4-2013

Modeling a non-uniform memory access architecture for optimizing conjugate gradient performance with sparse matrices

Jacob Hemstad
College of Saint Benedict/Saint John's University

Modeling a Non-Uniform Memory Access Architecture for Optimizing Conjugate Gradient Performance with Sparse Matrices

AN HONORS THESIS

College of St. Benedict/St. John's University

In Partial Fulfillment

of the Requirements for Distinction

in the Department of Physics

by

Jacob Hemstad

April, 2013

Approved by | Date of Signature

Michael Heroux

Professor of Computer Science

Thesis Adviser

Lynn Zeigler

Professor of Computer Science

Dean Langley

Professor of Physics

Chair, Department of Physics

Thomas Kirkman

Professor of Physics

Tony Cunningham

Director, Honors Thesis Program

Modeling a Non-Uniform Memory Access Architecture for Optimizing Conjugate Gradient Performance with Sparse Matrices

Jacob Hemstad in colloboration with Brandon Hildreth

December 10, 2013

Abstract

The last ten years have seen the rise of a new parallel computing paradigm with diverse hardware architectures and software interfaces. One of the common architectures, known as 'non-uniform memory access' (NUMA), structures parallel computers so cores can access certain parts of memory faster than others. In our work, we sought to model a specific NUMA machine and use that model to inform optimizations for performing the Conjugate Gradient method. We used the model to come up with a segmented design that puts data that a core needs in memory where it can access it fast. Our segmented solution proved to be effective over the control with a maximum speed-up of 11.1x faster.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Parallel Paradigm Shift

Moore's Law promises that transistor density doubles approximately every two years. In the past, the additional transistors were used in part for frequency scaling, or the ramping of chip frequency, and Moore's Law provided a biennial doubling in chip performance. However, in the last 10 years frequency scaling has ceased to be a viable option, as can be seen in Figure 1 which shows how Intel's chip manufacturing trends over the last thirty years has leveled off. The trade off of increasing frequency is increased power consumption as seen in equation 1, which gives the power consumption of a CPU where C is the capacitance being switched per cycle, V is voltage, and F is the processor frequency.

$$P = C * V^2 * F \tag{1}$$

From Figure 1, one can see the balance between frequency and power consumption stabilized around 3.5GHz in 2004 with Intel's cancellation of their Tejas chip slated to operate at 7GHz[4]. With the end of frequency scaling, the additional transistors provided by Moore's Law are no longer needed for increased frequencies and can instead by used for additional cores on a single die. The availability of transistors for additional cores marked the industry wide shift to parallel computing and multi-core processors. Parallel computing had been around for years at this point in the field of high performance computing, but it did not penetrate into the marketplace until the fall off of frequency scaling.

1.2 Parallel Computing

Parallel computing is a mammoth field with volumes and volumes of literature beyond the scope of this paper. We offer only the most cursory of overviews and will illuminate only the details that are relevant to our work.

At the most basic level, parallel computing is a form of computation in which calculations are carried out simultaneously. Large problems are divided into independent pieces and then solved concurrently. There are many different levels of parallel computing from bit-level to task level. Likewise there are many different

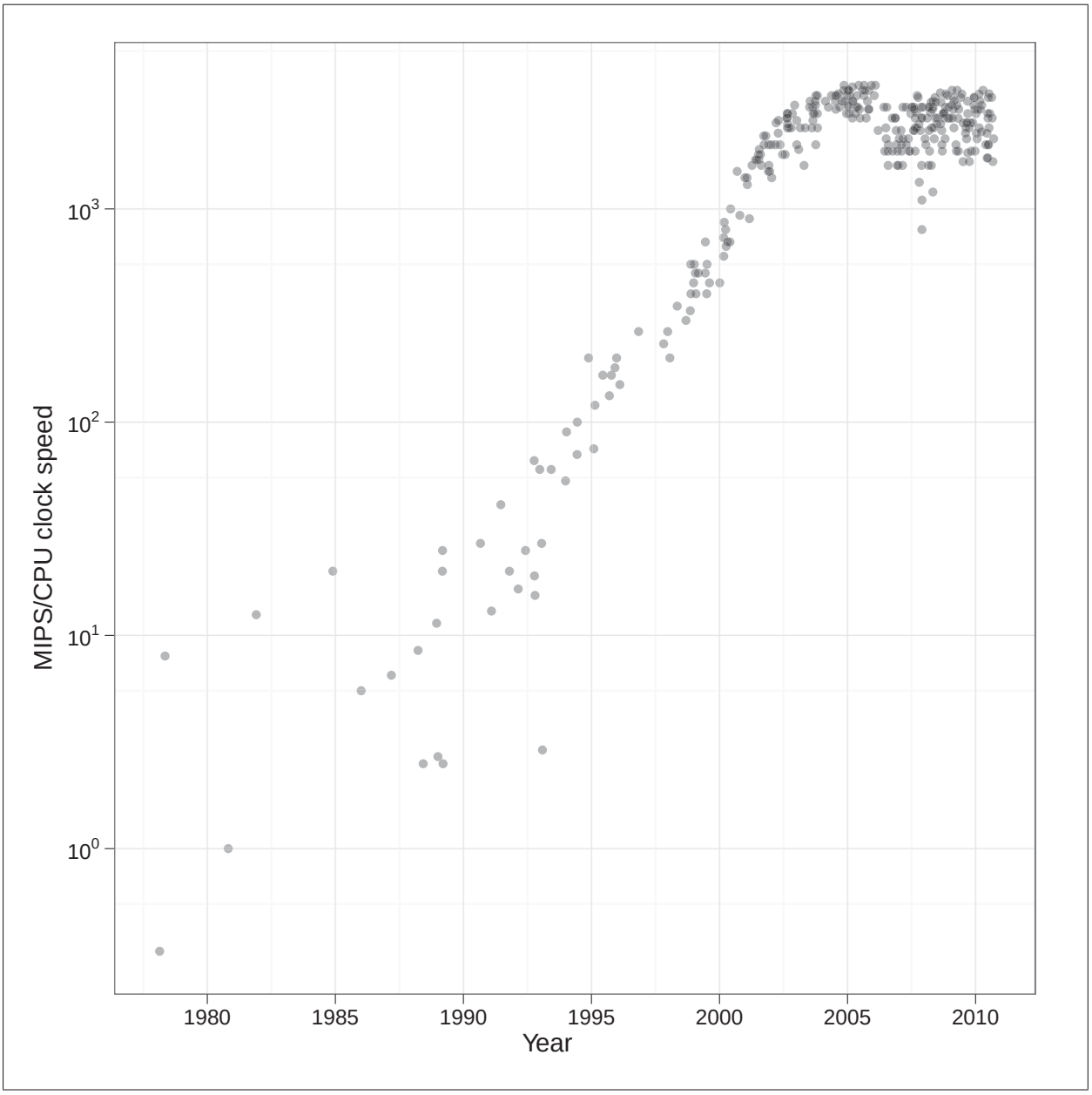

Figure 1: This graph shows the trend of MIPS/clock frequency for Intel processors over the last 30 years.[6]

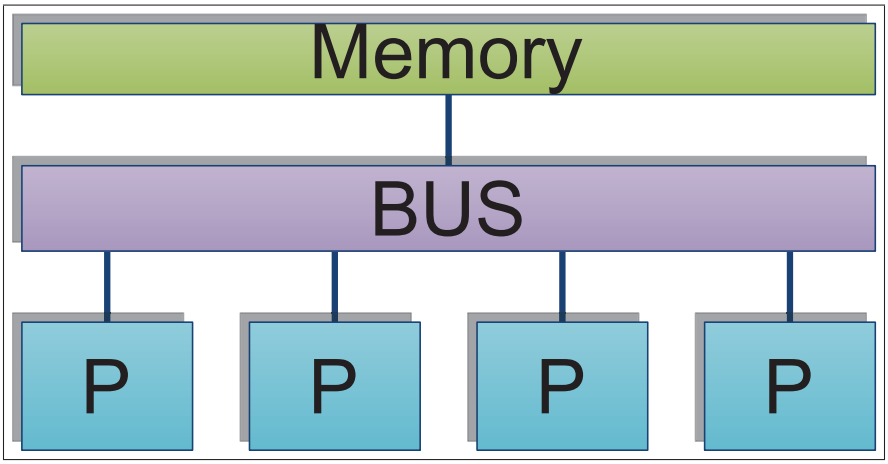

Figure 2: Diagram of a basic symmetric multiprocessing architecture. Four processors are connected to the shared memory pool via a system bus[2].

forms of parallel computers from single multi-core chips to giant super computers like Blue Gene. Writing programs for parallel execution introduces numerous difficulties not faced in sequential programming such as race conditions, mutual exclusion, and synchronization.

1.2.1 Symmetric Multiprocessing

One of the most common smaller scale parallel architectures is *symmetric multiprocessing* (SMP). Most consumer multicore chips are implementations of SMP. In this architecture scheme, two or more identical processors (or cores in multicore chips) are connected to shared memory via a common bus, as seen in Figure 2 where four cores share a bus to access shared memory. Usually in SMP, each core has private access to its own fast cache memory and universal access to the shared DRAM. Read/write operations to the shared memory are serialized and the system bus quickly becomes a major bottleneck as more processors are added and compete for DRAM access. This limits the number of cores feasible on a SMP system and one does not usually see more than 16 cores sharing a common bus.

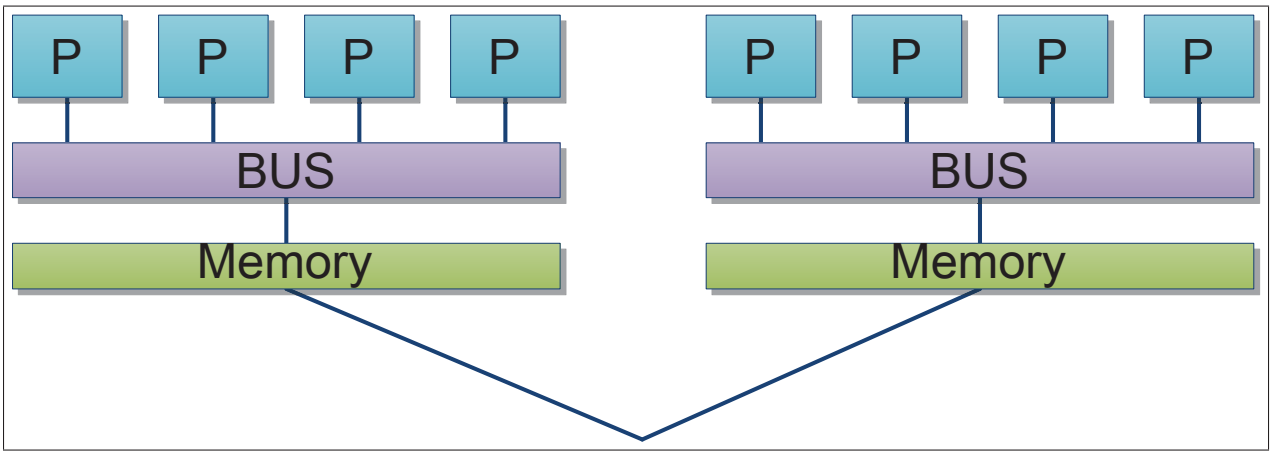

Figure 3: Diagram of a basic non-uniform memory access architecture. Two sets of four cores are connected to nearby memory via a system bus and the two NUMA regions are then connected by a high speed interconnect.[1]

1.2.2 Non-Uniform Memory Access

SMP is a *Uniform Memory Access* (UMA) architecture because all of the cores have equally fast access to all of the shared memory and are equally affected by the bandwidth bottleneck of the shared bus. *Non-Uniform Memory Access* (NUMA) architectures were devised to overcome the performance bottleneck in larger UMA environments by providing a small number of cores with a separate bus to its own section of DRAM. These cores and their direct access DRAM are called a ***NUMA region*** and regions are connected by a high speed interconnect providing all cores with access to the entire memory space. Memory directly connected to a core is called *nearby* and the rest of the memory space is *distant*. A simple model of this can be seen in Figure 3.

The benefit of this design are the separate buses in each NUMA region. When cores access memory in their region, the buses of other regions are unaffected. Assuming cores only access memory nearby, the bandwidth bottleneck of traditional SMP implementations is greatly reduced. However, the trade off of this design is that distant memory reads are quite slow and can slow down the entire system by adding to the bandwidth demands.

1.3 C++ and OpenMP

There are many programming interfaces that provide for parallel programming from Intel's Threading Building Blocks to NVIDIA's CUDA. The interface we use in our work is OpenMP—a shared memory multiprocessing API that supports C, C++, and Fortran on most architectures and operating systems. OpenMP was chosen because it is relatively easy to use with a highly flexible interface. It is an implementation of multithreading, where a master thread executes during the serial portions of code and when a parallel region is encountered a number of threads are spawned and the parallel computation divided among them. Each thread has an associated thread id number which can be accessed with the OpenMP function *omp_get_thread_num*(). Additionally our code is done in C++ as the existing work had already been completed in C++.

Parallel regions are established in OpenMP with preprocessor directives, *#pragma* statements in C++, that spawn the threads. There are several such directives, but relevant to our work are the *omp parallel* and *omp parallel for* directives. They are used to mark a region of code to be ran in parallel among a thread team generated by OpenMP. Their use can be seen below:

```
#pragma omp parallel for{
for(int i = 0; i < n; ++i){
 y[i]=x[i]+z[i];
}
```

The *omp parallel for* directive above creates the appropriate number of threads (often a fixed number specified ahead of time) and splits the iterations of the *for* loop among the threads. For example, if n=100 and four threads are in use, then thread 0 would be responsible for summing x[0] and z[0] through x[24] and z[24]. The *omp parallel for* directive is specialized for the automatic parallelization of a *for* loop, the *omp parallel* directive is much more general in that all the code within its specified region will be run by each thread.

1.3.1 Compiler Environment

The compiler for this work was GNU's *g++* version 4.5.1 which supports OpenMP 3.0. The compilation flags used were *-O3 -funroll-all-loops -malign-double -fopenmp -DWALL*, where *-O3 -funroll-all-loops -malign-*

double are optimization flags, *-fopenmp* enables the use of OpenMP, and *-DWALL* specifies the use of wall time.

2 The HPCCG Mini-application

2.1 Finite Difference Method

Our goal is to optimize a linear algebra solution to a system of differential equations—namely the conjugate gradient method for solving the linear system $\vec{y} = A\vec{x}$, where A is a known matrix, $\vec{y}$ is a known vector, and $\vec{x}$ is an unknown vector . Specifically, we are optimizing Sandia National Laboratories' *HPCCG* mini-application—one of many benchmarking mini-apps included in the Mantevo project[5]. *HPCCG* benchmarks the performance of numerically solving, with the conjugate gradient method, differential equations through the finite difference/element/volume method. An example of the finite difference method in one dimension is presented in the context of the steady-state heat equation, given by LaPlace's equation.

On the interval $[0, 1]$ we have $u''(x) = 0$ with boundary conditions $u(0) = a$ and $u(1) = b$. The solution to this equation, $u(x)$, will describe the distribution of heat at a point x on a wire of length 1 where each of its ends is held at a constant temperature a and b, respectively. From the definition of the derivative of a function u, we know

$$u'(x+h) = \lim_{h \to 0} \frac{u(x+h) - u(x)}{h} \tag{2}$$

for $h \approx 0$ we can approximate the derivative as $u'(x+h) \approx \dfrac{u(x+h) - u(x)}{h}$, which implies $u'(x-h) \approx \dfrac{u(x) - u(x-h)}{h}$. We can use the same definition of the derivative to find the second derivative

$$u''(x) = \lim_{h \to 0} \frac{u'(x+h) - u'(x)}{h} \tag{3}$$

combined with the definitions of the first derivative from above we find

$$u''(x) \approx \frac{u(x+h) - 2u(x) + u(x-h)}{h^2} \tag{4}$$

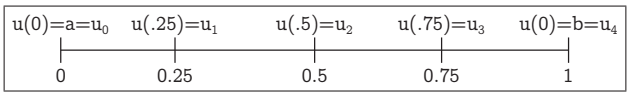

Figure 4: Represents the $n = 5$ discretization of a unit length wire for the finite difference solution to the heat equation. Notice that with this discretization, all the points are 0.25 distance away from immediate neighbors, implying h in Equation 4 is 0.25.

Since this is a numerical approximation, it is impossible to find $u(x)$ for all values of x. Instead we discretize the interval of interest into n finite number of points, where better approximations will be had for increasing values of n. In this example we allow $n = 5$, resulting in an interval as seen in Fig. 4.

We know $u_0 = a$ and $u_4 = b$ from the boundary conditions and can formulate a relationship between the u_i using the above finite difference equation (4). In general, the relationship among a point u_i and its neighbors u_{i-1} and u_{i+1} are given in Eq. 5. With this we can construct a linear system 6 to solve for the vector (u_1, u_2, u_3) and achieve a numerical solution to the steady state heat equation.

For simplicity we have presented a one-dimensional example where each point is concerned only with the difference between immediate neighbors, generating a tri-diagonal matrix, but this method can be expanded into higher dimensions. The linear systems *HPCCG* solves are generated from a 3D cube discretized into sub-cubes, where each sub-cube is concerned with all the neighbors sharing a face, edge, or corner—26 neighbors in all. This results in 27 element diagonal matrices of such size where it is computationally prohibitive to solve by Gaussian elimination. Thus, alternative methods must be used. Further on we will discuss one such alternative.

$$\frac{1}{h^2}\left(u_{i-1} - 2u_i + u_{i+1}\right) = 0 \tag{5}$$

$$\begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} a \\ 0 \\ b \end{bmatrix} \tag{6}$$

2.2 Sparse Matrices: Compressed Row Storage

Notice that the above finite difference matrices, for arbitrary integer values of n, take on the general form in 7:

$$\begin{bmatrix} 2 & -1 & \dots & 0 & 0 \\ -1 & 2 & -1 & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} a \\ 0 \\ 0 \\ \vdots \\ b \end{bmatrix} \tag{7}$$

This matrix can be called 'sparse'—since many of its values are zero it is more memory efficient to only store information about the non-zero values. There are several formats to store sparse matrices including: compressed row storage, compressed column storage, block compressed row storage, compressed diagonal storage, jagged diagonal storage, and skyline storage [3]. *HPCCG* uses compressed row storage (CRS) format for its sparse matrices. In CRS, there are three things stored for each row: the number of non-zeros, an array of the column indices of the non-zeros, and an array of the non-zero values. An example of the information needed to describe a full 4x4 sparse matrix in this format is seen in Fig. 5.

A C++ implementation for populating a *mxm* sparse matrix can be seen here:

```
double ** values = new double*[m];
int ** indices = new int*[m];
int * nonzeros = new int [m];
//Rows in the matrix
for(i = 0; i < m; ++i){
 nonzeros[i]=/*number of nonzeros in row i*/;
 values[i] = new double[nonzeros[i]];
 indices[i] = new int[nonzeros[i]];
 //For all nonzeros in row i
 for(j = 0; col < nonzeros[i]; ++j){
  values[i][j]=/*value of jth entry in row*/;
  indices[i][j]=/* column index of jth entry*/;
 }
}
```

Figure 5: An example of the data stored to represent a sparse matrix in the compressed row storage format.

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 6 \\ 5 & 8 & 1 & 2 \end{bmatrix}$$

$nonzeros[0] = 2$	$indices[0] = [0, 1]$	$values[0] = [2, 1]$
$nonzeros[1] = 0$	$indices[1] = [\,]$	$values[1] = [\,]$
$nonzeros[2] = 2$	$indices[2] = [0, 3]$	$values[2] = [4, 6]$
$nonzeros[3] = 4$	$indices[3] = [0, 1, 2, 3]$	$values[3] = [5, 8, 1, 2]$

2.3 The Conjugate Gradient Method

The Conjugate Gradient method is an iterative method that is used to solve linear systems $\vec{y} = A\vec{x}$ where A is a known matrix, $\vec{y}$ is a known vector, and $\vec{x}$ is an unknown vector. The proof of this method is beyond the scope of this paper, but those who wish to know more should look to the paper *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain* by Jonathan Shewchuk 1994 [7].

The essential routine of the conjugate gradient method is to give an initial guess for the vector $\vec{x}$, called $\vec{x_0}$, compute a sequence $\vec{x_k}$ $(k = 1, 2, 3...)$ such that each $\vec{x_k}$ is "closer" to the actual value of $\vec{x}$. We can understand $\vec{x_k}$ to be "close" to $\vec{x}$ if we first let $\vec{x_k} = \vec{x}$ for some k. Then for that value of k we define $\vec{r_k} = b - A\vec{x_k} = \vec{0}$ and it follows $\|\vec{r_k}\| = 0$, where $\vec{r}$ is called the residual vector. Then, for an arbitrary k, let $\vec{r_k} = \vec{b} - A\vec{x_k}$. If $\|\vec{r_k}\|$ is 'small' $(\approx 10^{-6})$, then we can say $\vec{x_k}$ is close to $\vec{x}$.

Given the initial guess, $\vec{x_0}$, calculating the subsequent $\vec{x_k}$ values is given by the algorithm below. For the matrix A, vectors $\vec{x}, \vec{b}, \vec{r}, \vec{p}, \vec{q}$, and scalars $\rho, \beta, \gamma, \alpha$ then:

1: $\vec{x}_0 = \vec{p}_0$
2: $A\vec{p}_0 = \vec{q}_0$
3: $\vec{r}_0 = \vec{b} - \vec{q}_0$
4: $\rho_0 =< \vec{r}_0, \vec{r}_0 >$
5: $\beta_0 = 0$
6: **while** $k \leq$ max k $\wedge$ $\|\vec{r}_k\| >$ tolerance **do**
7: **if** $k = 1$ **then**
8: $\vec{p}_1 = \vec{r}_1$
9: **else**
10: $\boxed{\rho_{k-1} =< \vec{r}_{k-1}, \vec{r}_{k-1} >}$
11: $\beta_{k-1} = \rho_{k-1}/\rho_{k-2}$
12: $\boxed{\vec{q}_k = A\vec{p}_k}$
13: $\gamma_k =< \vec{p}_k, \vec{q}_k >$
14: $\alpha_k = \rho_{k-1}/\gamma_k$
15: $\vec{x}_k = \vec{x}_{k-1} + \alpha_k\vec{p}_k$
16: $\boxed{\vec{r}_k = \vec{r}_{k-1} - \alpha_k\vec{q}_k}$
17: $k = k + 1$
18: **end if**
19: **end while**

An explanation of why this algorithm works can be found in the above mentioned Shewchuk paper, but this information is of minor importance to our purposes. It is sufficient to understand that this is an iterative process where each $\vec{x}_k$ is closer to the true $\vec{x}$, and the closeness is measured by $\|\vec{r}_k\|$. The process continues until a specified maximum number of iterations is achieved ($max\ k$) or $\|\vec{r}_k\|$ is small, i.e. less than a set 'tolerance'.

Of specific importance to our discussion further on are the three types of linear algebra operations present in this algorithm, highlighted above: the inner or dot product (e.g. line 10), matrix-vector multiplication (e.g. line 12), and vector-scalar multiplication and vector-vector addition (e.g. line 16). From here on out we will likely abbreviate these operations as, respectively: ddot, matvec, and waxpby (i.e. $\vec{w} = a\vec{x} + b\vec{y}$).

It will be relevant to future discussion to look at the complexity of these three functions in terms of the approximate number of mathematical operations and memory read/writes they require. For n length vectors, ddot requires n multiplications and $n - 1$ additions, for approximately $2n$ total mathematical operations. ddot requires reading in the value of each vector once, for a total of $2n$ memory reads.

Waxpby requires $2n$ multiplications to scale each vector and n additions to add the scaled vectors, for a total of $3n$ mathematical operations. It requires n reads of both vectors and n writes to the result vector for

Table 1: Shows the approximate number of mathematical operations and memory operations (read/write) necessary for each of CG's core functions. Where n is the length of the vector and z is the number of non-zeros in a sparse matrix.

	Number of Operations	Number of Memory Operations
DDOT	$2n$	$2n$
WAXPBY	$3n$	$3n$
MATVEC	$2z$	$2z + n$

$3n$ total reads/writes.

For a sparse matrix with z non-zero values, it requires approximately z multiplications and z additions, for a total of $2z$ mathematical operations. In terms of reads and writes it requires one read of all z non-zeros and z reads of the multiplicative vector as well as n writes to the result vector for a total of $2z + n$ memory operations. The matrices relevant to our computations have approximately 27 non-zeros per row, thus for square nxn matrices, matvec will require approximately $55 * n$ operations—clearly dominating over the computational complexity of ddot and waxpby. A summary of each functions complexities is seen in Table 1.

3 Target Machine: *Beast*

Our target machine for performance tuning is *Beast* at Saint John's University. It's technical specifications are:

- Fedora 14 Linux x86_64
- Linux kernel version 2.6.35.14-95
- AMD Opteron 6168 (12 cores at 1.9GHz; 12MB L3 Cache)x4 (48 cores total)
- 2GB DDR3 SDRAM ECC Unbuffered DDR3 1333MHz Memory x32 (64 GB total)

Beast is a non-uniform memory access machine with 8 NUMA regions. Each of the four 12 core chips form two NUMA regions connected via separate buses to 8GB of DRAM. An interconnect system links all the regions' buses and provides every core with access to the entire memory space, but as mentioned above,

accessing distant memory comes at a performance cost.

3.0.1 Quantifying NUMA on Beast

To quantify how much slower distant memory access is on *Beast* we performed a number of benchmarks. In the first benchmark, we stored a 100MB array on a NUMA region and had all the cores in *Beast* read from it concurrently. We repeated this process in a round robin fashion among all the NUMA regions. We timed how long it took for each core to finish reading and in Fig. 6 we present the results of this benchmark as a color-coded image where each color corresponds to one of the 8 NUMA regions. The numbers within each region are the ratios of the time it takes that region to read its data versus every other region's data. For example, in region 1 (red box), the black number is how much longer it takes to read data from region 0 (black box) than its own region, the gold number is how much longer it takes to read from region 2, etc. Notice the clear non-uniform behavior with ratios that vary from 1.2x to 2.1x increased time to access another region, with no noticeable pattern. We had guessed that regions sharing a physical chip (regions 0 and 1, or cores 0-11)would have an affinity for each other and have faster access to each others DRAM than other regions, but that hypothesis was invalidated by these results.

To test to what degree the non-uniform memory access is due to bandwidth contention (all the cores reading from the same block in memory at the same time) or latency (the distance and interconnects between the socket and DRAM) we performed a similar benchmark where each core takes turns reading the data array individually. The result of this test showed all cores reading distant regions 1.1x slower due to latency. Clearly latency plays a role, but bandwidth obviously dominates.

3.1 *Beast:* An Abstract Model

From the above results, we know that when the cores in Beast are accessing the same region in the shared memory space, performance can be reduced anywhere from 1.2x to 2.1x. Thus, to achieve the best performance, one must take into account this non-uniform behavior and design around it. To facilitate the design task of optimizing HPCCG for *Beast*, we first built a simplified, abstract machine and execution model to help make decisions about how to structure our solution.

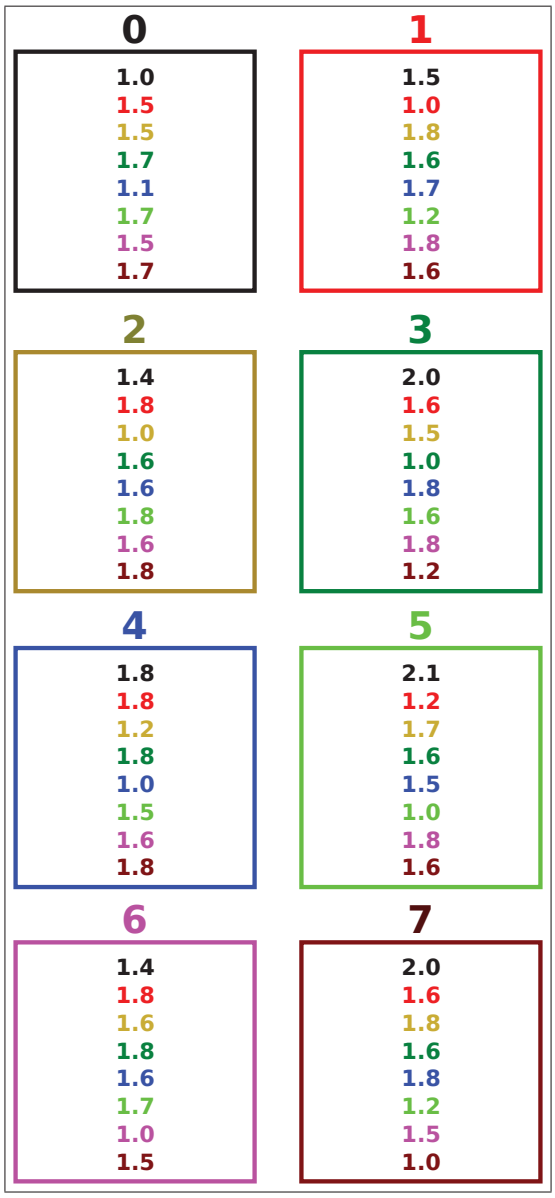

Figure 6: Shows the result of the NUMA benchmark of *Beast* where each region stores a 100MB array from which all the cores read. The time it takes for each core within a region to read the data array is recorded and averaged with its fellow cores. These timings are used to generate ratios of access time comparing the cores accessing nearby data to accessing distant data. The ratios are color coded such that text in a given color represents how much longer it takes to read from the region of the same color.

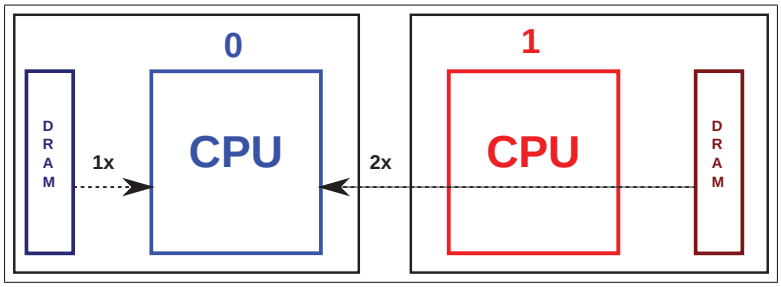

Figure 7: A simplified, abstract machine model that represents the non-uniform memory access found on *Beast*. In this model, when the CPU in region 0 accesses the distant DRAM in region 1, it is twice as slow as accessing the nearby DRAM in region 0.

3.1.1 Machine Model

In the abstract machine model, we homogenized the various increased access times and simply say that accessing distant memory is 2x as slow as accessing nearby memory. A simplified two core, two NUMA region diagram can be seen in Fig. 7. In this model, we see when the CPU in region 0 accesses the distant DRAM in region 1, it is twice as slow as accessing the nearby DRAM in region 0.

3.1.2 Execution Model

To see how the machine model informs performance questions, we present two execution models. In the first model, seen in Fig. 8, all of the data needed for a given computation is located in region 0. If the work is to be split between the two cores, then both cores will need to access the DRAM in region 0. If we call the time it takes CPU 0 to access DRAM 0 'λ', then we know from the machine model that it will take CPU 1 2λ time to access DRAM 0. Thus, memory access time for the computation in this scenario is 2λ.

In the second execution model, seen in Fig. 9, the data that CPU 0 needs for its portion of the work resides in DRAM 0 and the data CPU 1 needs resides in DRAM 1. We know from the machine model that each core can access its data in its nearby DRAM concurrently in λ time, for a total memory access time of λ.

The obvious conclusion we draw from the abstract machine and execution models is that any parallel computation ran on a non-uniform memory access machine like *Beast* should partition data so CPUs mainly work on nearby data.

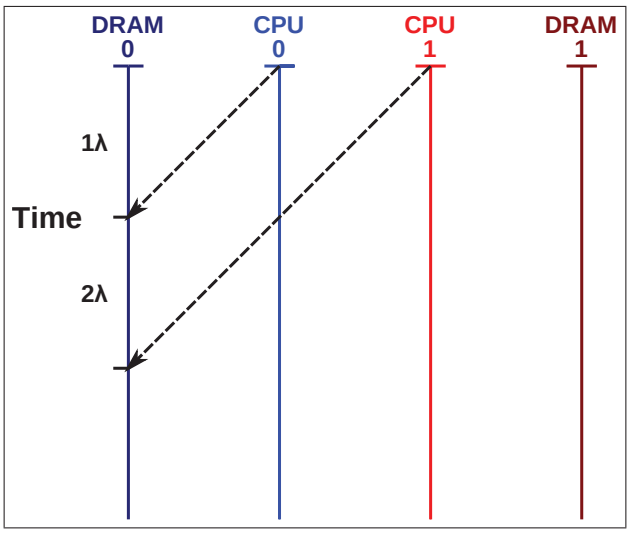

Figure 8: In the first execution model, all the data for computation is located in region 0's DRAM. It takes CPU 0 λ time to access this data and CPU 1 2λ time to do the same, resulting in a total memory access time of 2λ

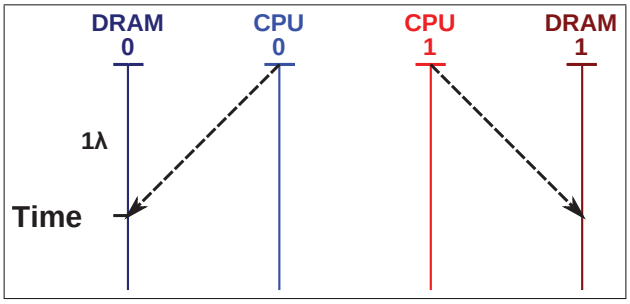

Figure 9: The second execution model has the data partitioned such that the data needed by CPU 0 is in DRAM 0 and the data needed by CPU 1 is in DRAM 1. Each CPU can access its own DRAM in λ time, resulting in a total memory access time of λ.

4 Explicitly Segmented Matrices and Vectors

4.1 The Control

In light of the NUMA benchmark and resulting machine and execution models, we sought to improve HPCCG's standard OpenMP implementation that makes no consideration of the non-uniform access. However, we must first describe HPCCG's normal design to serve as a backdrop to our improvements. This implementation will serve as the control dataset against which we will compare the performance of our new implementations.

Recall from previous discussion that HPCCG is a benchmark of parallel conjugate gradient (CG) performance for sparse matrices and is made up of three core functions: *ddot*, *matvec*, and *waxpby*. We will show and explain how these three functions are implemented in the control as they make up 99% of the computation time of the CG.

4.1.1 DDOT

First the *ddot* function. The function will compute the inner product of two passed vectors x and y, where it multiplies each x_i and y_i and sums the products. Note that these are vectors in the mathematical sense and are implemented with primitive *double* arrays, not the C++ vector class. The function uses an OpenMP *parallel for* directive which splits the n iterations of the *for* loop among the available threads (i.e. for n=100 and 4 threads, each thread will be responsible for 25 elements in the vectors). Additionally this pragma specifies *reduction+:local_result*, which hints to the compiler to optimize for the local results of each thread being summed together. The final result of the dot product is stored in the passed *result* pointer.

```
//From ddot.cpp
//Routine to compute the dot product of two vectors where:
//n - number of vector elements
//x, y - input vectors
//result - on exit will contain result.
int ddot (const int n, const double * const x, const double * const y,
          double * const result){
 double local_result = 0.0;
#pragma omp parallel for reduction(+:local_result)
    for (int i=0; i<n; i++)
     local_result += x[i]*y[i];
```

```
  *result = local_result;
  return(0);
}
```

4.1.2 WAXPBY

Next is the *waxpby* function which takes two vectors x and y, multiplies them respectively by the passed scalars *alpha* and *beta* and adds the two vectors—storing the result in the passed w vector. The *omp parallel* directive is used to mark a parallel region where the *omp for* directives split the n iterations of the *for* loop among the available threads and the two *if* checks are to prevent needless multiplications.

```
//From waxpby.cpp
//Routine to compute the update of a vector with
//the sum of two scaled vectors where:
//w = alpha*x + beta*y
//n - number of vector elements
//x, y - input vectors
//alpha, beta - scalars applied to x and y respectively.
//w - output vector.
int waxpby (const int n, const double alpha,
 const double * const x, const double beta,
 const double * const y, double * const w){
#pragma omp parallel
{
 if (alpha==1.0)
#pragma omp for
  for (int i=0; i<n; i++) w[i] = x[i] + beta * y[i];
 else if(beta==1.0)
#pragma omp for
  for (int i=0; i<n; i++) w[i] = alpha * x[i] + y[i];
 else
#pragma omp for
  for (int i=0; i<n; i++) w[i] = alpha * x[i] + beta * y[i];
}
  return(0);
}
```

4.1.3 MATVEC

Finally, *matvec* computes the matrix vector product $y = A * x$. This is complicated by the fact that we are dealing with sparse matrices. The HPC_Sparse_Matrix struct is defined in the HPCCG code and simply

stores the necessary fields for the compressed row storage format described above. The struct has the following fields:

- *A->local_nrow* field tells how many rows are in the matrix
- *A->ptr_to_vals_in_row[i]* field gives the array of non-zero values in row i
- *A->ptr_to_inds_in_row[i]* field gives the array of non-zero value indices for row i
- *A->nnz_in_row[i]* field tells how many non-zeros are present in row i

With this information the matrix vector product can proceed where the *omp for* divides the *nrow* iterations among the available threads so that each thread is responsible for a portion of the rows of A. Then, for each row of the matrix, i, the inner *for* loop iterates across the non-zero values in the row and multiplies it by the corresponding vector value in x and then sums these values into y_i.

```
//From HPC_sparsemv.cpp
//Routine to compute matrix vector product y = Ax where:
//A - known matrix
//x - known vector
//y - On exit contains Ax.
int HPC_sparsemv( HPC_Sparse_Matrix *A,
 const double * const x, double * const y){
#pragma omp parallel
{
int nrow = A->local_nrow;
#pragma omp for
 for (int i=0; i< nrow; i++){
  double sum = 0.0;
  double *cur_vals = A->ptr_to_vals_in_row[i];
  int *cur_inds =  A->ptr_to_inds_in_row[i];
  int cur_nnz = A->nnz_in_row[i];
 for (int j=0; j< cur_nnz; j++)
  sum += cur_vals[j]*x[cur_inds[j]];
 y[i] = sum;
 }
}
  return(0);
}
```

The most important thing to note is that the data used by these functions are all simply primitive data arrays that are declared and initialized in serial portions of the code—meaning that *all* the data will likely be stored in memory in a single region near whichever core happens to be the master core. A representative

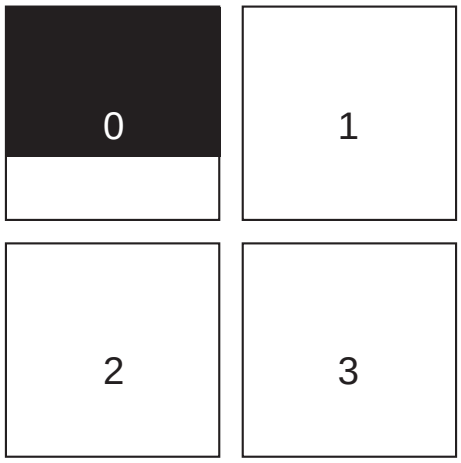

Figure 10: Represents the control implementation where data is represented by the black filled in area in each region. Notice that all the data is in region 0 and the rest are empty.

image is seen in Fig. 10 of the control's data distribution in four NUMA regions, where all the data is in region 0. If we apply the model we developed previously, then all the cores in region 0 will have λ time access to the data, but cores in the other regions will have 2λ access time. Clearly this is a non-optimal solution as 75% of the cores will suffer an extreme performance hit.

4.2 segMatrix and segVector Classes

To improve the performance of the HPCCG mini-app for Beast, we sought to build an implementation informed by the machine and execution models and avoid the pitfalls of the Control by ensuring that the data needed by a specific core is near that core.

Our solution is the introduction of two new classes: segMatrix and segVector. They take the data found in the control implementation and explicitly segment the data—ensuring it is distributed throughout the NUMA regions and cores will operate on primarily local data. We achieved this by reading in the data of the control implementation and copying it into new arrays that are declared and initialized in parallel sections of code. Since this occurs in a parallel region, it means that each thread will declare and initialize its portion of the data in nearby memory, so that future access will be fast. The data distribution of this new solution can be understood by looking at Fig. 11, where the data is now distributed throughout all the

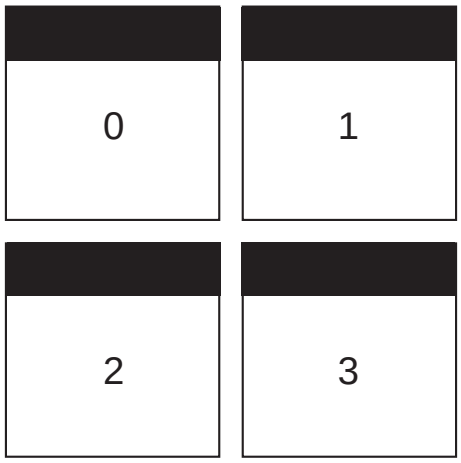

Figure 11: Represents the data distribution of the new segmented implementation where data is represented by the black filled in area in each region. Notice that the data is evenly distributed throughout all the regions.

regions. However, there is no way to reference data declared inside parallel regions once outside that region. We overcame this with a global array as long as there are many threads and at the close of the initialization parallel region, each thread stores a pointer to its local data at the index in the global array that corresponds to the thread's thread number. Therefore, whenever a thread needs to work with its portion of data in future parallel computations, it simply acquires the pointer to its data and proceeds accordingly.

A simple example of this for four threads can be seen in Fig. 12. The four thread local arrays are created in a parallel region and then each thread stores a pointer in the global array to its local data at the index of the thread number, e.g. thread 0 stores to the global array at index 0, thread 1 to index 1, etc. In future parallel regions, threads simply reference the global pointer at the index of their thread number to begin work with their local data.

4.2.1 Thread Pinning

Before continuing we need to underscore the importance of *thread pinning*. Our segmented implementation relies on a specific thread number running on the same physical core from one parallel region to the next. For example, say thread 0 initializes its data on physical core 0. Then, in another parallel region, thread 0 is now on core 9 and when it retrieves the pointer for thread 0's data it will be accessing data that is far away from

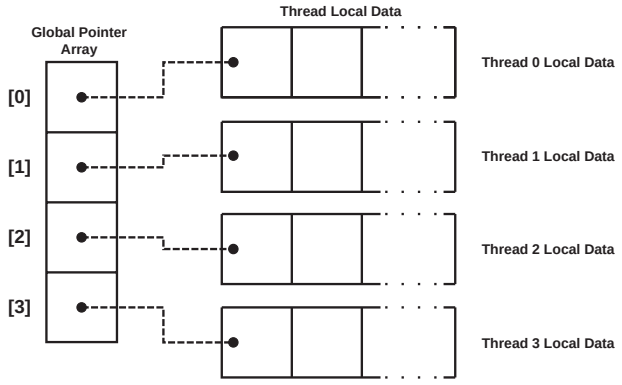

Figure 12: An example of how the explicitly segmented data is stored and referenced. Each thread local array was created within a parallel region and resides near the core that created it. To reference it for future use, each thread also stores a pointer to its data to a globally scoped array.

core 9, resulting in 2x slower memory operations. Thread pinning ensures this does not happen by enforcing thread n will always run on physical core n. This is achieved in OpenMP with the GNU compiler by setting the environment variable *GOMP_CPU_AFFINITY*. For example, *export GOMP_CPU_AFFINITY=0-47* will bind thread 0 to core 0, thread 1 to core 1, etc.

4.2.2 segVector Implementation

We show below how the two segmented classes are implemented in C++ with OpenMP. The segVector class partitions chunks of a primitive array among the available threads, e.g. if the array has length 100 and we are using 4 threads, then thread 0 gets elements 0-24, thread 1 gets 25-49, etc. This can be seen in Figure 13. First, it takes an input array to be segmented, the length of the array, and the number of threads to be used. It then allocates the global pointer array, *ptr_array*, which holds the start address of each thread's local data. It also allocates two arrays, *thread_id* and *offsets*, which are used to access values of the segmented vector and will described in more detail shortly. It then enters the parallel region where each thread establishes its thread number, the range of vector values it is responsible for and then initializes its thread local array. Finally it stores a pointer to the thread local data in the global pointer array.

```
// From segVector.cpp
// Constructor accepts premade primitive array
// and initializes a segVector with its values
```

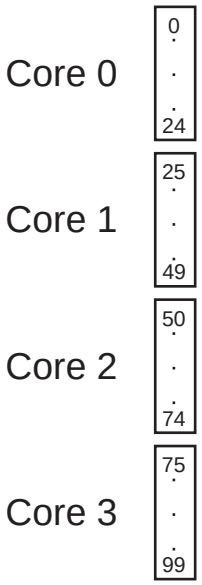

Figure 13: Diagram that provides an understanding of how a segVector partitions a standard array among cores. Core 0 is responsible for elements 0-24, Core 1 is responsible for elements 25-49, etc.

```
segVector::segVector(double * input_array,
 int length, int num_threads){
 // Holds address of each threads first element
 ptr_array=new double*[num_threads];
 // Holds location information for get() function
 // which is used to return the value of element 'n'
 // thread_id tells which thread element 'n' is on
 // offsets tells what the thread local
 // offset is for the specified element
 thread_id=new int[length];
 offsets=new int[length];
#pragma omp parallel
{
 int my_thread_num = omp_get_thread_num();
 int my_start, my_stop;
 // Computes range of elements thread is responsible for
 // from my_start to my_stop
 computeStartStop(my_thread_num, num_threads,
   length, my_start, my_stop, false);
 // How many elements this thread is responsible for
 int my_chunk_size = my_stop - my_start;

 // Thread local array
 double * my_data = new double[my_chunk_size];
 // Initialize local array to with values
```

```
 // this thread is responsible for from input array
 for(int i = 0; i<my_chunk_size; ++i){
 my_data[i] = input_array[my_start+i];
 }

 // Write to thread ID array and offset array
 int offset=0;
 for(int i = my_start; i < my_stop; ++i){
  thread_id[i]=my_thread_num;
  offsets[i]=offset;
  offset++;
 }
 // Stores start address of thread local
 // array to global pointer array
 ptr_array[my_thread_num] = my_data;
}
}
```

4.2.3 The *get(n)* Function

The *thread_ids* and *offsets* arrays are used in the segVector class function *get(n)*, which is used to return the value of an arbitrary element n in a segVector. In order to do this, one must know two things: which thread n was initialized by and what n's index is in that thread's local array. The *get(n)* function could be implemented where it computes both of these things (threadid and offset) on the fly, but it was found early on that it is much faster to simply store the information ahead of time in the *thread_ids* and *offsets* array for each n and simply retrieve it in order to return n's value.

```
// From segVector.cpp
// Returns the nth value of a segVector
// Uses the thread_id array to determine element n's thread location
// Uses offsets array to determine element n's thread local offset
double get(const int n) const{
 // Tells which thread element n is on
 int thread_num=thread_id[n];
 // Tells what the thread local offset is of element n
 int offset=offsets[n];
 // Points to the correct thread's local data
 double* element=ptr_array[thread_num];
 element+=offset;
 // Returns the desired element
 return *element;
}
```

4.2.4 segMatrix Implementation

The segMatrix class works in much the same way as segVector, except instead of partitioning elements in a vector, it partitions the rows of a sparse matrix. Recall from above that each row of a CRS sparse matrix has three things associated with it: the number of non-zeros, an array of the non-zero values, and an array of the column indices of each non-zero. Thus, each thread will be responsible for a sparse, sub-matrix of the actual matrix. Its constructor takes an HPC_Sparse_Matrix struct as its input matrix and the number of threads to be used. It then allocates three global arrays (*values_ptr*, *indices_ptr*, *nonzeros_ptr*) which will hold the pointers to the thread local data.

The parallel region begins and each thread discovers its thread number and computes the range of rows it is responsible for. Each thread then iterates through its rows from input matrix and populates the *my_row_nonzeros* array with the number of non-zeros per row. Likewise the *my_row_values* and *my_row_indices* arrays are populated in a double *for* loop by first iterating through the thread's rows and then through all the non-zeros in that row, grabbing the value and index of that matrix element. A pointer to each thread local array is then stored to the global array for future reference.

```
// from segMatrix.cpp
// Accepts an HPC_Sparse_Matrix struct and segments
// it among the number of threads specified
segMatrix::segMatrix(const HPC_Sparse_Matrix &input_matrix,
                     const int num_threads){
 num_rows = input_matrix.total_nrow;
 num_cols = input_matrix.local_ncol;
 // Global array holds addresses of
 // each thread local 2D array of row values
 values_ptr = new double**[num_threads];

 // Holds addresses of each
 // thread local 2D array of row indices
 indices_ptr = new int**[num_threads];

 // Holds addresses of each
 // thread local 1D array of row nonzeros
 nonzeros_ptr = new int*[num_threads];
#pragma omp parallel
{
 int my_thread_num = omp_get_thread_num();
 int my_start, my_stop;
 // Computes range of rows thread is responsible for
```

```
 // from my_start to my_stop
 computeStartStop(my_thread_num, num_threads,
  num_rows, my_start, my_stop, false);
 // How many rows thread owns
 int row_chunk_size = my_stop - my_start;

 // 1D array holds # non-zeros per row
 int * my_row_nonzeros = new int[row_chunk_size];
 for(int i=0; i < row_chunk_size; ++i){
  my_row_nonzeros[i] = input_matrix.nnz_in_row[my_start+i];
 }
 nonzeros_ptr[my_thread_num] = my_row_nonzeros;

 // Values
 // 2D array holds thread local row values
 double ** my_row_values = new double*[row_chunk_size];
 // Iterates across my rows
 for(int i=0; i < row_chunk_size; ++i){
  // Allocates storage for all non-zero values in row i
  my_row_values[i] = new double[my_row_nonzeros[i]];
  // For all non-zero columns in row i
  for(int j = 0; j < my_row_nonzeros[i]; ++j){
   // Store thread local values from sparse input matrix
   my_row_values[i][j] =
    input_matrix.ptr_to_vals_in_row[my_start+i][j];
  }
 }
  values_ptr[my_thread_num] = my_row_values;

 // Indices
 // 2D array holds thread local indices of non-zeros
 int ** my_row_indices = new int*[row_chunk_size];
 // Iterates across my rows
 for(int i=0; i < row_chunk_size; ++i){
  // Allocates storage for all non-zero values in row i
  my_row_indices[i] = new int[my_row_nonzeros[i]];
  // For all non-zero columns in row i
  for(int j = 0; j < my_row_nonzeros[i]; ++j){
   // Store thread local indices from sparse input matrix
   my_row_indices[i][j] =
    input_matrix.ptr_to_inds_in_row[my_start+i][j];
  }
 }
 indices_ptr[my_thread_num] = my_row_indices;
}
}
```

The *segVector* and *segMatrix* classes, in conjunction with thread pinning, ensure that the data is evenly

distributed throughout all the regions whose cores are active in the computation. Now the ddot, waxpby, and matvec functions can proceed with very few distant memory accesses. We now present how these functions have been reimplemented to use the new segmented data classes.

4.2.5 Segmented DDOT

The new dot product function takes two segVectors and computes their inner product. A diagrammatic understanding of the segmented implementation of ddot can be seen in Figure 14. Different from the control's *omp for* directive splitting iterations of a *for* loop, a *omp parallel* directive has each thread execute the code inside the parallel region. The *reduction(+:result)* tells the compiler that local result of each thread is to be summed. Inside the parallel region, each thread discovers its number, and the subset of vector elements it owns. The segVector class function, *getThreadPointer(thread_num)*, is used to get the pointer to the specified thread's local data for the x and y vectors. Finally the function computes the dot product for the range of vector elements it is responsible for, thus it is as if each thread is computing the dot product of a smaller sub-vector and then each thread's result is summed.

```
// From dot.cpp
// Takes x_ and y_ segVectors and
// returns result of the dot product of two segVectors
double dot(const segVector &x_, const segVector &y_){
 double result = 0.0;
#pragma omp parallel reduction(+:result){
 int my_thread_num = omp_get_thread_num();
 // Range of elements in x and y this thread is responsible for
 int my_start, my_stop;
 computeStartStop(my_thread_num, x_.getNumThreads(),
                  x_.getLength(), my_start, my_stop, false);
 int my_length = my_stop-my_start;
 // Pointers to this threads local data of x and y
 double * x = x_.getThreadPointer(my_thread_num);
 double * y = y_.getThreadPointer(my_thread_num);
 // Computes dot product on local sections of x and y
 for(int i=0; i<my_length; ++i){
  result+=x[i]*y[i];
 }
}
 // Final dot product of entire x and y
 return result;
}
```

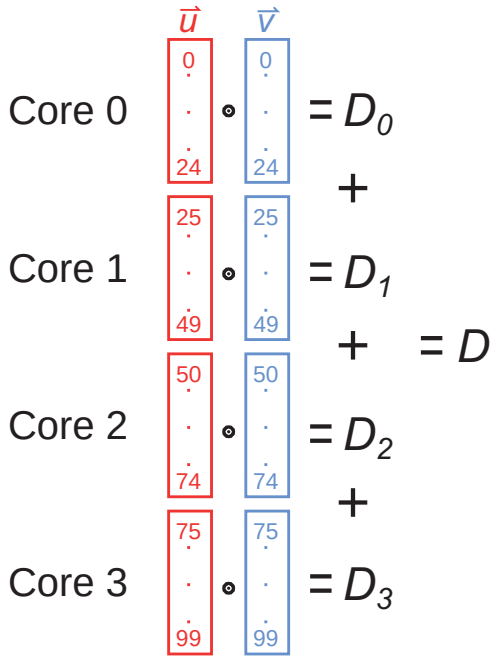

Figure 14: Diagram represents how the dot product is computed with the segmented data class. Each core computes the dot product on the local portions of the vectors and the local results of each core is summed for the complete dot product.

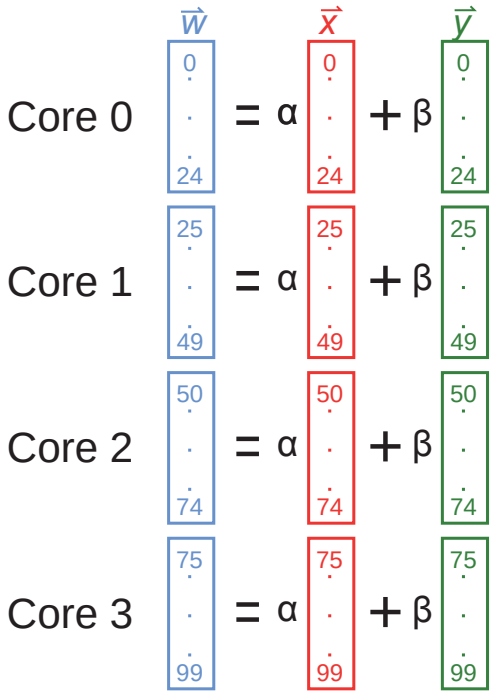

Figure 15: Diagram represents how waxpby is computed with the segmented data class. Each core scales and sums local portions of $\vec{x}$ and $\vec{y}$ and stores the results into local portion of $\vec{w}$.

4.2.6 Segmented WAXPBY

The new waxpby function works in much the same way as the new ddot. A diagrammatic understanding of the segmented implementation of waxpby can be seen in Figure 15. It takes two segVectors $x\_$ and $y\_$ and their multiplicative scalars *alpha* and *beta* as well as the result segVector $w\_$. The *omp parallel* directive specifies that each thread will execute the code in the parallel region where each thread acquires its number and the range of values it is responsible for in each vector. Each thread then acquires the pointer to its local data of $x\_$, $y\_$, and $w\_$ with the *getThreadPointer(thread_num)* function. Several *if* checks are performed in attempt to reduce needless multiplications and then the waxpby operation is performed on the thread local data.

```
// Takes segVectors x_, y_, and w_ with scalars alpha and beta and computes
// w_= alpha*x_ + beta*y_
void waxpby (const segVector &x_, const segVector &y_,
             const double alpha, const double beta, const segVector &w_){
```

```
#pragma omp parallel
 {
  int my_thread_num = omp_get_thread_num();
  // Range of elements this thread is responsible for
  int my_start, my_stop, dummy;
  computeStartStop(my_thread_num, x_.getNumThreads(),
                   x_.getLength(), my_start, my_stop, false);
  int my_length = my_stop-my_start;
  // Pointers to thread local data of x, y, and w
  double * x = x_.getThreadPointer(my_thread_num);
  double * y = y_.getThreadPointer(my_thread_num);
  double * w = w_.getThreadPointer(my_thread_num);
  // Performs waxpby operation on thread local data
  if(alpha == 0.0)
   for(int i=0; i<my_length; ++i)
    w[i] = y[i]*beta;
  else if (alpha == 1.0)
   for(int i=0; i<my_length; ++i)
    w[i] = x[i] + y[i]*beta;
  else if (beta == 0.0)
   for(int i=0; i<my_length; ++i)
    w[i] = x[i]*alpha;
  else if (beta == 1.0)
   for(int i=0; i<my_length; ++i)
    w[i] = x[i]*alpha + y[i];
  else
   for(int i=0; i<my_length; ++i)
    w[i] = x[i]*alpha + y[i]*beta;
 }
}
```

4.2.7 Segmented MATVEC

The new matvec function takes a segMatrix A, segVectors x and y and computes $A * x = y$. An *omp parallel* directive is used so that all threads will execute the code within the parallel region where each thread first determines its thread number and the range of rows in A and range of elements in x and y (same range of values) it is responsible for.

Notice that this is the only new, segmented function that can result in distant memory accesses. For example, if we have 4 threads (each in its own NUMA region) and 100 rows and columns in A and 100 elements in x, then thread 0 will be responsible for rows and elements 0-24. If any of the non-zero values in rows 0-24 have column indices 25-99, then this requires multiplying by an element in $\vec{x}$ 25-99 that is in

another NUMA region. Figure 16 shows this. If the band down the diagonal represents our diagonal matrix, then the green values represent the matrix values that require local $\vec{x}$ references and the orange require non-local $\vec{x}$ references. (Note: this is not meant to be accurate for our 27 element diagonal matrix.) This diagram also highlights an important aspect of our segmented design—we take advantage of the fact that we are working with diagonal matrices. If the matrix were dense, then a majority of the values in each row would require non-local $\vec{x}$ references and performance would be dismal.

Each thread then acquires pointers to its local data of A, x, and y. Then, in a double *for* loop, it iterates over its owned rows and for every non-zero in that row, it checks if the column index is between *my_start* and *my_stop*, i.e. a value in the green portion of Figure 16. If it is, then the matrix element is multiplied by the corresponding value in the locally referenced *my_x_values* vector. Otherwise, the necessary value of x is in another memory region and the segVector class *get()* function is used to return the needed value. After all the multiplications and additions for a given row are complete, the row result is stored in the corresponding value of y.

```
// From matvec.cpp
// A - segMatrix sparse matrix to be multiplied by x
// x - segVector to be multipied by A
// y - segVector stores results of  A*x
void matvec(const segMatrix &A, const segVector &x, segVector &y){
 int num_rows = A.getNumRows();
 int num_threads = A.getNumThreads();
#pragma omp parallel
{
 int my_thread_num = omp_get_thread_num();
 // Range of rows this thread is responsible for
 int my_start, my_stop;
 computeStartStop(my_thread_num, num_threads, num_rows,my_start, my_stop, false);
 int row_chunk_size = my_stop - my_start;
 // Pointers to thread local data of matrix A and vectors x, y
 double ** my_row_values = A.getValuesArray(my_thread_num);
 int ** my_row_indices = A.getIndicesArray(my_thread_num);
 int * my_row_nonzeros = A.getNonZerosArray(my_thread_num);
 double * my_x_values = x.getThreadPointer(my_thread_num);
 double * my_y_values = y.getThreadPointer(my_thread_num);
 int index;
 int local_vector_index;
 double row_result;
 int row_nonzeros;
 for(int i =0; i < row_chunk_size; ++i){
  row_result=0.0;
```

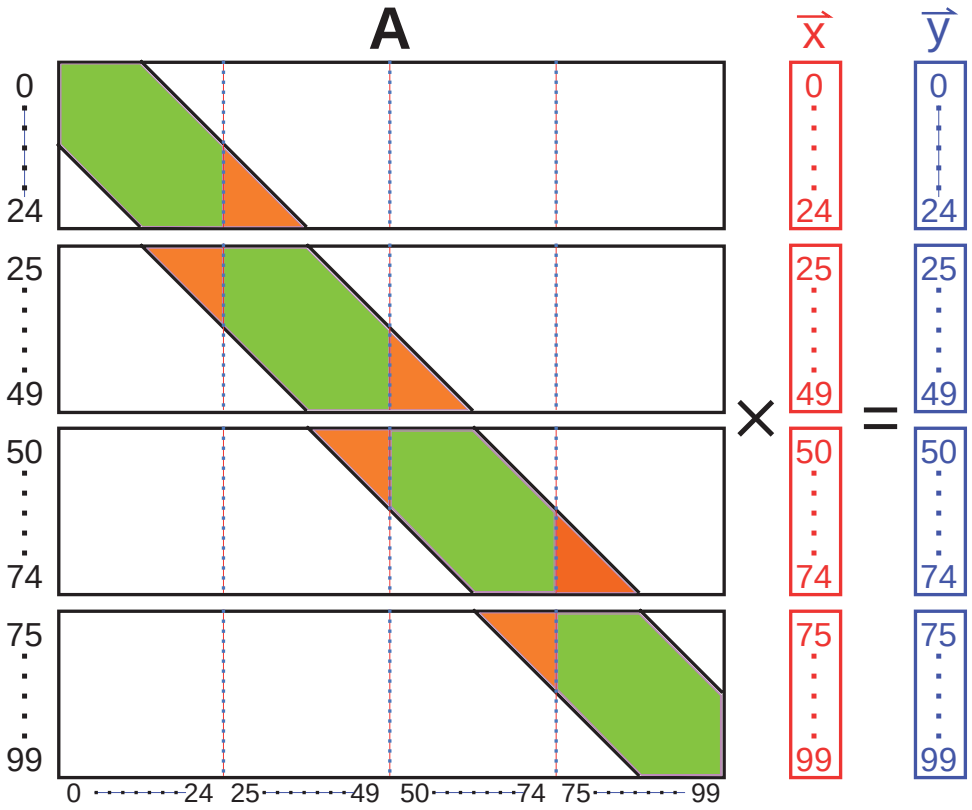

Figure 16: Represents the segmented implementation of matvec. Each core iterates over its rows and multiplies the non-zero values by the corresponding values in $\vec{x}$. The green portion of the matrix represents matrix values that require local references to $\vec{x}$ and the orange portion are values that require non-local references to $\vec{x}$.

```
  row_nonzeros = my_row_nonzeros[i];
  for(int j=0; j<row_nonzeros; ++j){
   index = my_row_indices[i][j];
   // Check if the vector value is near my thread
   if(index >= my_start && index < my_stop){
    local_vector_index = index - my_start;
    row_result += my_row_values[i][j]*my_x_values[local_vector_index];
   }
   // Otherwise use get() function
   else{
    row_result += my_row_values[i][j]*x.get(index);
   }
  }
  my_y_values[i]=row_result;
 }
}
}
```

4.3 Segmented Implementations

The segVector and segMatrix classes as described above are the foundation of our new implementation, but as we progressed in our solution design we thought of several variations to try to increase performance. These variations focus primarily on how to optimize the matvec function as it far more computationally complex than the other functions and dominates the computation of the CG. We discuss these variations below.

4.3.1 Standard Matrix

"Standard Matrix" is our baseline implementation and is simply the design as described above. It uses the full 27 element diagonal matrix with the matvec function implemented with the segVector *get(n)* function. The *get(n)* function is a point of weakness in this implementation because every time it is called, it requires an access to the *thread_id* and *offsets* arrays. This adds two additional memory operations to the already complex matvec function. Additionally, these arrays are not explicitly distributed through the NUMA regions and will result in distant memory access for most cores.

4.3.2 Best Matrix

In what we dub the Best Matrix implementation, we purposefully structure our matrix so it will result in each thread only making local memory references. These matrices are no longer representative of a real problem, but are an attempt to see how badly the *get(n)* function affects the performance of matvec in the CG. We build these matrices by scanning in and segmenting an HPC_Sparse_Matrix struct in parallel as in the normal Standard implementation except we only read in non-zero values whose indices are within the range of rows owned by a thread. For example, if thread 1 is responsible for rows 25-49 of the matrix, it will only store values whose indices are between 25 and 49. Thus, in the matvec function, each thread will only make local memory references. In summary, this implementation eliminates the orange portions of Figure 16.

4.3.3 Vector Pointer Array

In attempt to rid our solution of the *get(n)* function, we devised a method in the segMatrix class to store pointers to the needed values in a segVector for matrix vector multiplication. In the CG method, there is only one matrix and it is always being multiplied by the same vector. The values of the vector change, but their location in memory do not. We used this to our advantage and created a new constructor for the segMatrix class that accepts a segVector in addition to the standard arguments. In this new constructor, we create an additional, thread local, 2D array of the same dimensions as the values and indices arrays of the sparse matrix. The additional code for this implementation is seen below. In a similar double *for* loop, we iterate over each row of the matrix and create an array as long as there are non-zero values in the row. We then iterate over all the non-zeros in the row and determine their column index. Finally we acquire a pointer to the corresponding element in the segVector and store this to the thread local vector pointer array.

```
// 2D array of pointers to vector p values
double *** my_row_vec_ptrs = new double**[row_chunk_size];
// Vector pointers
for(int i = 0; i < row_chunk_size; ++i){
 // Room for this row's necessary pointers to vector values
 my_row_vec_ptrs[i] = new double*[my_row_nonzeros[i]];
 for(int j = 0; j < my_row_nonzeros[i]; ++j){
  // Column index for the jth non-zero value
  int matrix_column_index = my_row_indices[i][j];
```

```
  // Store pointer to corresponding vector value
  my_row_vec_ptrs[i][j] = (p.get_pointer(matrix_column_index));
  }
 }
vec_ptr[my_thread_num] = my_row_vec_ptrs;
```

The matvec function is reimplemented to use this pointer array to directly access the vector values without any computation or unnecessary distant memory access. In the snippet below, we see how it uses the same double *for* loop structure, but instead of using the *get(n)* function, it simply dereferences the pointer to the corresponding vector value.

```
for(int i =0; i < row_chunk_size; ++i){
 row_result=0.0;
 row_nonzeros = my_row_nonzeros[i];
 for(int j=0; j < row_nonzeros; ++j){
 // Simply dereference the pointer to the corresponding vector value
 row_result += my_row_values[i][j] *(*(my_vec_ptrs[i][j]));
}
my_y_values[i]=row_result;
}
```

Notice that this vector pointer array is the same size as the values and indices arrays and will increase the sparse matrix's size in memory by approximately 50%. Because our computation and environment are memory bound, the increased size may adversely affect performance and negate any benefit of eliminating the *get(n)* function.

4.3.4 Unsuccessful Implementations

There are several implementations that we tried where it was clear from early testing that they would not perform well at all. We mentioned one such implementation earlier where in place of the *get(n)* function, we compute the thread and thread local offset for an arbitrary element n in a segVector. We could see from initial testing that this implementation increased the runtime by a factor of two and we did not bother further exploring it.

Another idea we had was to store a full copy of the vector needed by the matrix vector multiplication in each NUMA region. The problem is that this vector is updated in other computations and we either needed to have each region perform the computation on its copy of the vector or perform the computation

on one copy and update the rest. We attempted both methods and they performed so poorly that we did not explore the idea further.

5 Results

We ran HPCCG on Beast with our three implementations and the control at problem dimensions of 100, 200, and 300. The problem size specifies the length, width, and height of the 3D cube that we are using to generate our matrix, thus, for a problem size of n, the matrix will be n^3xn^3 and the vectors will be length n^3. We ran each problem size using 6 cores, then 12 cores, etc. up to 48 cores. We repeated this ten times and recored the total number of MegaFLOPS (millions of floating point operations per second) for each problem size and cores in use. We present the average of the 10 runs and use the standard deviation as the measure of uncertainty. We also computed the maximum speed-up at each problem size of our solutions over the control by taking the highest MegaFLOPS of a given implementation and dividing it by the control's MegaFLOPS at the same number of cores. We propagated the error in the standard way seen in Equation 8.

$$\begin{aligned} R =& \frac{X}{Z} \\ \delta R =& |R| \sqrt{\left(\frac{\delta X}{X}\right)^2 + \left(\frac{\delta Z}{Z}\right)^2} \end{aligned} \tag{8}$$

5.1 Dimension Size 100

In Table 2 we can see the reported averages and standard deviation of all the implementations at problem size 100 for 6 through 48 cores. At this problem size, the matrix dimensions are 1,000,000x1,000,000 and the vectors are of length 1,000,000. The results were graphed in Fig. 17 with total MegaFLOPS on the y-axis and the number of cores used in the computation on the x-axis. From the graph, we can see the all the solutions performed nearly the same when 6 cores were in use. The three segmented solutions have nearly identical linear speed increases from 6 to 48 cores, with the Best Matrix solution slightly pulling ahead after 24 cores. The control has a gradual performance decrease and levels off as more cores are used. The maximum speed-ups of our solutions over the control are summarized in Table 3. Our Best Matrix solution

Table 2: Summarizes the total MegaFLOPS for each implementation at problem size 100 across 6 through 48 cores.

Cores	Control	Standard Matrix	Best Matrix	Vector Pointer
6	1710 ± 2	1522 ± 18	1551 ± 13	1351 ± 8
12	1229 ± 2	298 ± 64	3130 ± 25	2735 ± 20
18	1130 ± 2	4474 ± 55	4683 ± 45	4127 ± 36
24	1032 ± 3	5830 ± 128	6068 ± 148	549 ± 27
30	962 ± 3	7176 ± 153	758 ± 127	6810 ± 50
36	886 ± 7	8149 ± 172	8924 ± 165	8086 ± 83
42	975 ± 6	9205 ± 167	10320 ± 172	9302 ± 73
48	1038 ± 4	9980 ± 254	11562 ± 125	10126 ± 117

Table 3: Summarizes the maximum speed-ups of our three solutions over the control at problem size 100.

	Maximum Speed-up
Standard Matrix	9.6 ± 0.2
Best Matrix	11.1 ± 0.1
Vector Pointer	9.8 ± 0.1

shows a maximum speed up of 11.1 ± 0.1x faster than the control when 48 cores are in use.

5.2 Dimension Size 200

In Table 4 we can see the reported averages and standard deviation of all the implementations at problem size 200 for 6 through 48 cores. At this problem size, the matrix dimensions are 8,000,000x8,000,000 and vectors are of length 8,000,000. These results were graphed in Fig. 18 with total MegaFLOPS on the y-axis and the number of cores used in the computation on the x-axis. We can see the all the solutions performed nearly the same when 6 cores were in use and the three segmented solutions have nearly identical linear speed increases from 6 to 48 cores. The graph shows that the Best Matrix solution performed slightly better than the Standard Matrix solution, which performed marginally better than the Vector Pointer solution. The control has a gradual performance decrease and levels off as more cores are used. The maximum speed-ups of our solutions over the control are summarized in Table 5 and our Best Matrix solution shows a maximum speed up of 10.8 ± 0.1x faster than the control when 48 cores are in use.

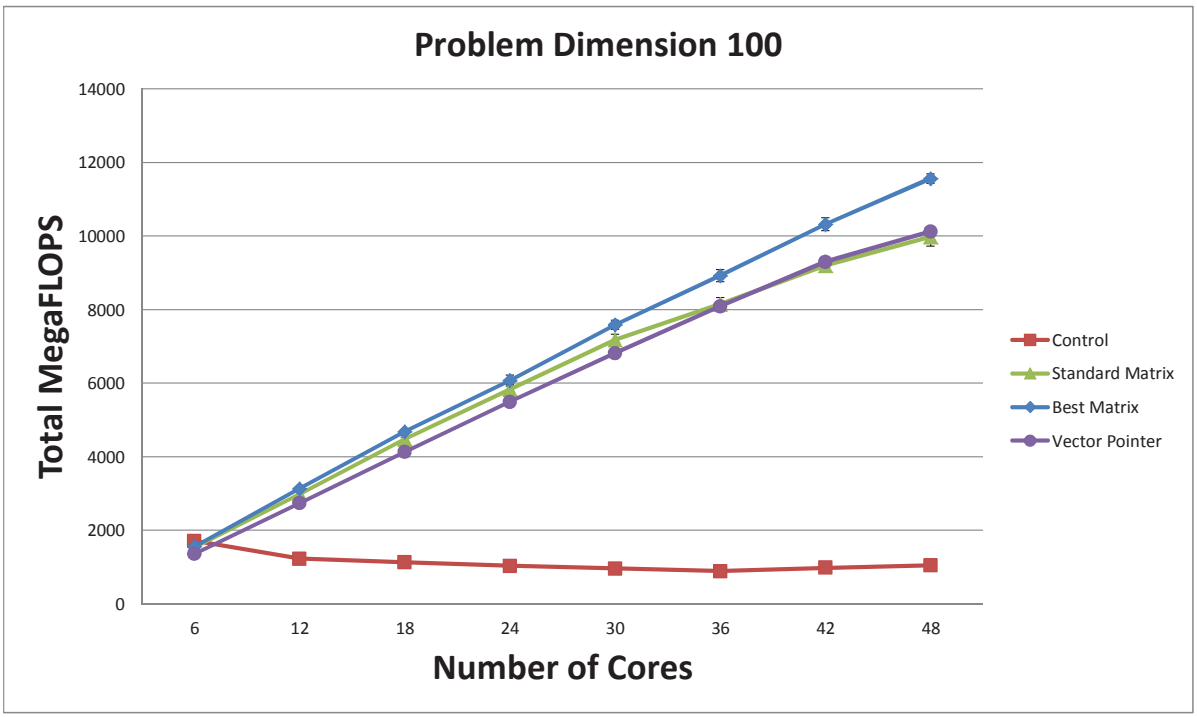

Figure 17: Shows the performance trend for the control and our three implementations at problem size 100 across 6 through 48 cores.

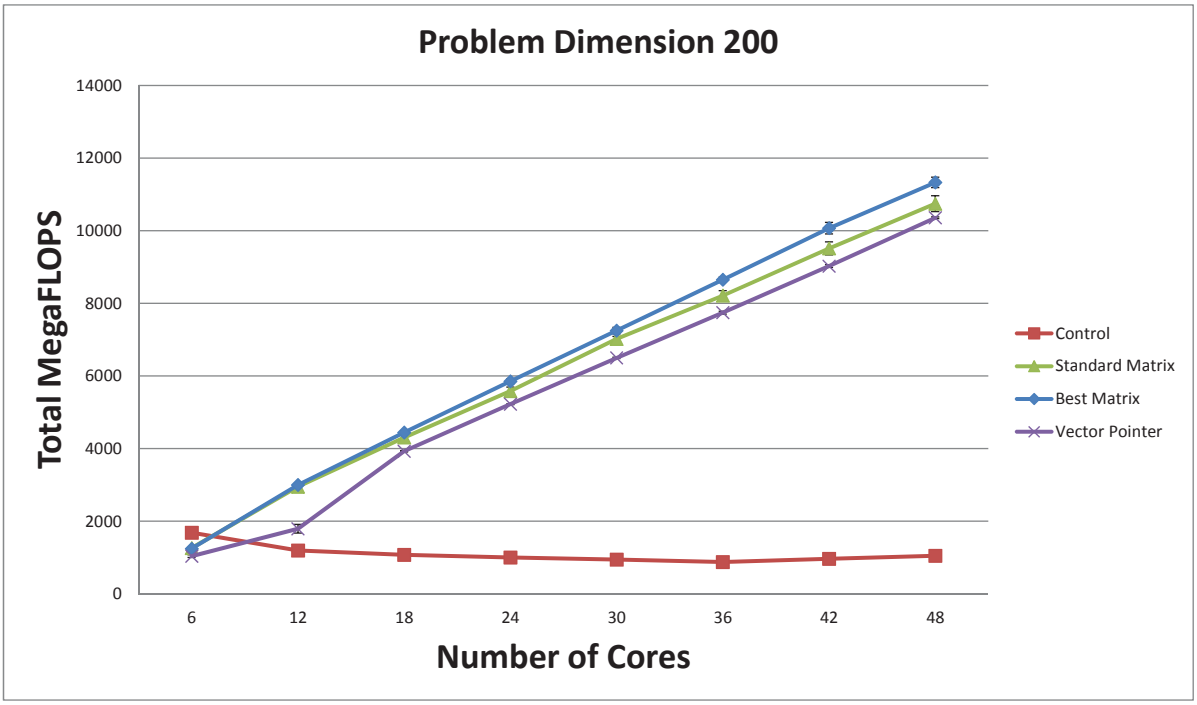

Figure 18: Shows the performance trend for the control and our three implementations at problem size 200 across 6 through 48 cores.

Table 4: Summarizes the total MegaFLOPS for each implementation at problem size 200 across 6 through 48 cores.

Cores	Control	Standard Matrix	Best Matrix	Vector Pointer
6	1685 ± 1	1262 ± 19	1245 ± 14	1037 ± 36
12	1195 ± 4	2953 ± 33	2998 ± 39	1795 ± 120
18	1077 ± 3	4302 ± 99	4442 ± 50	3927 ± 19
24	1003 ± 3	5583 ± 110	5854 ± 66	5223 ± 15
30	943 ± 2	7021 ± 70	7250 ± 90	6497 ± 18
36	876 ± 8	8213 ± 138	8652 ± 70	7743 ± 38
42	964 ± 6	9513 ± 179	10070 ± 158	9025 ± 37
48	1048 ± 4	10745 ± 218	11329 ± 142	10356 ± 23

Table 5: Summarizes the maximum speed-ups of our three solutions over the control at problem size 200.

	Maximum Speed-up
Standard Matrix	10.3 ± 0.2
Best Matrix	10.8 ± 0.1
Vector Pointer	9.9 ± 0.1

5.3 Dimension Size 300

In Table 6 we can see the reported averages and standard deviation of all the implementations at problem size 300 for 6 through 48 cores. At this problem size, the matrix dimensions are 27,000,000x27,000,000 and vectors are of length 27,000,000. The results were graphed in Fig. 19 with total MegaFLOPS on the y-axis and the number of cores used in the computation on the x-axis. The performance of this problem size is far more erratic than previous results with the control outperforming our segmented solutions when 6 and 12 cores are in use and then dropping sharply in performance when 18 cores are in use with a gradual rise in performance to 42 cores. We theorize that at the 300 problem size, the data does not fit fully in one NUMA region and runs into the next. Thus, the addition of the six cores in the next NUMA region have a chance to work with nearby data and would show the performance boost when 12 cores are used. The segmented solutions follow nearly identical increases in performance, within error, from 6 to 48 cores. The maximum speed-ups of our solutions over the control are summarized in Table 7 and our Vector Pointer solution shows a maximum speed up of 3.00 $\pm$ 0.18x faster than the control when 48 cores are in use.

Table 6: Summarizes the total MegaFLOPS for each implementation at problem size 300 across 6 through 48 cores.

Cores	Control	Standard Matrix	Best Matrix	Vector Pointer
6	1129 ± 8	440 ± 6	443 ± 1	549 ± 30
12	2354 ± 4	766 ± 5	762 ± 3	668 ± 27
18	917 ± 12	1344 ± 11	1340 ± 22	1359 ± 16
24	708 ± 6	2211 ± 18	2185 ± 11	1768 ± 102
30	1104 ± 6	2730 ± 16	2715 ± 17	2534 ± 162
36	1397 ± 3	2952 ± 13	2897 ± 13	3281 ± 238
42	1577 ± 7	3770 ± 28	3698 ± 12	3822 ± 263
48	1327 ± 5	3764 ± 24	3758 ± 16	3985 ± 233

Table 7: Summarizes the maximum speed-ups of our three solutions over the control at problem size 300.

	Maximum Speed-up
Standard Matrix	2.84 ± 0.02
Best Matrix	2.83 ± 0.02
Vector Pointer	3.00 ± 0.18

Notice that the speed-up as well as the range of MEGAflops is much lower at this problem size than previously seen. We are unable to fully explain why this is, but we hypothesize that it is due to the large amount of data involved in the computation at this problem size in comparison to the previous problem sizes. At the 300 problem size, the total memory footprint of our code is ≈22GB, in comparison to ≈7GB at problem size 200, and ≈4GB at problem size 100. The memory footprint at 300 problem size is not enough to overflow DRAM, forcing storage on disk, but it is considerable and we could find no other reason for this performance reduction.

6 Conclusion

We wished to test the impact of non-uniform memory access on *Beast* and explore how to optimize HPCCG's implementation for this unique architecture. We began by quantifying to what degree memory access on *Beast* was non-uniform. From these results we theorized a machine model where accessing memory in a distant NUMA region is twice as slow as accessing memory nearby. With this model we devised explicitly

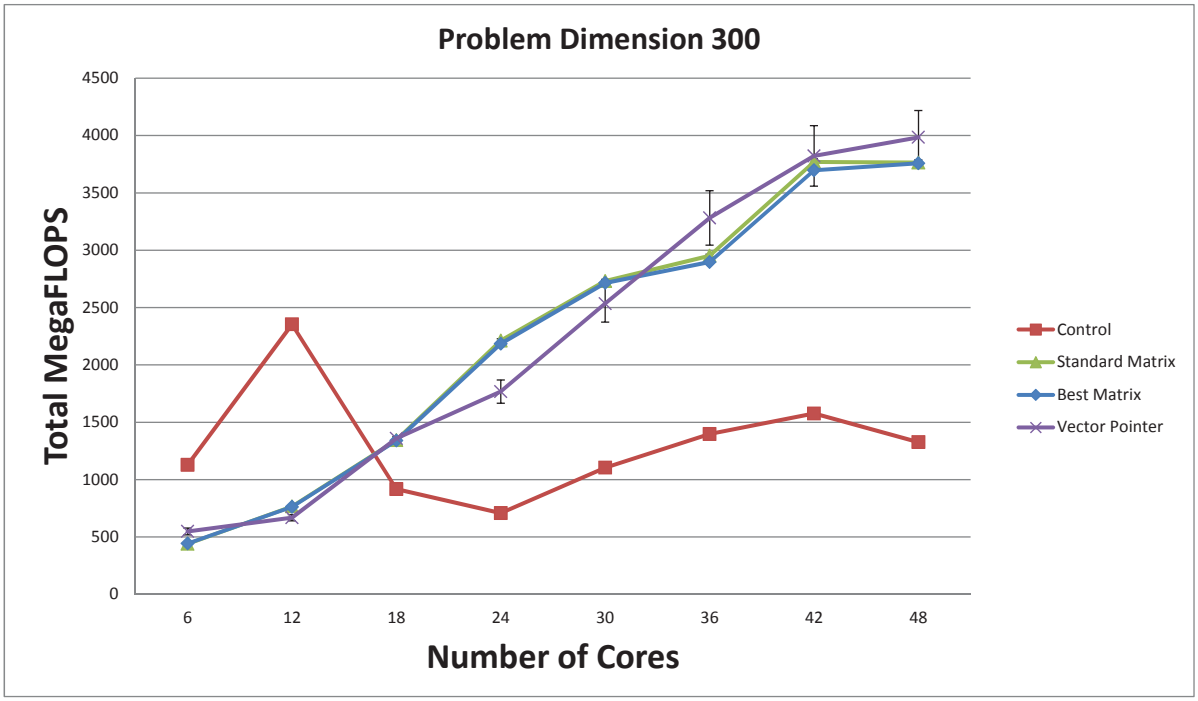

Figure 19: Shows the performance trend for the control and our three implementations at problem size 300 across 6 through 48 cores.

segmented data classes that ensured data is distributed throughout the memory space. We tested three implementations of our segMatrix and segVector solution against the control at problem sizes of 100, 200, and 300 using 6 through 48 of *Beast's* cores. Our results showed that our segmented solutions outperformed the control at nearly all problem sizes and number of cores used, for a maximum speed up of 11.1 $\pm$ 0.1x faster than the control. Additionally, the three different implementations performed at a relatively equal level.

From these results we conclude the following:

- In NUMA environments, explicitly distributing problem data through the memory space so cores primarily work on nearby data will realize greater performance than naive data placement.
- Our abstract model of Beast's NUMA environment appears to be a valid tool for reasoning about problem design and may prove useful for future problems.
- Specific to our task of optimizing HPCCG, it does not significantly matter how off region memory accesses are handled, as seen by the small variance between our three implementations.
- In order to achieve the greatest scalable performance when designing code for NUMA machines, great care must be used to ensure data is distributed through the memory regions so as to reduce the number of off region memory hits.

6.1 Future Work

In future work for further optimization of HPCCG for *Beast* we would look further at the matvec function and how we might have each thread read in the values of the multiplicative vector ahead of time into a thread local array. The structure of our sparse matrices result in a large amount reuse of the same values in the vector and our current implementations do not take advantage of this. We would also look at how we might structure the data in all the functions to take advantage of cache blocking. In other words, explicitly partitioning problem data into sizes that fit into each core's cache.

References

[1] Non-uniform memory access. Wikipedia http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access.

[2] Symmetric multiprocessing. Wikipedia http://en.wikipedia.org/wiki/Symmetric_multiprocessing.

[3] J. Dongarra. *Sparse Matrix Storage Formats in Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*, pages 372–378. SIAM, Philadelphia, 2000.

[4] Paul Dutton. Pentium 5 will launch with 64-bit windows elements, September 2006. Available: http://www.theinquirer.net/inquirer/news/1035334/pentium-v-will-launch-with--64-bit-windows-elements.

[5] Sandia National Laboratories. Mantevo project. Available: http://www.mantevo.org/.

[6] Paul E. McKenney. *Is Parallel Programming Hard, And, If So, What Can You Do About It?* kernel.org, Corvallis, OR, USA, 2011. Available: http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.

[7] Jonathan Richard Shewchuk. *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain.* Carnegie Mellon University, 1 1/4 edition, 1994.

A Segmented Code

```
//@HEADER
// ************************************************************************
//
//               HPCCG: Simple Conjugate Gradient Benchmark Code
//                 Copyright (2006) Sandia Corporation
//
// Under terms of Contract DE-AC04-94AL85000, there is a non-exclusive
// license for use of this work by or on behalf of the U.S. Government.
//
```

```
// This library is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; either version 2.1 of the
// License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful, but
// WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
// USA
// Questions? Contact Michael A. Heroux (maherou@sandia.gov)
//
// ************************************************************************
//@HEADER

#ifndef GENERATE_MATRIX_H
#define GENERATE_MATRIX_H
#ifdef USING_MPI
#include <mpi.h>
#endif
#include "HPC_Sparse_Matrix.hpp"

void generate_matrix(int nx, int ny, int nz, HPC_Sparse_Matrix **A, double **x,
    double **b, double **xexact);
#endif
```

```
//@HEADER
// ************************************************************************
//
//               HPCCG: Simple Conjugate Gradient Benchmark Code
//                 Copyright (2006) Sandia Corporation
//
// Under terms of Contract DE-AC04-94AL85000, there is a non-exclusive
// license for use of this work by or on behalf of the U.S. Government.
//
// This library is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; either version 2.1 of the
// License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful, but
// WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// Lesser General Public License for more details.
```

```
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
// USA
// Questions? Contact Michael A. Heroux (maherou@sandia.gov)
//
// ************************************************************************
//@HEADER

////////////////////////////////////////////////////////////////////////

// Routine to read a sparse matrix, right hand side, initial guess,
// and exact solution (as computed by a direct solver).

////////////////////////////////////////////////////////////////////////

// nrow - number of rows of matrix (on this processor)

#include <iostream>
using std::cout;
using std::cerr;
using std::endl;
#include <cstdlib>
#include <cstdio>
#include <cassert>
#include "generate_matrix.hpp"
void generate_matrix(int nx, int ny, int nz, HPC_Sparse_Matrix **A, double **x,
    double **b, double **xexact)

{
#ifdef DEBUG
  int debug = 1;
#else
  int debug = 0;
#endif

#ifdef USING_MPI
  int size, rank; // Number of MPI processes, My process ID
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#else
  int size = 1; // Serial case (not using MPI)
  int rank = 0;
#endif


  int local_nrow = nx*ny*nz; // This is the size of our subblock
  assert(local_nrow>0); // Must have something to work with
```

```
 int local_nnz = 27*local_nrow; // Approximately 27 nonzeros per row (except for
     boundary nodes)

 int total_nrow = local_nrow*size; // Total number of grid points in mesh
 long long total_nnz = 27* (long long) total_nrow; // Approximately 27 nonzeros
     per row (except for boundary nodes)

 int start_row = local_nrow*rank; // Each processor gets a section of a chimney
     stack domain
 int stop_row = start_row+local_nrow-1;


 // Allocate arrays that are of length local_nrow
 int *nnz_in_row = new int[local_nrow];
 double **ptr_to_vals_in_row = new double*[local_nrow];
 int    **ptr_to_inds_in_row = new int   *[local_nrow];
 double **ptr_to_diags       = new double*[local_nrow];

 *x = new double[local_nrow];
 *b = new double[local_nrow];
 *xexact = new double[local_nrow];


 // Allocate arrays that are of length local_nnz
 double *list_of_vals = new double[local_nnz];
 int *list_of_inds = new int   [local_nnz];

 double * curvalptr = list_of_vals;
 int * curindptr = list_of_inds;

 long long nnzglobal = 0;
 for (int iz=0; iz<nz; iz++)
   for (int iy=0; iy<ny; iy++)
     for (int ix=0; ix<nx; ix++) {
int curlocalrow = iz*nx*ny+iy*nx+ix;
int currow = start_row+iz*nx*ny+iy*nx+ix;
int nnzrow = 0;
ptr_to_vals_in_row[curlocalrow] = curvalptr;
ptr_to_inds_in_row[curlocalrow] = curindptr;
for (int sz=-1; sz<=1; sz++)
  for (int sy=-1; sy<=1; sy++)
    for (int sx=-1; sx<=1; sx++) {
      int curcol = currow+sz*nx*ny+sy*nx+sx;
      if (curcol>=0 && curcol<total_nrow) {
 if (curcol==currow) {
   ptr_to_diags[curlocalrow] = curvalptr;
   *curvalptr++ = 27.0;
 }
 else
   *curvalptr++ = -1.0;
```

```
  *curindptr++ = curcol;
  nnzrow++;
       }
     }
 nnz_in_row[curlocalrow] = nnzrow;
 nnzglobal += nnzrow;
 (*x)[curlocalrow] = 0.0;
 (*b)[curlocalrow] = 27.0 - ((double) (nnzrow-1));
 (*xexact)[curlocalrow] = 1.0;
      } // end ix loop

  if (debug) cout << "Process␣"<<rank<<"␣of␣"<<size<<"␣has␣"<<local_nrow;

  if (debug) cout << "␣rows.␣Global␣rows␣"<< start_row
    <<"␣through␣"<< stop_row <<endl;

  if (debug) cout << "Process␣"<<rank<<"␣of␣"<<size
    <<"␣has␣"<<local_nnz<<"␣nonzeros."<<endl;

  *A = new HPC_Sparse_Matrix; // Allocate matrix struct and fill it
  (*A)->title = 0;
  (*A)->start_row = start_row ;
  (*A)->stop_row = stop_row;
  (*A)->total_nrow = total_nrow;
  (*A)->total_nnz = total_nnz;
  (*A)->local_nrow = local_nrow;
  (*A)->local_ncol = local_nrow;
  (*A)->local_nnz = local_nnz;
  (*A)->nnz_in_row = nnz_in_row;
  (*A)->ptr_to_vals_in_row = ptr_to_vals_in_row;
  (*A)->ptr_to_inds_in_row = ptr_to_inds_in_row;
  (*A)-> ptr_to_diags = ptr_to_diags;

  return;
}
```

```
//@HEADER
// ***************************************************************************
//
//               HPCCG: Simple Conjugate Gradient Benchmark Code
//                 Copyright (2006) Sandia Corporation
//
// Under terms of Contract DE-AC04-94AL85000, there is a non-exclusive
// license for use of this work by or on behalf of the U.S. Government.
//
// This library is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; either version 2.1 of the
```

```
// License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful, but
// WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
// USA
// Questions? Contact Michael A. Heroux (maherou@sandia.gov)
//
// ************************************************************************
//@HEADER

#ifndef HPC_SPARSE_MATRIX_H
#define HPC_SPARSE_MATRIX_H

// These constants are upper bounds that might need to be changes for
// pathological matrices, e.g., those with nearly dense rows/columns.

const int max_external = 100000;
const int max_num_messages = 500;
const int max_num_neighbors = max_num_messages;


struct HPC_Sparse_Matrix_STRUCT {
  char   *title;
  int start_row;
  int stop_row;
  int total_nrow;
  long long total_nnz;
  int local_nrow;
  int local_ncol;  // Must be defined in make_local_matrix
  int local_nnz;
  int  * nnz_in_row;
  double ** ptr_to_vals_in_row;
  int ** ptr_to_inds_in_row;
  double ** ptr_to_diags;

#ifdef USING_MPI
  int num_external;
  int num_send_neighbors;
  int *external_index;
  int *external_local_index;
  int total_to_be_sent;
  int *elements_to_send;
  int *neighbors;
  int *recv_length;
```

```
  int *send_length;
  double *send_buffer;

#endif
};
typedef struct HPC_Sparse_Matrix_STRUCT HPC_Sparse_Matrix;
#endif
```

```
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <omp.h>
#include "segVector.h"
#include "dot.h"
#include "waxpby.h"
#include "segMatrix.h"
#include "matvec.h"
#include "HPC_Sparse_Matrix.hpp"
#include <cmath>
#include "mytimer.hpp"
#include "generate_matrix.hpp"
#include <assert.h>
#include "results_log.h"

#define TICK()  t0 = mytimer() // Use TICK and TOCK to time a code section
#define TOCK(t) t += mytimer() - t0
#define MATVEC_OPTION 2 // Specifies which implementation of matvec() to use. See
    matvec.cpp

int main(int argc, char * argv[]) {
  if(argc != 3) {
    std::cout << "Usage: " << argv[0] << " Length(x*y*z) Num_Threads" << std::endl
        ;
    exit(1);
  }
  int nx = atoi(argv[1]);
  int ny = atoi(argv[1]);
  int nz = atoi(argv[1]);
  // nx*ny*nz 3D cube
  int length = nx*ny*nz;
  int num_threads = atoi(argv[2]);


  double t0 = 0.0, t1 = 0.0, t2 = 0.0, t3 = 0.0, t4 = 0.0;

  omp_set_num_threads(num_threads);

  double *x, *b, *xexact;
  HPC_Sparse_Matrix  *sparse_matrix;
```

```
generate_matrix(nx,ny,nz,&sparse_matrix,&x,&b,&xexact);
segMatrix seg_A(*sparse_matrix,num_threads); //Matrix A


// segVectors r, x, p, q, b
segVector seg_r(length,num_threads);
segVector seg_x(x,length,num_threads);
segVector seg_b(b,length,num_threads);
segVector seg_p(length,num_threads);
segVector seg_q(length,num_threads);
//Scalars
double normr=0.0;
double rho=0.0;
double oldrho=0.0;
double tolerance=0.0;
int max_iter = 150;
int niters = 0;
// Used for prints
int print_freq = max_iter/10;
if (print_freq>50) print_freq=50;
if (print_freq<1)  print_freq=1;

// Start timing right away
double t_begin = mytimer();

// x=p
// waxpby(const segVector &x_, const segVector &y_, const double alpha_, const
    double beta_, const segVector &w_);
TICK(); waxpby(seg_x,seg_x,1.0,0.0,seg_p);  TOCK(t1);


// A*p = q
// matvec(const segMatrix &A, const segVector &x, segVector &y, const int option
    );
TICK(); matvec(seg_A, seg_p, seg_q,MATVEC_OPTION); TOCK(t2);

// r=b-q
// waxpby(nrow, 1.0, b, -1.0, Ap, r)
 TICK(); waxpby(seg_b,seg_q,1.0,-1.0,seg_r); TOCK(t1);


// rho=<r,r>
rho = dot(seg_r,seg_r);

//sqrt(<r,r>)
normr = sqrt(rho);

std::cout << "Initial␣Residual␣=␣" << normr << std::endl;

for(int k=1; k<max_iter && normr > tolerance; ++k ) {
```

```
// p(1) = r(1)
  if(k == 1){
    TICK();
    waxpby(seg_r,seg_r,1.0,0.0,seg_p);
    TOCK(t1);
  }else{
    oldrho = rho;

    // #1
    // rho=<r,r>
    TICK();
    rho = dot(seg_r,seg_r);
    TOCK(t3);

    // #2
    // beta=rho/oldrho
    double beta = rho/oldrho;

    // #3
    // p(i)= r(i-1)+beta(i-1)*p(i-1)
    TICK();
    waxpby(seg_r,seg_p,1.0,beta,seg_p);
    TOCK(t1);
 }

 normr = sqrt(rho);

 if(k%print_freq == 0 || k+1 == max_iter)
    std::cout << "Iteration␣=␣" << k << "␣␣␣Residual␣=␣" << normr << std::endl;

 // #4
 // q(i)=A*p(i)
 TICK();
 matvec(seg_A,seg_p,seg_q,MATVEC_OPTION);
 TOCK(t2);

 double alpha = 0.0;

 // #5
 // alpha=<p(i),q(i)>
 TICK();
 alpha = dot(seg_p,seg_q);
 TOCK(t3);

 // #6
 // alpha=rho(i-1)/alpha
 alpha = rho/alpha;

 // #7
 // x(i)=x(i-1) + alpha(i)*p(i)
```

```
    TICK();
    waxpby(seg_x,seg_p,1.0,alpha,seg_x);
    TOCK(t1);

    // #8
    // r(i)=r(i-1) - alpha(i)*q(i)
    TICK();
    waxpby(seg_r,seg_q,1.0,-alpha,seg_r);
    TOCK(t1);

    niters=k;
}


    t0=mytimer()-t_begin;

    double fniters = niters;
    double fnrow = sparse_matrix->total_nrow;
    double fnnz = sparse_matrix->total_nnz;
    double fnops_ddot = fniters*4*fnrow;
    double fnops_waxpby = fniters*6*fnrow;
    double fnops_sparsemv = fniters*2*fnnz;
    double fnops = fnops_ddot+fnops_waxpby+fnops_sparsemv;
    double results[9];


  //std::cout << "Final Residual = " << normr << std::endl;
  //std::cout << "Threads used = " << num_threads << std::endl;
  //std::cout << "   waxpby time = " << t1 << " sec" << std::endl;
  //std::cout << "   matvec time = " << t2 << " sec" << std::endl;
  //std::cout << "   dot time = " << t3 << " sec" << std::endl;
  //std::cout << "   Total time = " << t0 << " sec" << std::endl;

   std::cout << "Number␣of␣iterations␣=␣" << niters << ".\n" << std::endl;
      std::cout << "Final␣residual␣=␣" << normr << ".\n" << std::endl;
      std::cout << "**********␣Performance␣Summary␣(times␣in␣sec)␣**********" <<
          std::endl << std::endl;
      std::cout << "Total␣Time/FLOPS/MFLOPS␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣=␣"
     << t0 << "/" << fnops << "/"
           << fnops/t0/1.0E6 << "." << std::endl;
      std::cout << "DDOT␣␣Time/FLOPS/MFLOPS␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣=␣"
     << t3 << "/" << fnops_ddot << "/"
           << fnops_ddot/t3/1.0E6 << "." << std::endl;
      std::cout << "WAXPBY␣Time/FLOPS/MFLOPS␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣=␣"
     << t1 << "/" << fnops_waxpby << "/"
           << fnops_waxpby/t1/1.0E6 << "." << std::endl;
      std::cout << "SPARSEMV␣Time/FLOPS/MFLOPS␣␣␣␣␣␣␣␣␣␣␣␣␣␣=␣"
     << t2 << "/" << fnops_sparsemv << "/"
           << fnops_sparsemv/t2/1.0E6 << "." << std::endl;
```

```
    results[0]=t0;
    results[1]=fnops/t0/1.0E6;
    results[2]=t3;
    results[3]=fnops_ddot/t3/1.0E6;
    results[4]=t1;
    results[5]=fnops_waxpby/t1/1.0E6;
    results[6]=t2;
    results[7]=fnops_sparsemv/t2/1.0E6;
    results_log("standard_matrix",nx,num_threads,results);

  return 0;
}
```

```
#ifndef DOT_H
#define DOT_H
double dot(const segVector &x_, const segVector &y_);
#endif
```

```
#include "segVector.h"
#include "omp.h"
#include "computeStartStop.hpp"
#include <assert.h>
double dot(const segVector &x_, const segVector &y_){
 double result = 0.0;
#pragma omp parallel reduction(+:result)
{
 int my_thread_num = omp_get_thread_num();
 int my_start, my_stop;
 double * x = x_.getThreadPointer(my_thread_num);
 double * y = y_.getThreadPointer(my_thread_num);
 computeStartStop(my_thread_num, x_.getNumThreads(), x_.getLength(), my_start,
     my_stop, false);
 int my_length = my_stop-my_start;
 for(int i=0; i<my_length; ++i){
  result+=x[i]*y[i];
 }
}
  return result;
}
```

```
#ifndef WAXPBY_H
#define WAXPBY_H

void waxpby(const segVector &x_, const segVector &y_, const double alpha_, const
    double beta_, const segVector &w_);

#endif
```

```
#include "segVector.h"
#include "omp.h"
#include "computeStartStop.hpp"
#include <cstdio>
#include <iostream>
#include <assert.h>
/*
 *
 */
// Takes segVectors x_, y_, and w_ with scalars alpha and beta and computes
// w_= alpha*x_ + beta*y_
void waxpby (const segVector &x_, const segVector &y_, const double alpha, const
    double beta, const segVector &w_){
#pragma omp parallel
 {
  int my_thread_num = omp_get_thread_num();
  // Range of elements this thread is responsible for
  int my_start, my_stop, dummy;
  computeStartStop(my_thread_num, x_.getNumThreads(), x_.getLength(), my_start,
      my_stop, false);
  int my_length = my_stop-my_start;
  // Pointers to thread local data of x, y, and w
  double * x = x_.getThreadPointer(my_thread_num);
  double * y = y_.getThreadPointer(my_thread_num);
  double * w = w_.getThreadPointer(my_thread_num);
  // Performs waxpby operation on thread local data
  if(alpha == 0.0)
   for(int i=0; i<my_length; ++i)
    w[i] = y[i]*beta;
  else if (alpha == 1.0)
   for(int i=0; i<my_length; ++i)
    w[i] = x[i] + y[i]*beta;
  else if (beta == 0.0)
   for(int i=0; i<my_length; ++i)
    w[i] = x[i]*alpha;
  else if (beta == 1.0)
   for(int i=0; i<my_length; ++i)
    w[i] = x[i]*alpha + y[i];
  else
   for(int i=0; i<my_length; ++i)
    w[i] = x[i]*alpha + y[i]*beta;
 }
}
```

```
#include "segMatrix.h"
#include "segVector.h"
#ifndef MATVEC_H
#define MATVEC_H
```

```
void matvec(const segMatrix &A, const segVector &x, segVector &y, const int option
    );
#endif
```

```
#define _GNU_SOURCE

#include "matvec.h"
#include "assert.h"
#include <omp.h>
#include "computeStartStop.hpp"
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <sched.h>

// y=Ax
void matvec(const segMatrix &A, const segVector &x, segVector &y, const int option
    ){
  //::cout<<"-----MatVec-----"<<std::endl;
  //std::cout<<"Main thread is currently on CPU: " << sched_getcpu() << std::endl;

assert(x.getLength()==y.getLength());
assert(A.getNumCols()==x.getLength());

// Most basic implementation, uses the get() for segMatrix and get() for segVector
if(option==0){
  for(int rows = 0; rows < A.getNumRows(); ++rows){
    double row_result=0.0;
    for(int cols = 0; cols < A.getNumCols(); ++cols){
      row_result+=x.get(cols)*A.get(rows,cols);
    }
    y[rows]=row_result;
  }
}

else if(option==1){

  int num_rows = A.getNumRows();
  int num_threads = A.getNumThreads();

#pragma omp parallel
{
  int my_thread_num = omp_get_thread_num();

//  #pragma omp critical
//    std::cout << "Thread " << omp_get_thread_num() << " is currently on CPU: "
    << sched_getcpu() << std::endl;

  int my_start, my_stop, dummy;
```

```
  computeStartStop(my_thread_num, num_threads, num_rows, my_start, my_stop, false)
      ;
  int row_chunk_size = my_stop - my_start;

  double ** my_row_values = A.getValuesArray(my_thread_num);
  int ** my_row_indices = A.getIndicesArray(my_thread_num);
  int * my_row_nonzeros = A.getNonZerosArray(my_thread_num);

  for(int i =0; i < row_chunk_size; ++i){
    double row_result=0.0;
    for(int j=0; j<my_row_nonzeros[i]; ++j){
      row_result += my_row_values[i][j]*x.get(my_row_indices[i][j]);
    }
    y[i+my_start]=row_result;
  }
}
  }

// segMatrix directly accesses thread local vector values instead of using get()
else if(option==2){

  int num_rows = A.getNumRows();
  int num_threads = A.getNumThreads();

  #pragma omp parallel
  {
    int my_thread_num = omp_get_thread_num();

    int my_start, my_stop, dummy;
    computeStartStop(my_thread_num, num_threads, num_rows,my_start, my_stop, false
        );
    int row_chunk_size = my_stop - my_start;

    double ** my_row_values = A.getValuesArray(my_thread_num);
    int ** my_row_indices = A.getIndicesArray(my_thread_num);
    int * my_row_nonzeros = A.getNonZerosArray(my_thread_num);
    double * my_vector_values = x.getThreadPointer(my_thread_num);
    double * my_y_values = y.getThreadPointer(my_thread_num);
    int row_index;
    int local_vector_index;
    double row_result;
    int row_nonzeros;

    for(int i =0; i < row_chunk_size; ++i){
      if(my_row_nonzeros[i]==0) continue;
      row_result=0.0;
      row_nonzeros = my_row_nonzeros[i];
      for(int j=0; j<row_nonzeros; ++j){
        row_index = my_row_indices[i][j];
        // Check if the vector value is near my thread
```

```
        if(row_index >= my_start && row_index < my_stop){
          local_vector_index = row_index - my_start;
          row_result += my_row_values[i][j]*my_vector_values[local_vector_index];
        }
        // Otherwise use get() function
        else{
          row_result += my_row_values[i][j]*x.get(row_index);
        }
      }
      my_y_values[i]=row_result;
    }
  }

}


}
```

```
#ifndef SEGMATRIX_H
#define SEGMATRIX_H
#include "HPC_Sparse_Matrix.hpp"
#include <omp.h>
#include <iostream>
#include <cstdio>
#include "computeStartStop.hpp"
#include <assert.h>
#include "HPC_Sparse_Matrix.hpp"
#include "segVector.h"
#include <sched.h>

class segMatrix
{
private:
  // Dimensions of Sparse Matrix, often these values are equal
  int num_rows, num_cols;

  // Holds addresses of each thread local 2D array of row values
  double *** values_ptr;

  // Holds addresses of each thread local 2D array of row indices
  int *** indices_ptr;

  // Holds address of each thread local 2D array of pointers to segVector values
  double **** vec_ptr;

  // Holds addresses of each thread local 1D array of row nonzeros
  int ** nonzeros_ptr;
```

```
  // Provides for mapping the get() function to the correct thread local value
  int * thread_id;
  int * offsets;

  int num_threads;

public:
  segMatrix(const HPC_Sparse_Matrix &input_matrix, const int num_threads_);
  double get(const int m_row, const int n_col) const;
  int getNumRows() const;
  int getNumCols() const;
  int getNumThreads() const;
  double** getValuesArray(const int thread_num_) const;
  int** getIndicesArray(const int thread_num_) const;
  int* getNonZerosArray(const int thread_num_) const;
  double *** getVecPtrArray(const int thread_num_) const;
  void printMatrix() const;
};


#endif
```

```
#define _GNU_SOURCE

#include "segMatrix.h"

//#define THREAD_PIN_TEST

segMatrix::segMatrix(const HPC_Sparse_Matrix &input_matrix, const int num_threads_
    ){

#ifdef THREAD_PIN_TEST
  std::cout<<"-----segMatrix␣Creation-----"<<std::endl;
  std::cout<<"Main␣thread␣is␣currently␣on␣CPU:␣" << sched_getcpu() << std::endl;
#endif

  num_rows = input_matrix.total_nrow;
  num_cols = input_matrix.local_ncol;
  assert(num_rows>0);
  assert(num_cols>0);
  num_threads=num_threads_;

  // Holds addresses of each thread local 2D array of row values
  values_ptr = new double**[num_threads];

  // Holds addresses of each thread local 2D array of row indices
  indices_ptr = new int**[num_threads];

  // Holds location information for get() function
```

```
  // thread_id tells for a given row value of the logical matrix which thread it
      is on
  // offsets tells for a given row value of the logical matrix which row of the
      thread local
  // 2D matrix it is on
  nonzeros_ptr = new int*[num_threads];
  thread_id = new int[num_rows];
  offsets = new int[num_rows];

#pragma omp parallel
{
  int my_thread_num = omp_get_thread_num();

#ifdef THREAD_PIN_TEST
  #pragma omp critical
    std::cout << "Thread " << my_thread_num << " is currently on CPU: " <<
        sched_getcpu() << std::endl;
#endif

  int my_start, my_stop, dummy;
  computeStartStop(my_thread_num, num_threads, num_rows, my_start, my_stop, false)
      ;


  // How many rows thread owns
  int row_chunk_size = my_stop - my_start;

  // 1D array holds # non-zeros per row
  int * my_row_nonzeros = new int[row_chunk_size];
  for(int i=0; i < row_chunk_size; ++i){
 my_row_nonzeros[i] = input_matrix.nnz_in_row[my_start+i];
  }
  nonzeros_ptr[my_thread_num] = my_row_nonzeros;

  // Values
  // 2D array holds thread local row values
  double ** my_row_values = new double*[row_chunk_size];
    // Iterates across my rows
  for(int i=0; i < row_chunk_size; ++i){
    // Allocates storage for all non-zero values in row i
    my_row_values[i] = new double[my_row_nonzeros[i]];
    // For all non-zero columns in row i
    for(int j = 0; j < my_row_nonzeros[i]; ++j){
      // Store thread local values and indices from sparse input matrix
      my_row_values[i][j] = input_matrix.ptr_to_vals_in_row[my_start+i][j];
    }
  }
  values_ptr[my_thread_num] = my_row_values;

  // Indices
```

```
  // 2D array holds thread local indices of non-zeros
  int ** my_row_indices = new int*[row_chunk_size];
    // Iterates across my rows
  for(int i=0; i < row_chunk_size; ++i){
    // Allocates storage for all non-zero values in row i
    my_row_indices[i] = new int[my_row_nonzeros[i]];
    // For all non-zero columns in row i
    for(int j = 0; j < my_row_nonzeros[i]; ++j){
      // Store thread local values and indices from sparse input matrix
      my_row_indices[i][j] = input_matrix.ptr_to_inds_in_row[my_start+i][j];
    }
  }
  indices_ptr[my_thread_num] = my_row_indices;

  //// Provides for mapping the get() function to the correct thread local value
  //int offset=0;
  //for(int i = my_start; i < my_stop; ++i){
    //thread_id[i]=my_thread_num;
    //offsets[i]=offset;
    //offset++;
  //}

}
}



// Returns the desired m,n element of the large logical matrix
double segMatrix::get(int m_row, int n_col) const{
  assert(m_row >= 0 && m_row < num_rows);
  assert(n_col >= 0 && n_col < num_cols);

  int thread_num=thread_id[m_row];
  int offset=offsets[m_row];
  double return_value=0.0;

  // Get to desired thread and thread local row
  int* row_indices = indices_ptr[thread_num][offset];
  double* row_values = values_ptr[thread_num][offset];
  int row_nonzeros = nonzeros_ptr[thread_num][offset];

  // Iterate over non-zero indices to see if n_col matches
  for(int col = 0; col < row_nonzeros; ++col){
    if(n_col == row_indices[col]){
      return_value= row_values[col];
    }
  }
  return return_value;
}
```

```
int segMatrix::getNumRows() const{
  return num_rows;
}

int segMatrix::getNumCols() const{
  return num_cols;
}

int segMatrix::getNumThreads() const{
  return num_threads;
}

double** segMatrix::getValuesArray(const int thread_num_) const{
  return values_ptr[thread_num_];
}

int** segMatrix::getIndicesArray(const int thread_num_) const{
  return indices_ptr[thread_num_];
}

int* segMatrix::getNonZerosArray(const int thread_num_) const{
  return nonzeros_ptr[thread_num_];
}

double *** segMatrix::getVecPtrArray(const int thread_num_) const{
  return vec_ptr[thread_num_];
}

void segMatrix::printMatrix() const{
   std::cout << "Printing␣Matrix:␣" << std::endl;
   std::cout << "-------------------------------------------------------" << std::
       endl;
   for(int i=0; i<getNumRows(); ++i){
     for(int j=0; j<getNumCols(); ++j){
       printf("%*.f",3,0,get(i,j));
     }
    std::cout<<std::endl;
}
   std::cout << "-------------------------------------------------------" << std::
       endl;
}
```

```
#include <assert.h>
#ifndef SEGVECTOR_H
#define SEGVECTOR_H

class segVector
{
```

```
private:
 int length;
 int chunk_size;
 int num_threads;
 double ** ptr_array;
 int* thread_id;
 int* offsets;
 double * input_array;
public:
 segVector(int chunk_size_, int num_threads_, bool simple_);
 segVector(int length_, int num_threads_);
 segVector(double * input_array_, int length_, int num_threads_);
 void scale(double multiplier_);
 void test(int diagnostic_level_) const;
 inline double& operator[] (const int n){
  assert(n<length);
  assert(n>=0);
  int thread_num=thread_id[n];
  int offset=offsets[n];
  double* element=ptr_array[thread_num];
  element+=offset;
  return *element;
 }
 int getLength() const;
 double* getThreadPointer(int thread_num_) const;
 int getNumThreads() const;
 int getChunkSize() const;
 void printVector() const;
 inline double get(const int n) const{
  assert(n<length);
  assert(n>=0);
  int thread_num=thread_id[n];
  int offset=offsets[n];
  double* element=ptr_array[thread_num];
  element+=offset;
  return *element;
 }
};


#endif
```

```
#define _GNU_SOURCE

#include <omp.h>
#include <iostream>
#include <cstdio>
#include "computeStartStop.hpp"
#include "segVector.h"
```

```
#include <assert.h>
#include <sched.h>

//#define THREAD_PIN_TEST

// Simple Constructor where n%(n/p)==0
// The bool value is irrelvant and is used only to overload the constructor
segVector::segVector(int chunk_size_, int num_threads_, bool simple_)
{
  chunk_size=chunk_size_;
  num_threads=num_threads_;
  ptr_array=new double*[num_threads]; // Holds address of each threads first
      element

#pragma omp parallel
{
  int my_thread_num = omp_get_thread_num();

  double * my_data = new double[chunk_size]; // Thread local array

  for(int i = 0; i<chunk_size; ++i){ // Initialize local array to all 1.0
    my_data[i] = 1.0;
  }
  ptr_array[my_thread_num] = my_data; // Stores start address of thread local
      array to global pointer array
}
}

// Constructor for when n%(n/p)!=0
// Chunksize is not constant across all threads
// void computeStartStop(int myThreadNum, int numThreads, int loopLength, int
    numGhost, int & myStart, int & myStop,
//  int & numLeftGhost, int & numRightGhost, bool debug)
segVector::segVector(int length_, int num_threads_)
{
#ifdef THREAD_PIN_TEST
  std::cout<<"-----segVector Creation-----"<<std::endl;
  std::cout<<"Main thread is currently on CPU: " << sched_getcpu() << std::endl;
#endif

  length=length_;
  num_threads=num_threads_;
  ptr_array=new double*[num_threads]; // Holds address of each threads first
      element
  thread_id=new int[length];
  offsets=new int[length];


#pragma omp parallel
{
```

```
  int my_thread_num = omp_get_thread_num();

#ifdef THREAD_PIN_TEST
  #pragma omp critical
    std::cout << "Thread␣" << omp_get_thread_num() << "␣is␣currently␣on␣CPU:␣" <<
        sched_getcpu() << std::endl;
#endif


  int my_start, my_stop, dummy;
  computeStartStop(my_thread_num, num_threads, length, my_start, my_stop, false);
  int my_chunk_size = my_stop - my_start;
  double * my_data = new double[my_chunk_size]; // Thread local array

  for(int i = 0; i<my_chunk_size; ++i){ // Initialize local array
    my_data[i] = 1.0;
  }

  int offset=0;
  for(int i = my_start; i < my_stop; ++i){ // Write to thread ID array
    thread_id[i]=my_thread_num;
    offsets[i]=offset;
    offset++;
  }

  ptr_array[my_thread_num] = my_data; // Stores start address of thread local
      array to global pointer array
}
}

/*
* Constructor accepts premade primitive array and initializes a segVector with its
     values
*/
segVector::segVector(double * input_array_, int length_, int num_threads_){

  length=length_;
  num_threads=num_threads_;
  ptr_array=new double*[num_threads]; // Holds address of each threads first
      element
  thread_id=new int[length];
  offsets=new int[length];
  input_array=input_array_;

#pragma omp parallel
{
  int my_thread_num = omp_get_thread_num();
  int my_start, my_stop, dummy;

  computeStartStop(my_thread_num, num_threads, length, my_start, my_stop, false);
```

```
  int my_chunk_size = my_stop - my_start;
  double * my_data = new double[my_chunk_size]; // Thread local array

  for(int i = 0; i<my_chunk_size; ++i){ // Initialize local array to values from
      input array
    my_data[i] = input_array[my_start+i];
  }

  // Write to thread ID array and offset array
  int offset=0;
  for(int i = my_start; i < my_stop; ++i){
    thread_id[i]=my_thread_num;
    offsets[i]=offset;
    offset++;
  }

  ptr_array[my_thread_num] = my_data; // Stores start address of thread local
      array to global pointer array
}
}

// Multiplies entire array by 'multiplier_'

void segVector::scale(double multiplier_)
{
#pragma omp parallel
{
  double my_multiplier=multiplier_;
  int my_thread_num = omp_get_thread_num();
  double * my_data = ptr_array[my_thread_num];

  for(int i = 0; i<chunk_size; ++i){
    my_data[i]=my_multiplier*my_data[i];
  }

}
}

// Diagnostic function:
// Level 0: Prints first element of each threads local array
// Level 1: Prints all the elements of each threads local array

void segVector::test(int diagnostic_level_) const{
  std::cout << "-----------------------------------------------------" << std::
      endl;
  if(diagnostic_level_==0){

    for(int i = 0; i < num_threads; ++i){
      std::cout << "First␣value␣of␣thread␣" << i << "␣" << "is:␣" << *ptr_array[i]
          << std::endl;
```

```
    }
  }
  std::cout << "-------------------------------------------------------" << std::
      endl;
}

void segVector::printVector() const{
  std::cout << "Printing␣Vector:␣" << std::endl;
   std::cout << "-------------------------------------------------------" << std::
       endl;
   for(int i = 0; i < getLength(); ++i){
    std::cout << get(i) << std::endl;
  }
   std::cout << "-------------------------------------------------------" << std::
       endl;

}

int segVector::getLength() const{
  return length;
}

double* segVector::getThreadPointer(int thread_num_) const{
  assert(thread_num_ >= 0 && thread_num_ < num_threads);
  return ptr_array[thread_num_];
}

int segVector::getNumThreads() const{
  return num_threads;
}

int segVector::getChunkSize() const{
  return chunk_size;
}
```

```
void computeStartStop(int myThreadNum, int numThreads, int loopLength, int &
    myStart, int & myStop, bool debug);
```

```
#include <iostream>
#include <omp.h>
void computeStartStop(int myThreadNum, int numThreads, int loopLength, int &
    myStart, int & myStop, bool debug) {
  int myChunkSize = loopLength/numThreads;                                  // n/p
  int chunkRemainder = loopLength%numThreads;                               // when n
      %(n/p) != 0 remainder need be distributed
  if (myThreadNum<chunkRemainder) {                                         //
      Distribute one element per thread starting with first thread
    myChunkSize++;
```

```
    myStart = myThreadNum * myChunkSize;                                    // If a
        thread gets a remainder its previous
                                                                            // thread
                                                                                had a
                                                                                remainder
                                                                                 =>
                                                                                start
                                                                                position

                                                                                changes
  }
  else {
    // This figures out the position of the last chunk that received a remainder
        element
    // and updates the start position of all threads that didn't receive an extra
        item
    myStart = chunkRemainder*(myChunkSize+1) + (myThreadNum-chunkRemainder)*
        myChunkSize;
  }
  myStop = myStart+myChunkSize;                                             // How far
      to go based on your length

  return;
}
```

```
//@HEADER
// ************************************************************************
//
//               HPCCG: Simple Conjugate Gradient Benchmark Code
//                 Copyright (2006) Sandia Corporation
//
// Under terms of Contract DE-AC04-94AL85000, there is a non-exclusive
// license for use of this work by or on behalf of the U.S. Government.
//
// This library is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; either version 2.1 of the
// License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful, but
// WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
// USA
```

```
// Questions? Contact Michael A. Heroux (maherou@sandia.gov)
//
// ************************************************************************
//@HEADER
#ifndef MYTIMER_H
#define MYTIMER_H
double mytimer(void);
#endif // MYTIMER_H
```

```

//@HEADER
// ************************************************************************
//
//               HPCCG: Simple Conjugate Gradient Benchmark Code
//                 Copyright (2006) Sandia Corporation
//
// Under terms of Contract DE-AC04-94AL85000, there is a non-exclusive
// license for use of this work by or on behalf of the U.S. Government.
//
// This library is free software; you can redistribute it and/or modify
// it under the terms of the GNU Lesser General Public License as
// published by the Free Software Foundation; either version 2.1 of the
// License, or (at your option) any later version.
//
// This library is distributed in the hope that it will be useful, but
// WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
// Lesser General Public License for more details.
//
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
// USA
// Questions? Contact Michael A. Heroux (maherou@sandia.gov)
//
// ************************************************************************
//@HEADER

/////////////////////////////////////////////////////////////////////////

// Function to return time in seconds.
// If compiled with no flags, return CPU time (user and system).
// If compiled with -DWALL, returns elapsed time.

/////////////////////////////////////////////////////////////////////////
#ifdef USING_MPI
#include <mpi.h> // If this routine is compiled with -DUSING_MPI
                 // then include mpi.h
double mytimer(void)
```

```
{
   return(MPI_Wtime());
}


#elif defined(UseClock)

#include <time.hpp>
double mytimer(void)
{
   clock_t t1;
   static clock_t t0=0;
   static double CPS = CLOCKS_PER_SEC;
   double d;

   if (t0 == 0) t0 = clock();
   t1 = clock() - t0;
   d = t1 / CPS;
   return(d);
}

#elif defined(WALL)

#include <cstdlib>
#include <sys/time.h>
#include <sys/resource.h>
double mytimer(void)
{
   struct timeval tp;
   static long start=0, startu;
   if (!start)
   {
      gettimeofday(&tp, NULL);
      start = tp.tv_sec;
      startu = tp.tv_usec;
      return(0.0);
   }
   gettimeofday(&tp, NULL);
   return( ((double) (tp.tv_sec - start)) + (tp.tv_usec-startu)/1000000.0 );
}

#elif defined(UseTimes)

#include <cstdlib>
#include <sys/times.h>
#include <unistd.h>
double mytimer(void)
{
   struct tms ts;
   static double ClockTick=0.0;
```

```
   if (ClockTick == 0.0) ClockTick = (double) sysconf(_SC_CLK_TCK);
   times(&ts);
   return( (double) ts.tms_utime / ClockTick );
}

#else

#include <cstdlib>
#include <sys/time.h>
#include <sys/resource.h>
double mytimer(void)
{
   struct rusage ruse;
   getrusage(RUSAGE_SELF, &ruse);
   return( (double)(ruse.ru_utime.tv_sec+ruse.ru_utime.tv_usec / 1000000.0) );
}

#endif
```