linear number at the leafs. Consider a tree of constant size k.

Finally we consider the cache complexity. The argument is similar to the argument for the CSR format [10]. We can assume all non-zero values are stored in post-order with respect to the tree traversal. Therefore the values are scanned in order and each block only loaded once and hence will cause m/B misses. Recall that we are analyzing the sequential cache complexity. Similarly all other data other than the input and output vectors x and y can be scanned in order with an appropriate layout. This will cause $O(n/(Bw))$ 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, 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 is bounded by $O(n/M^{\alpha})$ each potentially causing a miss. ■

IV. Combinatorial Multigrid and Applications

The purpose of this Section is to discuss Combinatorial Multigrid, the variant of Multigrid we used in our experiments. A thorough discussion of Multigrid algorithms is out of scope. There are many excellent survey papers and monographs on various aspects of the topic, and among them [14, 46].

Multigrid was originally conceived as a method to solve linear systems that are generated by the discretization of the Laplace (Poisson) equation over relatively nice domains. The underlying geometry of the domain leads to a hierarchy of grids $A = A_0, \ldots, A_d$ that look similar at different levels of detail; the picture that the word multigrid often invokes to mind is that of a tower of 2D grids, with sizes $2^{d-i} \times 2^{d-i}$ for $i = 0, \ldots, d$. Its provably asymptotically optimal behavior for certain classes of problems soon lead to an effort -known as *Algebraic Multigrid* (AMG)- to generalize its principles to arbitrary matrices. In contrast to classical Geometric Multigrid (GMG) where the hierarchy of grids is generated by the discretization process, AMG constructs the hierarchy of grids/matrices based only on the algebraic information contained in the matrix. AMG has been proven successful in solving more problems than GMG, though some times at the expense of robustness, a by-product of the limited theoretical understanding.

Combinatorial Multigrid (CMG) [29, 30, 31] is a recently proposed variant of Multigrid which, similarly to AMG, builds a hierarchy of matrices/graphs. The essential difference from AMG is that the hierarchy is constructed viewing the matrix as a graph, and using the *discrete geometry* of the graph, for example notions like graph separators and expansions. It is, in a way, a hybrid of GMG and AMG, or a discrete-geometric MG. The re-introduction of geometry into the problem allows us to prove sufficient and necessary conditions for the construction of a good hierarchy, and claim strong convergence guarantees for symmetric diagonally dominant (SDD) matrices with negative off-diagonals, based on recent progress on spectral graph theory and combinatorial preconditioning (see for example [12], [28]). The guarantees can be extended to more general classes of matrices via light-weight transformations to SDD problems with negative off-diagonals [13, 4]. For example, the general SDD case can be reduced to the one with only non-positive off-diagonals via a compact technique known as double cover [23].

As most variants of AMG, CMG uses the *Galerkin condition* to construct the matrix A_{i+1} from A_i. That 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 equality $A_{i+1} = R_i^T A_i R$.

CMG constructs the restriction operator R_i by grouping the variable/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, 38]. Classic AMG constructs more complicated restriction operators that can be viewed as (partially) overlapping clusters. In any case, the performance gains we see for CMG are expected to be similar in most variants of AMG, as the solve phase is dominated by the SpMV operations.

A. Why CMG on Multicores is useful—Applications

While AMG is a multigrid method designed to deal with arbitrary sparse matrices, it is probably fair to say that it has been tested and fine-tuned on systems derived by applications in engineering [25]. The graphs are regular and irregular meshes, and the magnitudes of the entries vary greatly, but in some sense regularly, reflecting the 'anisotropy' coefficient functions in the underlying PDE.

Over the last decade a wave of new applications from computed vision, machine learning, and data mining became known. The matrices derived are quite different from those in classical engineering applications. For example, in Machine Learning and Data Mining the graph connectivity may reflect a social network, or user preferences in a movie database [18]. Applications in computer vision produce 2D and 3D 'affinity' graphs, whose edges reflect the similarity between neighboring pixels. Here the graphs are more regular, but the anisotropies are far more rich in local and global scales, reflecting a wide range of features in the photographed scene.

In the case of data mining graphs, a principled contraction of the matrices/graphs in the hierarchy construction phase may create dense graphs, which are very hard -or even impossible- objects for AMG, requiring the use of additional 'sparsification' routines. Such routines, based on recent theoretical progress [32], are soon to be included in CMG. In this paper we focus on computer vision problems which don't suffer from the sparsification problem. SDD solvers have several applications to optimization problems in computer vision, used either as stand-alone solvers [21], or as subroutines -called $O(\log^2 n)$ times where n is the matrix dimension- in eigenvector-based segmentation algorithms [44], or even as subroutines in semi-definite program solvers [31].

An important new market for linear system solvers emerges with the advent of solver-based algorithms for image manipulation and analysis, especially in the context of *medical imaging*. For example, Optical Coherence Tomography (OCT) scans, that are widely used in Ophthalmology and quickly expanding in other domains produce 6-connected 3D lattices with more than 50 million nodes. The inevitably low-energy scanning, induces noise artifacts which manifest as very irregular (locally and globally) graph weights. The great potential of solver-based approaches to OCT scan analysis has been demonstrated recently in [45].

The medical imaging market is in many aspects different than the engineering market. The users, physicians and technicians, are not expected to have the background to deal with numerical convergence issues, or choosing parameters for the solver. Instead, what is needed is a truly high-performance "black-box" solver, that is fast enough to be usable in a very quick-paced or even interactive mode. The CMG solver is designed with these goals in mind.

In addition, the cost of acquiring, storing, and maintaining computing hardware is an important factor; today's higher-end multicore machines seem to be well suited for these applications.

B. CMG performance

We use CMG as our vehicle to demonstrating how the SpMV speed-ups extend to multigrid cycles. The purpose of this brief subsection is to discuss its actual performance. We report its performance only on one 2D and one 3D medical image, of sizes $1000x1500$ and $100x100x150$, which generated 4-connected and 6-connected lattices of 1.5M nodes. The CMG performance on them is typical. We note that the reported convergence rates are preliminary and not optimized. Improvements may be possible as long as the hierarchy construction abides by the sufficient and necessary conditions reported in [30].

In both cases the multigrid iteration is used as a preconditioner in a preconditioned CG iteration. The cost to construct the hierarchies in both examples is less than 5 seconds on a dual core P8600. the running time of one multigrid iteration is roughly 5 times that of one call to SpMV. The norm of the residual error $||Ax - b||$ decreases by a factor between 0.6 and 0.67 per iteration, with an average of 0.62. Merely 10 iterations are needed to reduce the norm of the residual error below 10^{-3}, and around 40 iterations to go below 10^{-10}. The graphs in these examples are weighted, with a particularly bad condition number; using CG without the preconditioning requires hundreds of iterations to achieve a residual error of 10^{-3}.

C. CMG parallel implementation details

As described above, the heart of CMG is the algorithm for constructing the hierarchy of graphs/matrices. The algorithm is very fast -in fact faster than most AMG coarsening schemes- and easily parallelizable. Its running time 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 in [31]. The pseudo-code for the CMG iteration is given in Figure 4.

When $t_i = 1$ the algorithm is known in the MG literature as the V-cycle, while when $t_i = 2$ it's known as the W-cycle. It has been known that the aggregate-based hierarchy MG, doesn't exhibit good convergence for the V-cycle, and a solution proposed in [38] is the substitution of steps 4-8, by a call to a preconditioned (by MG) Conjugate Gradient algorithm every

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

Fig. 4: The CMG solve phase

two or three levels. While this is a reasonable approach, the analysis of a recursive preconditioned CMG iteration is still an open problem, when with CMG we pursue the implementation of a solver with convergence guarantees. The theory in [28] 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

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

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 128K, 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 actually becomes slower than the plain 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 re-distributes work to lower levels of the hierarchy where as noted above the SpMV speed-ups are smaller. As a result the overall performance gains for CMG are less significant. The choice of t_i is therefore strongly preferred in the parallel setting. This may be an indication that other MG cycles that are 'heavier' on the top levels may benefit more by the fast parallel SpMV operations.

D. Single vs Double precision CMG

In the following discussion we ignore round-off errors. 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 matrix is SDD. We form $\hat{A}$ from A by writing A as $A = D + L$, 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, and ensuring that $\tilde{L}$ has zero (in single precision) row sums. Finally, we let $\tilde{L} = \tilde{D} + \tilde{L}$. 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

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

Under reasonable assumptions for the range of numbers used in A, we get $\kappa(B, \tilde{B}) < 1 + 10^{-07}$. 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 generalized to show that *each* generalized eigenvalue of the pair (A, B) us 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 to whether B or $\tilde{B}$ is the preconditioner. Indeed, in all our experiments, the two preconditioned CG iterations are virtually indistinguishable with respect to their convergence rates.

V. Implementation and Evaluation

This section describes an implementation of a hierarchical diagonal blocking SpMV routine and a study of its performance compared to other related variants. The arguments in previous sections indicate that we can use single-precision numbers in the inner-loop (preconditioner) of a CMG solver while still producing the final solution with the same accuracy as using double-precision numbers throughout. This motivates an implementation of a high-performance SpMV routine for single-precision numbers, a subroutine which we use to speedup our CMG solver.

A. Implementation of SpMV

We implemented an SpMV routine for single-precision symmetric matrices using the descriptions from Section III with the following simplifications: Whereas the algorithmic description stops the recursion when it reaches singleton elements, the actual implementation stops it after two levels. Furthermore, the diagonal blocks in the first level have size approximately 32K. In this implementation, therefore, a matrix

Machine Model	Speed (Ghz)	Layout (#chips×#cores)
Intel Nehalem X5550	2.66	2×4
Intel Harpertown E5440	2.83	2×4
AMD Shanghai 2384	2.70	2×4

TABLE I: Characteristics of the architectures used in our study.

is represented as on-diagonal entries, diagonal-block entries, and off-block entries (in a manner similar Figure 2 with only 2 inner-node levels and a level of leaf nodes). This simplified representation is well-suited for implementation and is shown to deliver 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 perform experiments on 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. There is a corresponding version for single-precision numbers, denoted by "simple par. (single)" in the figures. 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. Note that although the code is really simple, its performance matches, within 1%, highly optimized kernels for SpMV, such as Intel Math Kernel Library [2]. We decided to work with our own implementation because of 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]. These programs were then compiled with Intel Cilk++ build 8503 using the optimization flag `-O2`. To avoid the overhead in 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 3 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.

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.

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

Matrix	#rows/cols	#non-zero
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.

Matrix	Speedup simple par. (double)	Speedup blocked par. (single)
2d-A	3.9x	6.1x
3d-A	3.7x	6.7x
af_shell10	4.3x	10.7x
audikw_1	4.0x	10.8x
bone010	3.7x	9.6x
ecology2	3.4x	5.4x
nd24k	3.9x	9.7x
nlpkkt120	3.8x	7.8x
pwtk	3.7x	9.4x

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

Datasets. Our study involves a diverse collection of large sparse matrices, gathered from the University of Florida Matrix Collection [17] and a collection of mesh matrices generated from vision-inspired applications. We present a summary of these matrices in Table II.

Since the CMG solver works with the assumption that the matrix is SDD, we made negative all off-diagonal entries of the matrices, keeping their magnitude, and we adjusted the diagonals to get zero row-sums. The purpose of this was to test the performance of CMG on various sparse patterns.

C. Performance of SpMV

The first set of experiments concerns the performance of SpMV, which constitutes a large fraction of the work spent in solving a linear system as well as other high-performance computing applications. In these experiments, we are especially interested in understanding how the ideas outlined in previous sections perform on a variety of sparse matrices.

Speedup. Figure 5 and Table III show the performance of various SpMV routines (in GFlops) on the matrices in our collection. Several things are clear from this figure. First, on all these matrices, a simple parallel algorithm speeds up SpMV by 3.4x–4.5x. As will be apparent from the study of memory bandwidth below, we cannot improve the performance much further without any data reduction.

Second, but more importantly, both diagonal blocking and precision reduction can help enhance the speed of SpMV, but *neither idea alone yields as much performance improvement as their combination.* As the figure shows, in both cases, the speed is enhanced as we reduce the data; however, the maximum benefit is achieved through the combination of both ideas.

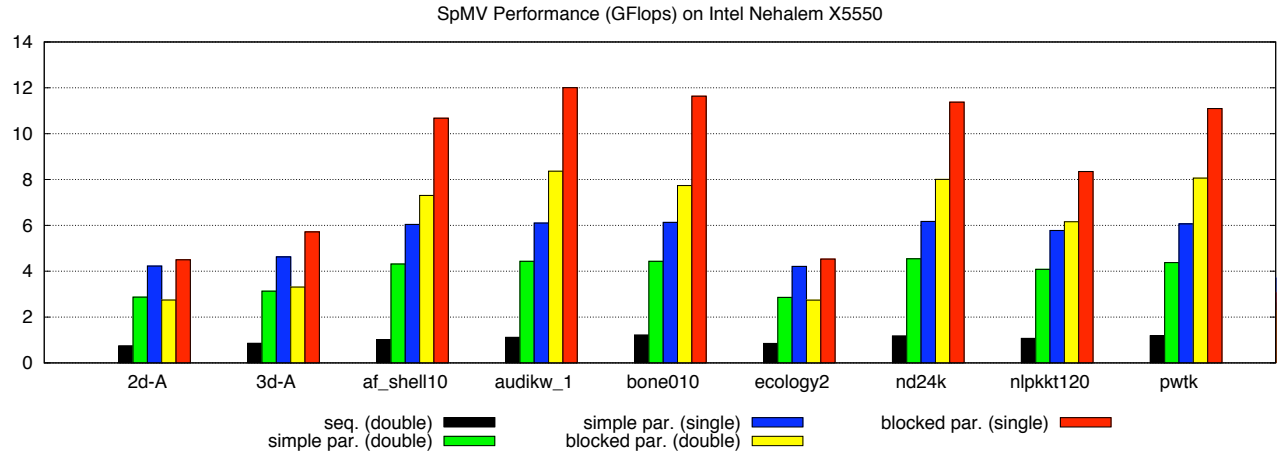

Fig. 5: Performance of different SpMV routines (**in GFlops**) on a variety of matrices.

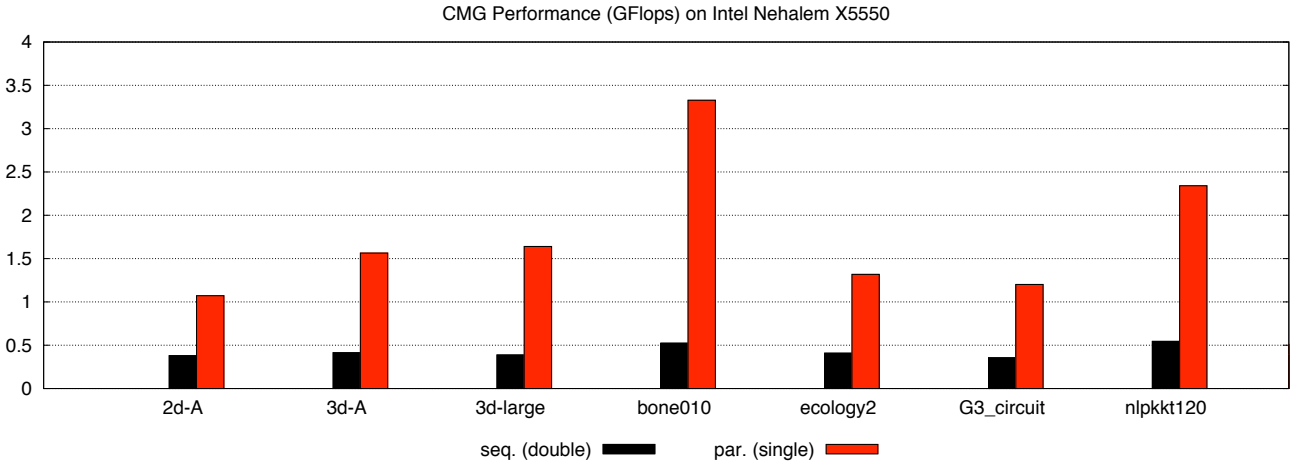

Fig. 6: Performance of a CMG solve iteration (in GFlops) on different linear systems.

By replacing double-precision numbers with single-precision numbers, we use 4 bytes per matrix entry instead of 8. Further, by using the (simplified) hierarchical diagonal blocking with the top-level block size $\sim$ 32K, we can represent the indicies of the entries in the diagonal blocks using 16-bit words, a saving from 32-bit words used to represent matrix indices in a normal CSR format. Combining both ideas, we not only further reduce the bandwidth but also improve the cache locality due to blocking. This is reflected in the additional speedup of more than a factor of 2 in the speedup of the single-precision blocked parallel version when compared to the speedup of the simple double-precision parallel code.

Memory Bandwidth. Presented in Figure 7 are bandwidth numbers for different SpMV routines on the three machines we consider. First and most importantly, *blocked parallel single precision scales the best on all three machines.* In all cases, the performance seems to be compute bound for a single core but reach close to peak bandwidth on 8 cores. On the Nehalem, it achieves a factor of 6.5 speedup, compared to a factor of less than 4 for the simple double-precision parallel SpMV.

Second, all the algorithms almost reach the same bandwidth limit on 8 processors. In fact, we have not been able to get better bandwidth on the STREAM benchmarks [35, 34]. The trend in bandwidth 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 due to the limited bandwidth.

D. Performance of CMG

The previous section shows substantial improvements in SpMV performance and the results from Section IV-D indicates that we can afford to reduce the precision from double to single in the preconditioner without harming the accuracy of the final solution. Thus, we should benefit from using our single-precision SpMV implementation in a combinatorial multigrid solver. We quantitatively investigate this in the following.

Figure 6 shows the performance of *one call* to the precondition solve of two CMG solvers. The first CMG solver is a sequential program which uses the baseline implementation of SpMV and double-precision numbers throughout. The other

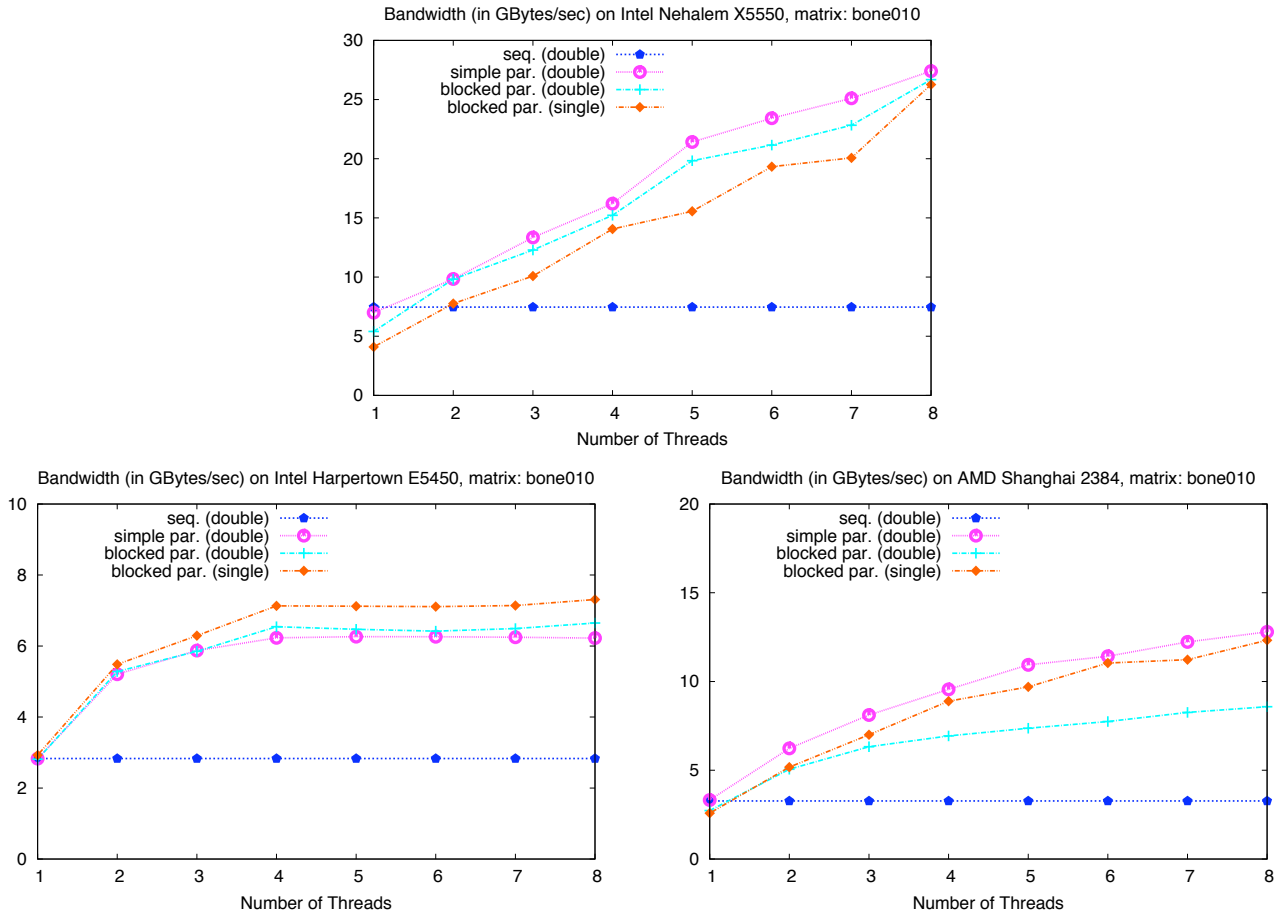

Fig. 7: Bandwidth consumption of SpMV on Intel Nehalem X5550, AMD Shanghai 2384, and Intel Harpertown E5440 as the number of threads is varied.

solver is a parallel program that uses the SpMV routine and single-precision as described in previous sections; vector-vector operations in the CMG programs are also parallelized in a straightforward manner.

From the figure, several things are clear. First, the speedup—the ratio between the performance of the 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 2x, with the best case reaching beyond 4x. Second, the speedup of the CMG solver seems to be proportional to the speedup of SpMV, but not as big. 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 lower parallelization potential, for example vector-vector operators and SpMV operations with smaller matrices.

VI. Conclusions

We have demonstrated substantial speedups with methods that do not require compression and decompression code/hardware. It would be interesting to try more sophisticated compression methods such as difference encoding on either the matrix entries and/or the indices. At the present time memory bandwidth seems to be substantially more precious than chip real estate. Thus, special purpose hardware for compression and decompression may make a big win in speed at relatively modest cost.

References

[1] Intel + clik, 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. ISSN 0895-4798. doi: http://dx.doi.org/10.1137/060675940.

[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. ISBN 1-58113-840-7. 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. ISSN 0895-4798.

[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. ISBN 0-89871-462-1.

[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. doi: http://doi.acm.org/10.1145/1377596.1377597.

[17] T. A. Davis. University of florida sparse matrix collection. *NA Digest*, 92, 1994. URL http://www.cise.ufl.edu/research/sparse/matrices/.

[18] F. Fouss, A. Pirotte, and M. Saerens. A novel way of computing similarities between nodes of a graph, with application to collaborative recommendation. In *ACM International Conference on Web Intelligence*, pages 550–556, 2005.

[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. ISBN 978-3-540-88685-3.

[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] V. E. Henson and U. M. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics: Transactions of IMACS*, 41(1):155–177, 2002. URL citeseer.ist.psu.edu/henson00boomeramg.html.

[26] 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.

[27] 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.

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

[29] I. Koutis and G. Miller. The Combinatorial Multigrid Solver. Conference Talk, March 2009.

[30] 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.

[31] 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.

[32] I. Koutis, G. L. Miller, and R. Peng. Approaching optimality for solving sdd systems. *CoRR*, abs/1003.2958, 2010.

[33] 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. URL http://link.aip.org/link/?SMM/36/177/1.

[34] 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/.

[35] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

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

[37] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. A. Yelick. Minimizing communication in sparse matrix solvers. In *SC*. ACM, 2009. ISBN 978-1-60558-744-8.

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

[39] 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.

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

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

[42] 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.

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

[44] 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.

[45] 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.

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

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

[48] 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.