Reducing Memory Access Latencies using Data Compression in Sparse, Iterative Linear Solvers

An All-College Thesis

College of Saint Benedict/Saint John's University

by Neil Lindquist
April 2019

Project Title: Reducing Memory Access Latencies using Data Compression in Sparse, Iterative Linear Solvers

Approved by:

Mike Heroux
Scientist in Residence

Robert Hesse
Associate Professor of Math

Jeremy Iverson
Assistant Professor of Computer Science

Bret Benesh
Chair, Department of Mathematics

Imad Rahal
Chair, Department of Computer Science

Director, All College Thesis Program

Abstract

Solving large, sparse systems of linear equations plays a significant role in certain scientific computations, such as approximating the solutions of partial differential equations. However, solvers for these types of problems usually spend most of their time fetching data from main memory. In an effort to improve the performance of these solvers, this work explores using data compression to reduce the amount of data that needs to be fetched from main memory. Some compression methods were found that improve the performance of the solver and problem found in the HPCG benchmark, with an increase in floating point operations per second of up to 84%. These results indicate that, if similar improvements can be made with other linear systems, compression could improve the performance of real-world solvers.

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

Solving large, sparse linear systems of equations plays a vital role in certain scientific computations. For example, the finite element method solves a system of linear equations to approximate the solution to certain partial differential equations [15]. These problems can be large, with easily millions of variables or more [2]. So, solving these problems efficiently requires a fast linear solver.

Iterative solvers are often used to solve these large, sparse systems. These solvers take an initial guess then improve it until it is within some tolerance [15]. On modern computers, these solvers often spend most of their time fetching data from main memory to the processor where the actual computation is done [11]. This work tries to improve the performance of iterative solvers by compressing the data to reduce the time spent accessing main memory.

To avoid implementing an entire sparse linear solver, the High Performance Conjugate Gradient (HPCG) benchmark was used as the initial codebase for the project [4]. The HPCG benchmark is designed to measure the performance of computing systems when processing sparse solvers and does so by solving one such test problem. In addition, as a benchmark, HPCG has built in measurements of elapsed time, solver iterations and the number of floating point operations that needed to be computed. These factors all make the HPCG codebase a natural starting point for developing improvements to sparse, iterative linear solvers.

There are three main data structures that were experimented with in this project: the vector values, the matrix values, and the matrix indices. For each of these data structures, compression methods were found that were able to fulfill the requirements on read and write access. Additionally, two models were constructed to estimate the minimum performance a compression method would need to outperform the baseline. Both models indicated that vector values must be decoded with only a few instructions, but that decoding the matrix values has a much larger, but still limited, window for improvement. In the actual tests, a couple of configurations were found to be able to outperform the baseline implementation, with an increase in performance of up to 84%.

1.1 Previous Work

First, this work draws heavily on existing data compression methods. Some of the algorithms used were designed with scientific computations in mind, such as ZFP and SZ compression [13, 3]. Other algorithms are more general purpose, such as Huffman and Elias Codings [8, 5]. Section 2.2 goes into detail on what compression methods were used and how they work.

Much work has been done on various aspects of utilizing single precision floating point numbers while retaining the accuracy of double precision numbers in iterative linear solvers. One approach, which this work draws inspiration from, is to apply the preconditioner using single precision, while otherwise using double precision, which result in similar accuracy unless the matrix is poorly conditioned [1, 7]. This strategy of mixing floating point precision for various

parts of the solver algorithm leads to the use of making certain vectors single precision, as described in Section 2.2.2.

Another effort at compressing large, sparse Linear Systems is Compressed Column Index (CCI) format to store matrices [11]. This format is based on Compressed Sparse Row (CSR) matrix format except uses compression to reduce the size of the matrix indices. The index compression used by CCI is described in Section 2.2.8 and tested in this project. This project generalizes the ideas of CCI matrices, both by compressing additional data structures and using additional compression methods. However, only a single matrix is tested for this project, as opposed to the suite of matrices used to look at the performance of CCI matrices.

1.2 Mathematics of Conjugate Gradient

Conjugate Gradient is the iterative solver used by HPCG [4]. Symmetric, positive definite matrices will guarantee the converge of Conjugate Gradient to the correct solution within n iterations, where n is the rows of the matrix, when using exact algebra [15]. More importantly, Conjugate Gradient can be used as in iterative method, providing a solution, $\vec{x}$, where $\left\|\mathbf{A}\vec{x} - \vec{b}\right\|$ is within some tolerance, after significantly fewer than n iterations, allowing it to find solutions to problems where computing n iterations is infeasible [17]. As an iterative method, Conjugate Gradient forms update directions from Krylov subspaces of the form $\mathcal{K}_n(\vec{r}_0, \mathbf{A}) = \text{span}(\vec{r}_0, \mathbf{A}\vec{r}_0, \mathbf{A}^2\vec{r}_0, \ldots, \mathbf{A}^n\vec{r}_0)$, where $\vec{r} = \vec{b} - \mathbf{A}\vec{x}_0$.

To understand the Conjugate Gradient, first consider the quadratic form of $\mathbf{A}\vec{x} = \vec{b}$. The quadratic form is a function $f : \mathbb{R}^n \to \mathbb{R}$ where

$$f(\vec{x}) = \tfrac{1}{2}\vec{x}^T\mathbf{A}\vec{x} - \vec{b}\cdot\vec{x} + c \tag{1}$$

for some $c \in \mathbb{R}$. Note that

$$\nabla f(\vec{x}) = \tfrac{1}{2}\left(\mathbf{A} + \mathbf{A}^T\right)\vec{x} - \vec{b}$$

Then, when $\mathbf{A}$ is symmetric,

$$\nabla f(\vec{x}) = \mathbf{A}\vec{x} - \vec{b}$$

So, the solution to $\mathbf{A}\vec{x} = \vec{b}$ is the sole critical point of f [14]. Since $\mathbf{A}$ is the Hessian matrix of f at the point, if $\mathbf{A}$ is positive definite, then that critical point is a minimum. Thus, if $\mathbf{A}$ is a symmetric, positive definite matrix, then the minimum of f is the solution to $\mathbf{A}\vec{x} = \vec{b}$ [17].

The method of Steepest Decent is useful for understanding Conjugate Gradient, because they both use a similar approach to minimize Equation 1, and thus solve $\mathbf{A}\vec{x} = \vec{b}$. This shared approach is to take an initial $\vec{x}_0$ and move downwards in the steepest direction, within certain constraints, of the surface defined by Equation 1 [14]. Because the gradient at a point is the direction of maximal increase, $\vec{x}$ should be moved in the opposite direction of the gradient. Thus, to compute the next value of $\vec{x}$, use

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i\vec{r}_i \tag{2}$$

Algorithm 1 Steepest Decent [17].

$\vec{r}_0 \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
for $i = 0, 1, \ldots$ **until** $\|\vec{r}_i\| \leq \epsilon$ **do**
  $\alpha_i \leftarrow \frac{\vec{r}_i \cdot \vec{r}_i}{\vec{r}_i \cdot \mathbf{A}\vec{r}_i}$
  $\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{r}_i$
  $\vec{r}_{i+1} = \vec{r}_i - \alpha \mathbf{A}\vec{r}_i$
end for

for some $\alpha_i > 0$ and where $\vec{r}_i = -\nabla f(\vec{x}_i) = \vec{b} - \mathbf{A}\vec{x}_i$ is the residual of $\vec{x}_i$. Since $\mathbf{A}\vec{x} = \vec{b}$ is the only critical point and a minimum of the quadratic function, f, the ideal value of α_i is the one that minimizes $f(\vec{x}_{i+1})$. Thus, choose α_i such that

$$\begin{aligned} 0 &= \tfrac{\mathrm{d}}{\mathrm{d}\alpha_i} f(\vec{x}_{i+1}) \\ &= \tfrac{\mathrm{d}}{\mathrm{d}\alpha_i} f(\vec{x}_i + \alpha \vec{r}_i) \\ \alpha_i &= \frac{\vec{r}_i \cdot \vec{r}_i}{\vec{r}_i \cdot \mathbf{A}\vec{r}_i}. \end{aligned}$$

Note that by using Equation 2, we can derive

$$\vec{r}_{i+1} = \vec{r}_i - \alpha \mathbf{A}\vec{r}_i. \tag{3}$$

Because $\mathbf{A}\vec{r}_i$ is already computed to find α_i, using Equation 3 to compute the residual results in one less matrix-vector product per iteration. Algorithm 1 shows the resulting algorithm.

Example 1. Consider the linear system

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$$

and use $c = 0$. Note that the solution is

$$\vec{x} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}.$$

When starting at the origin, the iteration of Method of Steepest Decent becomes

$$\begin{aligned} \vec{x}_0 &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} & \vec{r}_0 &= \begin{bmatrix} 5 \\ 5 \end{bmatrix} & \alpha_0 &= 2/7 \\ \vec{x}_1 &= \begin{bmatrix} 10/7 \\ 10/7 \end{bmatrix} & \vec{r}_1 &= \begin{bmatrix} 5/7 \\ -5/7 \end{bmatrix} & \alpha_1 &= 2/3 \\ \vec{x}_2 &= \begin{bmatrix} 40/21 \\ 20/21 \end{bmatrix} & \vec{r}_2 &= \begin{bmatrix} 5/21 \\ 5/21 \end{bmatrix} & \alpha_2 &= 2/7 \\ \vec{x}_3 &= \begin{bmatrix} 290/147 \\ 50/49 \end{bmatrix} & \vec{r}_3 &= \begin{bmatrix} 5/147 \\ -5/147 \end{bmatrix} & \alpha_3 &= 2/3 \\ \vdots & & \vdots & & \vdots & \end{aligned}$$

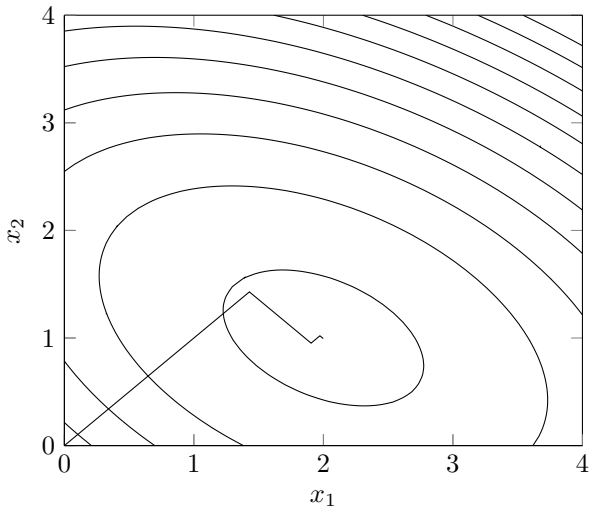


Figure 1: Contour graph of the quadratic function and the first six values of $\vec{x}$ produced by steepest decent for Example 1.

The $\vec{x}_i$s are plotted with a contour graph of the quadratic form in Figure 1. ■

The Conjugate Directions family of linear solvers, of which Conjugate Gradient is a member of, attempts to improve on the number of iterations needed by Steepest Decent. [17]. Note that, in Example 1, the directions of $\vec{r}_0$ and $\vec{r}_2$ are the same and the directions of $\vec{r}_1$ and $\vec{r}_3$ are the same. Thus, the same direction is traversed multiple times. Additionally, note that the two sets of residual directions are perpendicular to each other. Conjugate Directions attempts to improve on this, by making the search directions, $\vec{d}_0, \vec{d}_1, \ldots$, **A**-orthogonal to each other and only moving $\vec{x}$ once in each search direction. Two vectors, $\vec{u}, \vec{v}$, are **A**-orthogonal, or conjugate, if $\vec{u}^T \mathbf{A} \vec{v} = 0$. The requirement for Conjugate Directions is to make $\vec{e}_{i+1}$ **A**-orthogonal to $\vec{d}_i$, where $\vec{e}_i = \vec{x}_i - \mathbf{A}^{-1}\vec{b}$ is the error of $\vec{x}_i$. The computation of α_i changes to find the minimal value along $\vec{d}_i$ instead of $\vec{r}_i$.

$$\alpha_i = \frac{\vec{d}_i^T \vec{r}_i}{\vec{d}_i^T \mathbf{A} \vec{d}_i}.$$

Conjugate Gradient is a form of Conjugate Directions where the residuals are made to be **A**-orthogonal to each other [17]. This is done using the Conjugate Gram-Schmidt Process. To do this, each search direction, $\vec{d}_i$ is computed by taking $\vec{r}_i$ and removing any components that are not **A**-orthogonal to the previous $\vec{d}$'s [17]. So, let $\vec{d}_0 = \vec{r}_0$ and for $i > 0$ let

$$\vec{d}_i = \vec{r}_i + \sum_{k=0}^{i-1} \beta_{(i,k)} \vec{d}_k$$

Algorithm 2 Conjugate Gradient [15].

$\vec{r}_0 \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
$\vec{d}_0 \leftarrow \vec{r}_0$
for $i = 0, 1, \ldots$ **until** $\|\vec{r}_i\| \leq \epsilon$ **do**
  $\alpha_i \leftarrow \frac{\vec{r}_i \cdot \vec{r}_i}{\vec{d}_i \cdot \mathbf{A}\vec{d}_i}$
  $\vec{x}_{i+1} \leftarrow \vec{x}_i + \alpha_i \vec{d}_i$
  $\vec{r}_{i+1} \leftarrow \vec{r}_i + \alpha_i \mathbf{A}\vec{d}_i$
  $\beta_{i+1} \leftarrow \frac{\vec{r}_{i+1} \cdot \vec{r}_{i+1}}{\vec{r}_i \cdot \vec{r}_i}$
  $\vec{r}_{i+1} + \beta_{i+1}\vec{d}_i$
end for

with $\beta_{(i,k)}$ defined for $i > k$. Then, solving for $\beta_{(i,k)}$ gives

$$\beta_{(i,k)} = -\frac{\vec{r}_i \cdot \mathbf{A}\vec{d}_i}{\vec{d}_j \cdot \mathbf{A}\vec{d}_j}.$$

Note that each residual is orthogonal to the previous search directions, and thus the previous residuals. So, it can be shown that $\vec{r}_{i+1}$ is $\mathbf{A}$-orthogonal to all previous search directions, except $\vec{d}_i$ [17]. Then, $\beta_{(i,k)} = 0$ for $i - 1 \neq k$. To simplify notation, let $\beta_i = \beta_{(i,i-1)}$. So, each new search direction can then be computed by

$$\vec{d}_i = \vec{r}_i + \beta_i \vec{d}_{i-1}.$$

Algorithm 2 shows the final Conjugate Gradient algorithm.

Example 2. Consider the linear system used in Example 1 where

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}.$$

The result of applying Conjugate Gradient is

$$\vec{x}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \vec{r}_0 = \begin{bmatrix} 5 \\ 5 \end{bmatrix} \quad \vec{d}_0 = \begin{bmatrix} 5 \\ 5 \end{bmatrix} \quad \alpha_0 = 2/7$$
$$\vec{x}_1 = \begin{bmatrix} 10/7 \\ 10/7 \end{bmatrix} \quad \vec{r}_1 = \begin{bmatrix} 5/7 \\ -5/7 \end{bmatrix} \quad \beta_1 = 1/49 \quad \vec{d}_1 = \begin{bmatrix} 40/49 \\ -30/49 \end{bmatrix} \quad \alpha_1 = 7/10$$
$$\vec{x}_2 = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \quad \vec{r}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Note that after two iterations, $\vec{x}$ reaches the exact solution, compared to the iterations of Steepest Decent in Example 1. Figure 2 shows the values of $\vec{x}$ with the contour graph of the quadratic function. ■

One way to improve the Conjugate Gradient method is to precondition the system [15]. Instead of solving the original system, $\mathbf{A}\vec{x} = \vec{b}$, a Preconditioned

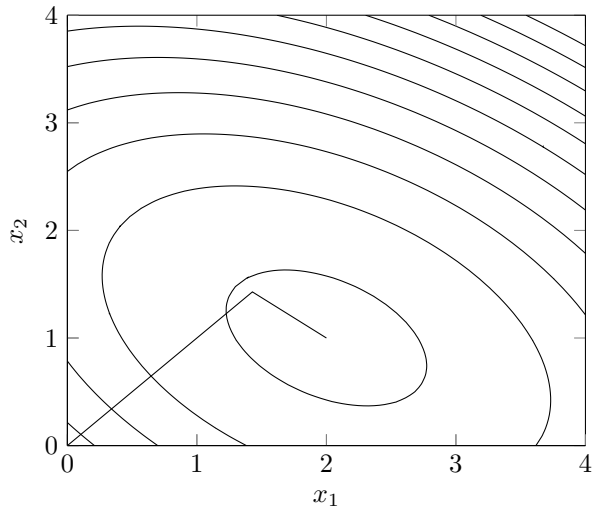


Figure 2: Contour graph of the quadratic function and each value of $\vec{x}$ produced by Conjugate Gradient for Example 2.

Conjugate Gradient solves $\mathbf{M}^{-1}\left(\mathbf{A}\vec{x} - \vec{b}\right) = 0$ instead, where $\mathbf{M}^{-1}$ is the preconditioner. Note that $\mathbf{M}$ should be similar to $\mathbf{A}$, but $\mathbf{M}^{-1}$ should be easier to compute than $\mathbf{A}^{-1}$. When $\mathbf{M}$ is similar to $\mathbf{A}$, the system becomes close to $\mathbf{I}\vec{x} = \mathbf{M}^{-1}\vec{b}$, which is easy to solve if $\mathbf{M}^{-1}\vec{b}$ can be computed cheaply. Algorithm 3 shows the preconditioned variant of the Conjugate Gradient.

1.3 Mathematics of the Multigrid Preconditioner

Multigrid solvers are a class of methods designed for solving discretized partial differential equations (PDEs) and take advantage of more information that just the coefficient matrix and the right-hand side [15]. Specifically, the solvers use discretizations of varying mesh sizes to improve performance of relaxation-based solvers. In HPCG, a multigrid solver with high tolerance is used as the preconditioner [4]. Because the solver provides an approximation to $\mathbf{A}^{-1}$, the preconditioned matrix approximates $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$. This reduces the condition number of the linear system, and so, reduces the number of iterations needed for Conjugate Gradient to converge [15].

The multigrid method uses meshes of varying sizes to improve performance of a relaxation style iterative solver [15]. Relaxation based solvers "relax" a few coordinates at a time to eliminate a few components of the current residual. Most of these solvers can quickly reduce the components of the residual in the direction of eigenvectors associated with large eigenvalues of the iteration matrix. Such eigenvectors are called high frequency modes. The other components, eigenvectors called low frequency modes, are difficult to reduce with standard relaxation. However, on a coarser mesh, many of these low frequency modes

Algorithm 3 Preconditioned Conjugate Gradient [15].

$\vec{r}_0 \leftarrow \vec{b} - \mathbf{A}\vec{x}_0$
$\vec{z}_0 \leftarrow \mathbf{M}^{-1}\vec{r}_0$
$\vec{d}_0 \leftarrow \vec{z}_0$
for $i = 0, 1, \ldots$ **until** $\|\vec{r}_i\| \leq \epsilon$ **do**
  $\alpha_i \leftarrow \frac{\vec{r}_i \cdot \vec{z}_i}{\vec{d}_i \cdot \mathbf{A}\vec{d}_i}$
  $\vec{x}_{i+1} \leftarrow \vec{x}_i + \alpha_i \vec{d}_i$
  $\vec{r}_{i+1} \leftarrow \vec{r}_i + \alpha_i \mathbf{A}\vec{d}_i$
  $\vec{z}_{i+1} \leftarrow \mathbf{M}^{-1}\vec{r}_{i+1}$
  $\beta_{i+1} \leftarrow \frac{\vec{r}_{i+1} \cdot \vec{z}_{i+1}}{\vec{r}_i \cdot \vec{z}_i}$
  $\vec{d}_{i+1} \leftarrow \vec{z}_{i+1} + \beta_{i+1}\vec{d}_i$
end for

are mapped to high frequency modes [15]. Thus, by applying a relaxation type iterative solver at various mesh sizes, the various components of the residual can be reduced quickly.

In HPCG, a symmetric Gauss-Seidel iteration is used by the multigrid as the relaxation iteration solver at each level of coarseness [4]. The symmetric Gauss-Seidel iteration consists of a forward Gauss-Seidel iteration followed by a backward Gauss-Seidel iteration. Letting $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$ where $\mathbf{L}$ is strictly lower triangular, $\mathbf{D}$ is diagonal, and $\mathbf{U}$ is strictly upper triangular, the iteration can be represented by

$$\vec{x}_{i^*} = \mathbf{D}^{-1}\left(\vec{b} - \mathbf{L}\vec{x}_{i^*} - \mathbf{U}\vec{x}_i\right)$$
$$\vec{x}_{i+1} = \mathbf{D}^{-1}\left(\vec{b} - \mathbf{U}\vec{x}_{i+1} - \mathbf{L}\vec{x}_{i^*}\right)$$

with $\vec{x}_{i^*}$ representing an intermediate vector. Note that while $\vec{x}_{i^*}$ and $\vec{x}_{i+1}$ are on both sides of the equation where they are respectively computed, they can be computed with this formulation by computing the entries in order as they become available for the product with L and U, respectively, by iterating in row order then in reverse row order. So, the update of a Gauss-Seidel step can be computed in place.

2 Problem Implementation

This project uses the High Performance Conjugate Gradient (HPCG) benchmark as the baseline implementation. This means that an implementation of the Conjugate Gradient algorithm with a multigrid precondition variant is used as the linear solver [4]. The benchmark's linear system is a discretization of a steady-state heat equation problem in three dimensions. The zero vector is the initial value for x.

The problem used to create the linear system used by HPCG, and thus by this project, is a three-dimensional partial differential equation (PDE) model [4].

This problem is approximating the function $u(x, y, z)$ over the three-dimensional rectangular region $\Omega \subset \mathbb{R}^3$ such that

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = 0,$$

with $u(x, y, z) = 1$ along the boundaries of Ω. Note that the solution is $u(x, y, z) = 1$ for the region Ω. The linear system is created by using the finite difference method with a 27-point stencil on the PDE over a rectangular grid with nodes of fixed distance. The matrix's diagonal consists of the value 26, and -1's fill the entries for the row's 26 grid neighbors. The right-hand side of the equation has a value of 14 for corner points, 12 for edge points, 9 for side points and 0 for interior points [10]. The solution vector consists of all 1's.

As HPCG is designed to emulate the performance characteristics of real-world problems without needing to be a robust solver, it only uses 3 levels of grid coarseness with only a single smoother iteration at the coarsest grid level. The smoother used by the multigrid is based on a symmetric Gauss-Seidel step; however, values are only synchronized between processors at the beginning of the step. The restriction operation simply samples half the points in dimension, resulting in a reduction of grid size by a factor of eight in each level of coarseness. To prolong the coarse grids, each coarse point is added to the fine point it was sampled from.

The problem is distributed over 60 processes, each with a cubic subproblem 96 nodes per side. The processors are distributed in a rectangular prism of size 5 by 4 by 3 processors. The resulting linear system has 53,084,160 total rows. MPI is used for interprocess communication.

2.1 Data Access Patterns of High Performance Conjugate Gradient

The Conjugate Gradient and Multigrid implementations in HPCG do not directly access the matrix and vector values, but instead use low-level functions to manipulate the data structures [4]. These low-level functions include copying a vector, setting a vector to zero, the dot product, a scaled vector sum, the matrix vector product, the symmetric Gauss-Seidel step, the multigrid restriction, and the multigrid prolongation. Further data accessing functions exist in HPCG, however, they are not part of the timing. So, any additionally restrictions can be overcome by converting to an uncompressed format, applying the function, then recompressing. The low-level functions used in the timed section of the code can be viewed together to produce the overall data access requirements, For the matrices, the matrices do not need to be mutable, the rows need to be readable in both a forward and backward iteration, the data for a given row has no restriction on its read order, and the diagonal for a given row must be accessible. The vectors, on the other hand, need both random read and write access, with the writes being immediately accessible to future reads.

In addition to the restrictions on usable compression schemes imposed by the data access patterns, they influence the effectiveness of compression schemes.

Note that in the inner loop of both the sparse matrix vector product and the symmetric Gauss-Seidel step do not have a data dependency between matrix values and the vector values; however, the vector values are dependent on the matrix indices. So, the matrix values can be fetched in parallel to the matrix indices and vector values [6]. This will result in ineffective compression when just compressing one part of the problem, as discussed in Section 2.2.9.

Copying a vector and setting a vector to zero provide the least data access requirements. Note that a vector's content can by copied by transferring the current representation of the values without any processing. Setting a vector to zero merely requires the ability to write vector values. These functions add little to the data access requirements and are both simple to implement with alternative vector representation.

The dot product and sum of scaled vectors are both straightforward functions. Each of them iterates over two or three vectors and applies a few arithmetic operations. The dot product accumulates the sum of the product of the pair of vector entries across the iterations. The sum of scaled vectors computes $w_i = \alpha x_i + \beta y_i$ for each set of entries. Note that the only data iteration between rows in either of these operations is the sum in the dot product, however addition is an associative operation. Thus, these functions can be arbitrarily parallelized or have their iteration reordered.

The matrix vector product iterates once over the rows and for each row sums the nonzero entries times the vectors corresponding entries [4]. Both the rows and the sum in each row may be iterated in any order or in parallel. Thus, the matrix information can be compressed for any iteration order of rows and any iteration order of the values in each row. However, the vector information must be able to be read at an arbitrary index. For each iteration, the matrix information is read only once, and the vector entries are read for each nonzero value in the corresponding column (8 to 27 times for this matrix). So, assuming the problem is too large for the matrix to fit entirely in the memory caches, the matrix data will always need to be read from main memory, while vector data will be able to utilize caches, resulting in up to 27-fold fewer reads than the matrix data. This hints that the compressing matrix information is more likely to provide an increase in performance of the matrix vector product than compressing the vector information.

The symmetric Gauss-Seidel step is similar to the sparse matrix-vector product, with added complications. First, the step has two iterations, one forward and one backward. Instead of simply summing the row-vector product, each row does the following calculation

$$x_i \leftarrow b_i - \frac{1}{a_{ii}} \sum_{j=1}^{n} a_{ij} x_j$$

where only terms with nonzero a_{ij} are computed, x_i, b_i are the ith elements of $\vec{x}, \vec{b}$ respectively and a_{ij} is the entry of $\mathbf{A}$ in the ith row and jth column [4]. Note that each x_i is used immediately in the subsequent rows, this means that any deviation from the base row iteration order or any parallelization of the rows

Strategy	Vector Values	Matrix Values	Matrix Indices
Single Precision	Yes	Yes	Not Able
Mixed Precision	Yes	Not Able	Not Able
1 Bit	Not Able	Yes	Not Able
Squeeze (SZ)	Yes	Yes	Yes
ZFP	Yes	Yes	Not Able
Elias Gamma	Not Able	Not Able	Yes
Elias Delta	Not Able	Not Able	Yes
Huffman	Not Able	No	Yes
Op Code	Not Able	Not Able	Yes

Figure 3: Overview of compression strategies.

may reduce the effectiveness of the step. Because any delay in writing the new values to $\vec{x}$ results in effectively parallelizing the iteration of the rows, the vector values must be written immediately or within a few iterations. Additionally, the Gauss-Seidel step has the additional requirement that the matrix diagonal of the current row must be accessible.

The restriction and prolongation functions used in the multigrid are the last matrix and vector value accessing functions used in the Conjugate Gradient implementation. Restriction samples points from two fine grid vectors and stores the difference in a coarse grid vector. Prolongation takes the entries in a coarse grid vector and adds them to select fine grid vectors. So, between these two functions, random read and write access is needed by vectors in all but the coarsest mesh.

2.2 Compression Strategies

Numerous compression strategies were considered for this project. Figure 3 lists the compressions tried for each of the main data structures. Note that most compression methods were only used with one or two of the data types, even if able to be reasonably used within the constraints of other types of data.

2.2.1 Restrictions on Compression Strategies

The restrictions on usable compression strategies primarily come from the data access requirements described in Section 2.1. These requirements were that matrix rows need to be readable in both a forward and backward iteration, the diagonal for a given row must be accessible, and the vectors have both random read access and random, immediate write access. Due to the highly regular nature of the particular matrix used and the existence of solvers specially optimized for solving this type of problem, the requirement that all compression techniques can handle any sparse matrix was added to increase the usefulness of this work [15]. Although, an exception was made to the requirement to handle general matrices for the 1-bit Compression described in Section 2.2.3 as

that compression method is designed to provide an upper bound for improvements from compressing matrix values. Finally, integer compression was limited to lossless compression methods to ensure that the proper vector entries were fetched, while floating point compression was allowed to be lossy.

Note that some cleverness can be used to work around some restrictions. By compressing the data in small blocks, sequential compression strategies can be used while retaining effectively random access reads and writes [13]. Then, at most, the individual block needs to be decompressed or recompressed for a single read or write. Similarly, a sequential compression method can be used on the matrix information by compressing the data twice, once for forward iteration and once for backwards iteration.

2.2.2 Single and Mixed Precision Floating Point Numbers

The most obvious compression of floating point data is using single precision representation instead of double precision representation. While it only has a compression rate of 1:2, it allows the compression and decompression of values using at most 1 extra hardware operation. Additionally, it provides the same data access properties as the double precision version. For the matrix values, single precision representation is lossless in the test problem, since each matrix value is an integer. However, for the vector values, using single precision floats resulted in a significant increase of Conjugate Gradient iterations due to the loss of precision. So, by making only select vectors single precision, a compromise can be found where vectors that need high precision can keep that precision and vectors that do not need as much precision can get improved performance.

2.2.3 1-bit Compression

To provide an estimated upper bound for improvements in performance from matrix value compression, 1-bit compression was devised. This scheme uses the fact that the matrix values in the test matrix are all either -1 or 26. Note that as implemented, this scheme can compress a limited number of matrices. However, certain compression schemes that modify the compression based on the data being compressed, such as Huffman coding described in Section 2.2.7, can achieve the same compression for the test matrix. Note that the upper bound provided for 1-bit compression is only an upper bound for the particular pair of vector and index compressions that 1-bit compression was used with. The importance of compressing multiple structures, as described in Section 2.2.9, is shown using 1-bit compression.

2.2.4 Squeeze (SZ) Compression

Squeeze (SZ) compression is a group of compression strategies based on using curve fitting and can be used for both integers and floating point values. The compression strategy referred to as SZ compression in this paper deviates from the original description by using a generalization of the core approach of the

Uncompressed	$v_i \leftarrow$ original ith value
Neighbor	$v_i \leftarrow v_{i-1}$
Linear	$v_i \leftarrow 2v_{i-1} - v_{i-2}$
Quadratic	$v_i \leftarrow 3v_{i-1} - 3v_{i-2} + v_{i-3}$
Neighbor's Neighbor	$v_i \leftarrow v_{i-2}$
Last Uncompressed	$v_i \leftarrow$ last value stored uncompressed
Increment	$v_i \leftarrow v_{i-1} + 1$

Figure 4: Prediction functions used in SZ compression.

original implementation of SZ compression [3]. SZ compression allows for string bounds to be placed on the compression error.

The compressed data is stored in two arrays, one storing the predictor each value is compressed with and the other storing values that could not be predicted within tolerance. To compress each value, the error between the prediction made by each predictor is compared. If the smallest error is within the user supplied tolerance, the associated predictor is stored. Otherwise, the value is appended to the list of uncompressed values and the predictor is stored as uncompressed. Because only the compressed value is available when decompressing, those values are used during compression when computing the value produced by each predictor. This allows error requirements to be met. The compression rate is

$$\frac{ps + \lceil \log_2(n) \rceil}{s}$$

where s be the number of bits used by an uncompressed value, p be the percent of values that are compressed, and n be the number of predictors available. Note that due to the granularity of the matrix values and indices, bounding the error to be less than one results in an effective error bound of 0. Thus, when compressing those data structure, only an error bound of 0 is used.

The predictors available are selected based on the nature of the data being compressed. Figure 4 shows all predictor functions used. For compressing vector values, the Neighbor, Linear and Quadratic predictors were used. Because the vector values represent a value at each grid point, these predictors attempted to capture smooth changes and relations in the data. The matrix indices were compressed using only the increment compression mode, since approximately two thirds of the indices fit that pattern. The matrix values were compressed with a few different combinations of predictors. These combinations were Neighbor alone, then Neighbor and Neighbor's Neighbor. These predictors were chosen to find the best way to compress a series of -1's with occasional 26's.

2.2.5 ZFP Compression

ZFP compression is a lossy floating point compression scheme designed for spatial correlated data [13]. ZFP compression is designed to take advantage of spatial relations for data up to 4 dimensions. Note that the matrix values were

	Compression Rate
$\text{gamma}(1) = 1_2$	1:32
$\text{gamma}(2) = 0\,10_2$	3:32
$\text{gamma}(3) = 0\,11_2$	3:32
$\text{gamma}(4) = 00\,100_2$	5:32
$\text{gamma}(5) = 00\,101_2$	5:32
$\text{gamma}(6) = 00\,110_2$	5:32
$\text{gamma}(7) = 00\,111_2$	5:32
$\text{gamma}(8) = 000\,1000_2$	7:32
$\text{gamma}(64) = 000000\,1000000_2$	13:32
$\text{gamma}(256) = 00000000\,100000000_2$	17:32
$\text{gamma}(1024) = 0000000000\,10000000000_2$	21:32

Figure 5: Select examples of Elias Gamma Coding.

compressed with ZFP, even though there is no spatial relation between points. Because the vectors represent points in 3 dimensions, 1- and 3- dimensional compression was tried. The matrix values were only compressed with 1 dimension. ZFP compresses its values by grouping the data into blocks of 4^d elements, where d is the number of dimensions compressing with [13]. When random access is required, each block is compressed at a fixed size to allow access to arbitrary blocks. ZFP was implemented using the existing C++ library. Both the high-level and low-level interfaces were tried for the vector compression.

2.2.6 Elias Gamma Coding and Delta Coding

Elias Gamma and Delta codings are a pair of similar compression methods that are designed to compress positive integers by not storing extra leading 0's [5]. Because these schemes are better at compressing smaller numbers, the matrix indices were stored as the offset from the preceding value. Then, because these codings are only able to compress positive integers, the indices of each row must be sorted in acceding order. Finally, the first index in each row is stored as the offset from -1, to ensure an index of 0 is properly encoded.

To encode an integer n with Gamma coding, let $N = \lfloor \log_2(n) \rfloor + 1$ be the number of bits needed to store n. Then, n is represented by $N-1$ zeros followed by the N bits of n [5]. Thus, n can be stored with only $2N-1$ bits. For small values of N this is highly affected, reaching compression ratios of up to 1:32. See Figure 5 for examples of gamma coding.

Delta coding is like Gamma coding, except instead of preceding the number

	Compression Rate
$\text{delta}(1) = 1_2$	1:32
$\text{delta}(2) = 010\,0_2$	4:32
$\text{delta}(3) = 010\,1_2$	4:32
$\text{delta}(4) = 011\,00_2$	5:32
$\text{delta}(5) = 011\,01_2$	5:32
$\text{delta}(6) = 011\,10_2$	5:32
$\text{delta}(7) = 011\,11_2$	5:32
$\text{delta}(8) = 00100\,000_2$	8:32
$\text{delta}(64) = 00111\,000000_2$	11:32
$\text{delta}(256) = 0001001\,00000000_2$	15:32
$\text{delta}(1024) = 0001011\,0000000000_2$	17:32

Figure 6: Select examples of Elias Delta Coding.

with $N-1$ 0's, the number is preceded by $gamma(N)$ and only the last $N-1$ bits are stored. So, n can be stored with only $N+2\lfloor\log_2(N)\rfloor$ bits. Figure 6 contains examples of delta coding. Note that delta coding provides better compression for large numbers, but worse compression for certain smaller numbers. Additionally, because decoding a delta encoded value requires decoding a gamma encoded value, decoding a delta coded value is more expensive than decoding a gamma coded value.

2.2.7 Huffman Coding

Huffman coding is an optimal prefix code usable for lossless compression [8]. A prefix code is a coding where each representable value is assigned a unique coding such that no code is the beginning of another code. However, Huffman coding does not take advantage of local patterns in the data, just the overall frequencies of each value. Additionally, Huffman coding can only be decoded sequentially, due to the variable length of storage for each value. So, while it can compress matrix values and indices, it is unable to meet the requirements to compress vector values. Note that the Huffman coding of the matrix values in the test problem is equivalent to the 1-bit coding described in Section 2.2.3. Thus, only matrix indices were tested with Huffman coding.

2.2.8 Opcode Compression

Opcode compression is based on the index compression used in Compressed Column Index (CCI) matrices [11]. Note that this integer compression is never

Opcode	Length
0	4 bits
100	5 bits
110	15 bits
101	20 bits
111	26 bits

Table 1: CCI format opcodes [11].

given its own name in the original description and so is referred to as opcode compression in this paper. Opcode compression is inspired by CPU instruction encodings which are separated into an "opcode" portion and a data portion (hence the name). To read each value, the first few bits are read to determine the number of bits used for the data portion, which stores the encoded value. Like Gamma and Delta coding, opcode compression reduces the number of leading 0's stored, and similarly is utilized by encoding the difference from the preceding index. If some opcodes are used significantly, that opcode can be shorted to save bits. This shortened opcode can be handled in a lookup table by placing the opcode's information at every location that begins with the opcode. For example, if 0, 10 and 11 are the possible opcodes, then the information for opcode 0 is located at the indices of 00 and 01.

The description of CCI matrix format uses a fixed decode table. However, when using a lookup table, using custom decode tables to adjust the compression for the specific matrix's sparsity pattern will not have a significant performance penalty to decoding. Table 1 shows the opcodes used for CCI format.

2.2.9 Combined Compression Strategies

In addition to compressing a single data structure at a time, compression strategies which compress multiple data structures were tried. This provided the opportunity to achieve an overall reduction in data that could not be achieved by compressing a single data structure. Additionally, as discussed in Section 2.1 and as shown in Section 4, compressing matrix values alone cannot provide performance improvement.

3 Performance Models

To help understand the requirements for an improvement in overall performance, models were constructed to estimate the amount of time spent fetching and decoding information. Two models were constructed, an analytical model that did not consider processor level parallelism and a simulation-based model that considered certain processor level parallelism. Both models are based on the sparse matrix-vector product, but, due to the similarity of the data access for the symmetric Gauss-Seidel step, should provide an estimate on both kernels.

L1 Cache Latency	4-5 cycles
L2 Cache Latency	12 cycles
L3 Cache Latency	38 cycles
Main Memory Latency	38 cycles + 58 ns
Clock Rate	2.2GHz

Table 2: Estimate cluster performance [9].

Additionally, the models only provide estimates for rows with 27 elements, because there are $O(n^3)$ of those rows and only $O(n^2)$ of other rows where the matrix has $O(n^3)$ rows. Finally, the models assume that the compression does not reduce the rate of convergence. The models were analyzed using the memory access latencies of the head node of the testing cluster. These latencies are shown in Table 2. The models were primarily used to find the minimum compression performance to outperform the baseline implementation. The following variables represent the relevant compression characteristics in this section

$$\begin{aligned}
\text{vectDecode} &= \text{time to decode one vector value} \\
\text{vectEncode} &= \text{time to encode one vector value} \\
\text{vectBytes} &= \text{the number of bytes per vector value} \\
\text{matIndDecode} &= \text{time to decode one matrix index} \\
\text{matIndBytes} &= \text{the number of bytes per matrix index} \\
\text{matValDecode} &= \text{time to decode one matrix value} \\
\text{matValBytes} &= \text{the number of bytes per matrix value}
\end{aligned}$$

Both models indicate that vector decoding must be incredibly efficient to see overall performance improvement, while matrix decoding can be less efficient. This implies matrix compression has greater potential for overall performance improvement. However, both models make significant assumptions and simplifications that limit the accuracy of these results. Additionally, the models provide slightly different results. For example, the analytical model indicates vector encoding time is negligible while the simulation model puts vector encoding time half the factor of vector decoding.

3.1 Analytical Model

The analytical model is a set of equations that computes the amount of time spent serially fetching and decoding the information. This model is implemented

using the following system of equations

$$
\begin{aligned}
&27 \cdot \text{vectDecode} + \text{vectEncode} + 18 \cdot \text{L1Time} \\
&+ \left(\frac{64 - \text{vectBytes}}{64} \cdot 9 \cdot \text{L1Time} + \frac{\text{vectBytes}}{64} \cdot (6 \cdot \text{L2Time} + 3 \cdot \text{RAMTime}) \right) \\
&+27 \cdot \text{matIndDecode} \\
&+27 \cdot \left(\frac{\text{matIndBytes}}{64} \cdot \text{RAMTime} + \frac{64 - \text{matIndBytes}}{64} \cdot \text{L1Time} \right) \\
&+27 \cdot \text{matValDecode} \\
&+27 \cdot \left(\frac{\text{matValBytes}}{64} \cdot \text{RAMTime} + \frac{64 - \text{matValBytes}}{64} \cdot \text{L1Time} \right)
\end{aligned}
$$

where

$$
\begin{aligned}
\text{L1Time} &= \text{the access latency for L1 cache} \\
\text{L2Time} &= \text{the access latency for L2 cache} \\
\text{RAMTime} &= \text{the access latency for main memory.}
\end{aligned}
$$

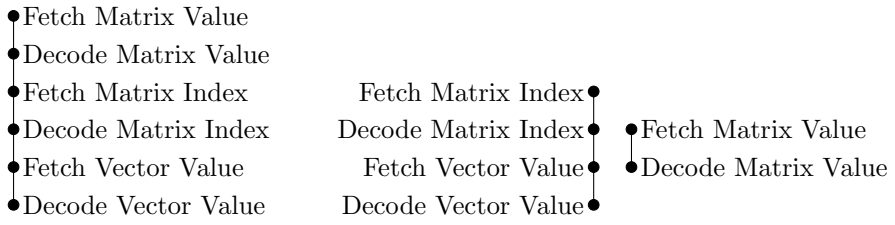
This model utilizes a few facts and assumptions. Firstly, the number of bytes per value divided by 64 provides the percent of values that will require fetching a cache line from main memory or higher caches, while 1 minus this value is the percent of values that will be able to only need to access L1 cache. Secondly, due to the matrix sparsity pattern, two thirds of vector readers will always be in L1 cache and, assuming there are 96^3 rows per process, two thirds of the remaining values will be in L2 cache. The model was only studied using the performance characteristics shown in Table 2. This model was used to create approximate bounds that need to be met to outperform the baseline implementation. The model can be simplified by letting

$$
\begin{aligned}
\text{decode} &= \text{vectDecode} + \text{matValDecode} + \text{matIndDecode} \\
\text{matBytes} &= \text{matValBytes} + \text{matIndBytes}.
\end{aligned}
$$

Then, the performance bounds are, approximately,

$$
\begin{aligned}
\text{vectEncode} &< 878.513 \\
\text{decode} &< 32.5375 - 0.037037 \cdot \text{vectEncode} \\
\text{matBytes} &< 12.9664 - 0.398506 \cdot \text{decode} - 0.0147595 \cdot \text{vectEncode} \\
\text{vectBytes} &< 107.34 - 3.29897 \cdot \text{decode} - 0.122184 \cdot \text{vectEncode} \\
&\quad - 8.27835 \cdot \text{matBytes}
\end{aligned}
$$

where all encode and decode times are in clocks. Note that the upper bound on the number of bytes per vector value is reduced by 3.29897 per 1 clock



(a) Serial Model. (b) Parallel Model.

Figure 7: Comparison of the data dependency graphs used by each model, where lower nodes are dependent on higher nodes.

increase in decode time. This indicates that only highly effective compression techniques will be effective for vector values. Matrix compression on the other hand, appears to be able to achieve a performance improvement with a slower decompression than required by the vector values.

3.2 Simulation Based Model

The parallel model attempts to estimate the performance required to outperform the baseline implementation while considering processor level parallelism. Figure 7 shows the dependencies used by each model. Note that this model assumes that the compiler and processor can fully parallelize any operations without data dependencies, while, the instructions must be at least read serially [6]. Additionally, the model assumes that the bytes of each compressed value is constant, which is not true for most matrix compression and for some vector compression. Also, the model was not used to analyze compressing multiple data structures simultaneously due to the difficulty of analyzing the resulting seven-dimension region. Lastly, the model was used with integral values for the bytes, decode times and encode times. The model then takes the same compression properties as the first model and computes the time to fetch and decode the values and encode the result value over 10 matrix rows with 27 entries per row. Appendix A contains source code for this model.

The bounds on outperforming the baseline implementation were hard to determine for the simulation model due to the nature of the model as a sum of maximizations. However, some boundaries were computed. Table 3 shows the restrictions when compressing only a single data structure. The matrix index and value columns contain the maximum time to decode a value, in clocks. The vector column contains equations that restrict the decode and encode time, with values in clocks. Note that the vector limits are not the only way to compute the restriction. These compression bounds appear to be significantly related to the frequency at which multiple values are fetched from main memory simultaneously.

Bytes	Matrix Index Decode	Matrix Value Decode	Vector Encode and Decode
1	66	41	$4.75 \geq 1.75 \cdot \text{decode} + \text{encode}$
2	51	14	$4.75 \geq 1.75 \cdot \text{decode} + \text{encode}$
3	66	40	$2 \geq 2 \cdot \text{decode} + \text{encode}$
4	0	0	$0 = \text{decode} = \text{encode}$
5	-	37	$4.75 \geq 1.75 \cdot \text{decode} + \text{encode}$
6	-	9	Not Possible
7	-	35	$0 = \text{decode} = \text{encode}$
8	-	0	$0 = \text{decode} = \text{encode}$

Table 3: Maximum decode times in clocks for single data structure compression, according to the simulation based model.

4 Test Results

Tables 4, 5, and 6 show the compression results for compressing just the vector values, matrix values, and matrix indices, respectively. These tables, and all following tables of test results, contain the HPCG rating; the GFLOP rating with convergence overhead; the number of iterations needed for convergence; and, where computed, the compression rate based on the number of cache lines fetched, which may be different then the memory allocated. Note that some compression strategies had multiple variations that were tested. Sections 4.1, 4.2, and 4.3 provide details on the compression of each specific data structure.

Note that for comparing times, ten runs of the baseline implementation with the standard test settings had a range of 0.6128 and a standard deviation of 0.1846 for the HPCG rating. For the effective GFLOP/s, there was a range of 0.4629 and a standard deviation of 0.1504. Thus, when comparing similar values, it should be noted that the values likely vary by a few tenths of a GFLOP/s between runs.

Next, because compressing a single data structure failed to improve performance, both matrix data structures were compressed. SZ and single precision compression where tried for the matrix values and using SZ, gamma and delta compression for the matrix indices. Table 7 shows the results of these combined schemes. The combined scheme with the best performance used SZ compression for both values and indices. The only other approach that outperformed the baseline implementation used 32-bit compression for the values and gamma compression for the indices.

Finally, vector compression was combined with the successful combined matrix compression. Only the best few versions of mixed precision vector compression were used. Table 8 shows the results for these compression strategies. The first column indicates which vectors were stored in 32bit; the rest of the columns correspond to their counterparts in Table 7. Note that vector compression improved the performance of the SZ compressed matrices but reduced the performance of the gamma compressed matrices. So, the best implementation

Compression	HPCG Rating	GFLOPs Rating	Iterations	Compression Rate
Baseline	15.3654	15.7394	50	1:1
Single Precision	7.023 01	7.101 26	115	1:2
Mixed Precision				
$\vec{d}$	5.184 76	5.226 74	150	11:12
$\vec{b}, \vec{x}$	15.3701	15.7503	50	5:6
$\vec{b}, \vec{x}, \mathbf{A}\vec{d}$	15.0428	15.4048	51	3:4
$\vec{b}, \vec{x}, \vec{d}$	5.208 32	5.250 84	150	3:4
$\vec{b}, \vec{x}, \vec{d}, \mathbf{A}\vec{d}$	5.232 08	5.275 03	150	2:3
$\vec{b}, \vec{x}, \vec{d}, \mathbf{A}\vec{d}, \vec{z}$	6.958 94	7.035 99	115	7:12
$\vec{b}, \vec{x}, \vec{d}, \vec{z}$	6.913 43	6.988 23	115	2:3
$\vec{b}, \vec{x}, \vec{z}$	12.2146	12.4516	64	3:4
ZFP - 1d				
16 bits/value	0.690 138	0.690 913	51	1:4
32 bits/value	0.393 97	0.394 225	50	1:2
ZFP - 3d				
11 bits/value	nan	nan	≥ 500	11:64
12 bits/value	2.776 71	2.789 38	53	3:16
14 bits/value	2.794 16	2.806 92	51	7:32
16 bits/value	2.720 78	2.733 11	51	1:4
24 bits/value	2.487 93	2.4982	50	3:8
SZ				
7 values/block	5.891 38	5.947 58	57	8:7
8 values/block	5.787 11	5.841 15	57	1:1 to 2:1
12 values/block	4.985 36	5.025 68	57	2:3 to 4:3
15 values/block	4.515 94	4.548 98	57	8:15 to 16:15
16 values/block	4.535 36	4.568 68	57	1:2 to 3:2
24 values/block	3.677 48	3.699 07	57	1:3 to 4:3
32 values/block	3.1735	3.189 73	57	1:4 to 5:4

4: Results of compressing vector values.

Compression	HPCG Rating	GFLOPs Rating	Iterations	Compression Rate
Baseline	15.3654	15.7394	50	1:1
Single Precision	12.6331	12.8958	50	1:2
1-bit	15.1743	15.5452	50	1:64
SZ				
1 mode	13.5037	13.813	50	$\sim$ 2:11
2 modes	13.8195	14.131	50	$\sim$ 1:7
ZFP				
High Level API	0.817 469	0.818 857	50	1:2
Low Level API	0.960 338	0.962 149	53	1:2

5: Results of compressing matrix values.

Compression	HPCG Rating	GFLOPs Rating	Iterations	Compression Rate
Baseline	15.3654	15.7394	50	1:1
SZ	14.9322	15.2964	50	$\sim$1:2
Elias Gamma	14.6553	15.0062	50	$\sim$1:9
Elias Delta	14.3036	14.6485	51	$\sim$1:8
Huffman				
No First Index				
4-bit window	10.654	10.8548	51	$\sim$1:9
8-bit window	10.8941	11.1043	51	$\sim$1:9
12-bit window	10.8158	11.0251	51	$\sim$1:8
First Index				
4-bit window	10.9359	11.148	51	$\sim$1:9
8-bit window	11.1134	11.3336	51	$\sim$1:9
16-bit window	10.3323	10.5819	51	$\sim$20:11
Op Code				
CCI Format [11]	8.512 57	8.635 78	51	$\sim$3:10
Best Result	8.522 72	8.647 52	51	$\sim$1:6

6: Results of compressing matrix indices.

Compression		HPCG	GFLOPs	
Value	Index	Rating	Rating	Iterations
Baseline		15.3654	15.7394	50
SZ	SZ	18.9702	19.5759	50
SZ	Gamma	13.661	13.9794	51
SZ	Delta	10.9903	11.1961	50
32 bits	SZ	14.1796	14.5156	51
32 bits	Gamma	17.6676	18.1835	51
32 bits	Delta	12.56	12.8181	51

7: Results of combined matrix value and index compression schemes.

Compression					
32-bit Vectors	Value	Index	HPCG Rating	GFLOPs Rating	Iterations
Baseline			15.3654	15.7394	50
None	SZ	SZ	18.9702	19.5759	50
$\vec{b}, \vec{x}$	SZ	SZ	23.967	24.9875	50
$\vec{b}, \vec{x}, \mathbf{A}\vec{d}$	SZ	SZ	27.5974	28.9699	50
None	32 bits	Gamma	17.6676	18.1835	51
$\vec{b}, \vec{x}$	32 bits	Gamma	16.6048	17.0684	50
$\vec{b}, \vec{x}, \mathbf{A}\vec{d}$	32 bits	Gamma	16.5665	17.0294	50

8: Results of combined vector, matrix value and matrix index compression schemes.

for the test problem uses mixed precision vectors with $\vec{b}, \vec{x}$ and $\mathbf{A}\vec{d}$ stored in single precision, and SZ compressed matrix values and indices.

4.1 Vector Compression

As shown in Table 4 and discussed previously, vector compression was not successfully used to improve performance. However, certain combinations of single and double precision vector values were able to perform close to the baseline performance. These mixed precision implementations were able to improve performance when combined with other compressions, as shown in Table 8. Note that the single precision implementation has a 2.3-times increase in iterations to converge over the baseline implementation and that the GFLOPs rating of the single precision implementation is reduced by a factor of approximately 2.19 from the baseline implementation. This hints that, even without increasing the number of Conjugate Gradient iterations, vector compression requires a compression rate better than 1:2 to provide much of an improvement in performance.

The compression schemes used mixed precision vectors are listed by which vectors are stored in single precision; the unlisted vectors are stored in double precision. When only $\vec{b}, \vec{x}$, and optionally $\mathbf{A}\vec{d}$, are stored in single precision, then the code can perform close to the baseline implementation. Note that $\vec{b}$ only contains integers in the test problem, so using single precision does not result in any loss of precision. Similarly, $\vec{x}$ is not used to compute other values. Thus, any error that occurs in $\vec{x}$ will not be propagated into the computation of other values. Additionally, making $\vec{x}$ single precision does not affect the accuracy of the solution. Using single precision for $\vec{x}$ and $\vec{b}$ did not affect the value of $\|\vec{x}_{\text{computed}} - \vec{x}_{\text{exact}}\|_2$. However, making $\vec{x}, \vec{b}$ and $\mathbf{A}\vec{d}$ single precision increased the value from 1235.61 to 1658.75.

ZFP had poor performance when compressing vector information. The high-level array API was used to provide the necessary random access. That API provides an adjustable cache for decoded values. Figure 8 shows the performance of 1d ZFP compression versus the cache size. Note that the two large jumps occur when the values only need to be decoded once per matrix-vector product and when the values can be left permanently decoded. The results in Table 4 use the default cache size of 2 blocks for 1 dimensional compression and $2 \cdot \lceil nz/2 \rceil \cdot \lceil ny/2 \rceil$ blocks for 3-dimensional compression. The array API also allows selecting the compression rate, with a 16-bit granularity for 1-dimensional compression and a 1-bit granularity for 3-dimensional compression [13]. These granularity restrictions and the resulting iterations needed were used to select the tested compression rates.

SZ compression has two main configurable settings, the number of values in each block and the error bound. There were two measures of error that were considered, absolute error and pointwise relative error. The performance was tested with both a single error being bounded, and both errors being bounded. Absolute error is the absolute value of the difference between predicted and actual. The pointwise relative error is the absolute error divided by the actual value. Table 4 contains results for various block sizes with both an absolute

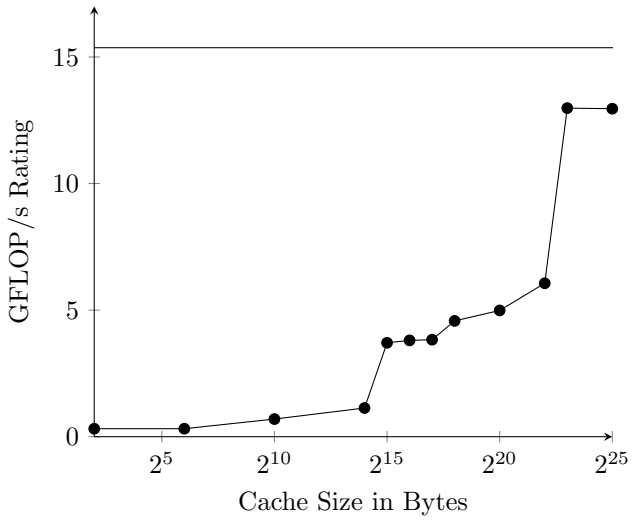


Figure 8: Performance of 1d ZFP compression with 16 bits per value versus the size of the decoded values cache. The horizontal line shows the performance of the baseline implementation.

error bound of 10^{-10} and a pointwise relative error bound of 10^{-10}. Table 9 contains a comparison of various error bounds for a block size of 8 values per block. Note that an absolute bound of 10^{-2} was unable to converge within 500 iterations.

4.2 Matrix Value Compression

Like vector compression, matrix value compression alone was unable to outperform the baseline implementation. As shown in Table 5, 1-bit compression underperformed the baseline implementation, indicating compressing the matrix values alone in unable to improve performance. Both SZ compression and single precision compression were not significantly under the baseline, indicating that they may be usable in conjunction with other compression techniques. On the other hand, ZFP compression performed over a magnitude slower than the baseline, making it unlikely that it could improve performance, even when combined with other techniques.

As mentioned in Section 2.2.4, there were two possible sets of predictors that could be used for matrix value compression. One setup uses only the Neighbor predictor, and the other setup uses both the Neighbor and Neighbor's Neighbor predictors. Both setups performed similarly, with the 2-predictor version performing slightly better. The 1-predictor version was used when combining with other compression methods. Also, because the only matrix values are -1 and 26, all reasonable error bounds provide equivalent compression.

ZFP compression had multiple configurations that could be used. However,

Error Bound	HPCG Rating	GFLOPs Rating	Iterations
10^{-2} relative	3.668 59	3.690 56	69
10^{-6} relative	5.7806	5.834 61	57
10^{-10} relative	5.825 27	8.879 61	57
10^{-14} relative	5.813 57	5.8677	57
10^{-18} relative	5.732 77	5.785 96	57
10^{-2} absolute	NA	NA	≥ 500
10^{-6} absolute	4.518 27	4.551 58	57
10^{-10} absolute	5.140 58	5.183 13	57
10^{-14} absolute	5.643 38	5.694 36	57
10^{-18} absolute	5.816 42	5.870 97	57
10^{-2} absolute and 10^{-10} relative	5.755 38	5.809	57
10^{-10} absolute and 10^{-2} relative	5.225 92	5.270 53	57
10^{-10} absolute and 10^{-10} relative	5.787 11	5.841 15	57

9: Results of compressing vector values with SZ compression using various error bounds.

the possible configurations were not fully experimented with since the matrix values lacks the spatial relations that ZFP is designed for and ZFP performs poorly on the configurations that were tested, for both vector and matrix value compression. First, unlike ZFP vector compression, only the 1-dimensional codec was tested because the data does not have spatial patterns, let alone multiple dimensions of them. Second, due to the simple access pattern for matrix values, both the high-level array API and the low-level API were tried. The compression rate was kept at 32 bits per value. None of the configurations tested were able to provide even mediocre performance, so ZFP compression was not tested further for matrix compression.

4.3 Matrix Index Compression

Matrix index compression was unable to outperform the baseline implementation. Both SZ compression and the Elias codings were able to perform in the ballpark of the baseline implementation. Neither Huffman coding nor Op Code compression were able to perform close to the performance of baseline implementation.

Variations of the Huffman coding implementation were experimented with to find the best set of parameters. The first parameter was whether to compress the first index or to leave it uncompressed. Because Huffman coding needs a codeword for each represented value, compressing the first index will reduce the memory of that value but reduce the efficiency of compressing the rest of the values [8]. Since neither version was obviously better, both were tried. The second parameter was how many bits to view at a time when decoding. Second, decoding was implemented using chaining of lookup tables. Decoding

Opcode	Length
0	1 bits
100	7 bits
110	14 bits
101	20 bits
111	29 bits

10: Opcode set with the best performance.

is implemented using chained lookup tables. So, each lookup checked the next `window_size` bits, which either provided the value and number of bits consumed or a further table to check the next `window_size` bits on [16]. So, a higher window size increases the memory needed, but reduces the total number of lookups to do, providing a tradeoff between the amount of cache space used up and the average time to decode each value.

Although Opcode compression has outperformed uncompressed values in some settings, it was unable to perform well in this code base [11]. Several sets of opcodes were tried, including the original shown in Figure 1. The set of opcodes with the best performance is shown in Figure 10; it was designed for the best performance on the matrices used. The difference in success of the CCI opcodes from previous works likely comes from a difference in the data access patterns, cache sizes or decoding implementation.

4.4 Testing Environment

The compression rates were either analytically computed or computed from the compressed size. Other numbers were provided by HPCG's benchmark results. The HPCG rating is the benchmark's rating, located in the yaml results file as the "GFLOP/s Summary: Total with convergence and optimization phase overhead" field. The GFLOP/s rating can similarly be found in the "GFLOP/s Summary: Total with convergence overhead" field. The iteration count was determined using the content of the "Iteration Count Overhead" field. All results were obtained with a minimum run time of 300 seconds.

The timings presented were obtained when using a per-process problem size of 96^3 matrix rows across 60 processes, as described in Section 2. The cluster used for timings had a 20-core, 2.2 GHz, Intel Xeon E5-2698 v4 head node and an additional five 8-core, 1.7GHz, Intel Xeon E5-2605 nodes. One process was created for each core, with a single OMP thread per process.

The code was implemented using version 3.0.0 of the HPCG benchmark [4]. Many of the implementations with timings listed in this paper can be found at `https://github.com/Collegeville/HPCG-ZFP` [12]. The code was compiled with the OpenMPI cxx wrapper using GCC version 4.8.5. OpenMPI 3.0.2 was used for the compiler wrapper and MPI runtime. The `O3` and `fopenmp` flags were used for compilation, in addition to a selection of warning flags and the `std` flag, as necessary. No `HPCG_OPTS` flags were enabled.

5 Conclusions

Data compression was able to successfully increase the performance of the sparse linear solver in HPCG [4]. The best performance increase came from using SZ compression on the matrix indices and values and using single precision values for select vectors with an increase in the effective GFLOPs of about 84%. However, there are only a few compression strategies that outperform the baseline.

When considering more general matrices, note that the effectiveness of SZ compression is highly dependent on local relationships between compressed values. So, SZ based compression strategies will likely lose performance on many other matrices. However, because the performance of single precision "compression" is unaffected by the values being compressed and that the effectiveness of gamma compression is proportional to the number of significant bits, using 32bit matrix values and gamma compressed matrix indices will likely perform more consistently than the SZ based compression approach and may perform better on some matrices.

5.1 Future Work

There are two general directions in which this work can be extended: different problem setups and different compression methods. The first direction extends this idea of this project to more general situations, including different linear systems and different solvers. The second direction is to provide either better compression for this problem or demonstrate that the effective compression methods cannot be significantly outperformed.

Note that the stencil matrix used by HPCG is very consistent and has a large amount of repetition; this makes it easy to compress. So, experimenting with other matrices would provide a more general idea of the performance, and should be done before applying this work to production solvers. The SuiteSparse Matrix Collection provides linear systems that could be used for this investigation [2]. However, note that the HPCG implementation makes some assumptions about the matrix in the problem setup, so some of the setup sections would need to be rewritten [4].

Another aspect that could be further experimented with is the solver used. Other solvers and preconditioner likely have different data access patterns than the solver used in HPCG. These differences may change the available compression strategies, have different precision requirements, or have different ratios of memory fetches to arithmetic. So, different solvers may have better optimal compression strategies. Additionally, GPU accelerated solvers may performance differently due to the differences between GPU and CPU execution and performance.

The last area this work could be extended is different compression methods and variants of previously used compression methods. One such topic would be looking at using different error tolerances for different vectors, like what is done with single precision compression, for SZ or another compression. Another

possible variation would be to compress additional data structures, such as the matrix diagonals. Finally, other compression strategies could be tried.

6 References

[1] Alfredo Buttari, Jack Dongarra, Jakub Kurzak, Piotr Luszczek, and Stanimir Tomov. Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy. *ACM Trans. Math. Softw.*, 34(4):17:1–17:22, July 2008.

[2] Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[3] S. Di and F. Cappello. Fast Error-Bounded Lossy HPC Data Compression with SZ. In *2016 IEEE International Parallel and Distributed Processing Symposium*, pages 730–739, May 2016.

[4] Jack Dongarra, Michael Heroux, and Piotr Luszczek. HPCG Benchmark: a New Metric for Ranking High Performance Computing Systems. Technical Report UT-EECS-15-736, Electrical Engineering and Computer Science Department, Knoxville, Tennessee, November 2015.

[5] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, sixth edition, 2017.

[7] J. D. Hogg and J. A. Scott. A fast and robust mixed-precision solver for the solution of sparse symmetric linear systems. *ACM Trans. Math. Softw.*, 37(2):17:1–17:24, April 2010.

[8] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[9] Intel Bradwell. `https://www.7-cpu.com/cpu/Broadwell.html`. Accessed: 2018-11-06.

[10] David R. Kincaid and E. Ward Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Pure and applied undergraduate texts. American Mathematical Society, 2002.

[11] Orion Sky Lawlor. In-memory data compression for sparse matrices. In *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '13, pages 6:1–6:6, New York, NY, USA, 2013. ACM.

[12] Neil Lindquist. Code for Reducing Memory Access Latencies using Data Compression in Sparse, Iterative Linear Solvers . `https://github.com/Collegeville/-ZFP`.

[13] P. Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683, Dec 2014.

[14] J. Nearing. *Mathematical Tools for Physics*. Dover books on mathematics. Dover Publications, 2010.

[15] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[16] Michael Schindler. Practical Huffman coding, Oct 1998. Accessed: 2019-02-07.

[17] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

A Performance Model Source Code

Below is the implementation of the simulation-based performance model. The model was implemented in Common Lisp and used with Steel Bank Common Lisp version 1.4.0.

```
;;; Global Variables

; Cluster properties
(defparameter *l1-time* 5)
(defparameter *l2-time* 12)
(defparameter *main-mem-time* 1656/10)

; Model parameters
(defparameter *rows-to-check* 128)

; Default compression settings
(defparameter *bytes-per-mat-ind* 4)
(defparameter *bytes-per-mat-val* 8)
(defparameter *bytes-per-vect* 8)
(defparameter *inds-decode-time* 0)
(defparameter *vals-decode-time* 0)
(defparameter *vect-decode-time* 0)
(defparameter *vect-encode-time* 0)


```

```
21 ;;; Model Implementation
22
23 (defmethod fetch ((obj (eql :mat-inds)) (i integer))
24   "Computes the cost of fetching the ith matrix index"
25   (if (/= (floor (* (1- i) *bytes-per-mat-ind*) 64)
26           (floor (* i *bytes-per-mat-ind*) 64))
27     *main-mem-time*
28     *l1-time*))
29
30 (defmethod fetch ((obj (eql :mat-vals)) (i integer))
31   "Computes the cost of fetching the ith matrix value"
32   (if (/= (floor (* (1- i) *bytes-per-mat-val*) 64)
33           (floor (* i *bytes-per-mat-val*) 64))
34     *main-mem-time*
35     *l1-time*))
36
37 (defmethod fetch ((obj (eql :vect)) (i integer))
38   "Computes the code of fetching the ith vector value"
39   (cond
40     ; 2/3rds of values were used by the previous index
41     ((/= (mod i 3) 2) *l1-time*)
42     ; 2/9ths of values were used by y-1
43     ((< (/ i 3) 6) *l2-time*)
44     ; 1/9th of values are being used for the first time
45     (t (if (/= (floor (* (1- (/ i 27)) *bytes-per-vect*)
46                       64)
47                (floor (* (/ i 27) *bytes-per-vect*)
48                       64))
49         *main-mem-time*
50         *l1-time*))))
51
52
53 (defun 1-row ()
54   "Computes the average cost to load a row.
55    *rows-to-check* provides the number of rows to use"
56   (+ (/ (loop
57           :for i :from 0 :below (* *rows-to-check* 27)
58           :for inds-fetch-time = (fetch :mat-inds i)
59           :for vals-fetch-time = (fetch :mat-vals i)
60           :for vect-fetch-time = (fetch :vect i)
61           :for total-vals-time = (+ vals-fetch-time
62                                     *vals-decode-time*)
63           :for total-vect-time = (+ inds-fetch-time
64                                     *inds-decode-time*
65                                     vect-fetch-time
66                                     *vect-decode-time*)
```

```
67              :summing (min total-vals-time total-vect-time))
68            *rows-to-check*)
69         *vect-encode-time*))
70
71
72  (defun 1-row-with-props (ind-size val-size vect-size
73                           ind-decode val-decode
74                           vect-decode vect-encode)
75    "Like 1-row, but sets the compression properties"
76    (let ((*bytes-per-mat-ind* ind-size)
77          (*bytes-per-mat-val* val-size)
78          (*bytes-per-vect* vect-size)
79          (*inds-decode-time* ind-decode)
80          (*vals-decode-time* val-decode)
81          (*vect-decode-time* vect-decode)
82          (*vect-encode-time* vect-encode))
83      (1-row)))
```