2013

The Performance of Data Segmentation of the Conjugate Gradient on a Non-Uniform Memory Access Parallel System

Brandon Hildreth
College of Saint Benedict/Saint John's University

Follow this and additional works at: https://digitalcommons.csbsju.edu/honors_theses

Part of the Computer Sciences Commons

Recommended Citation
https://digitalcommons.csbsju.edu/honors_theses/6

This Thesis is brought to you for free and open access by DigitalCommons@CSB/SJU. It has been accepted for inclusion in Honors Theses, 1963-2015 by an authorized administrator of DigitalCommons@CSB/SJU. For more information, please contact digitalcommons@csbsju.edu.
The Performance of Data Segmentation of the Conjugate Gradient on a Non-Uniform Memory Access Parallel System

An Honors Thesis

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

By: Brandon Hildreth
In Collaboration With: Jacob Hemstad
Advisor: Dr. Mike Heroux

April 2013
Project Title: The Performance of Data Segmentation on a NUMA Parallel System

Approved by:

____________________________________________  __________________________
Michael A. Heroux
Scientist in Residence, Department of Computer Science

____________________________________________  __________________________
Robert Hesse
Associate Professor, Department of Mathematics

____________________________________________  __________________________
Lynn R. Ziegler
Professor, Department of Computer Science

____________________________________________  __________________________
James Schnepf
Chair, Department of Computer Science

____________________________________________  __________________________
Anthony Cunningham
Director, Honors Thesis Program
Abstract

Parallel computing is an ever growing field in computer science that is used in everything from cross-discipline research to consumer goods, and because of this it is important to study the efficiency of parallel systems and methods. This thesis will examine storage and execution techniques of matrix vector operations involved in the Conjugate Gradient Method on a parallel, non-uniform memory access (NUMA) aware system. In order to fully take advantage of the NUMA memory system, various attempts of segmenting the data are explored. This idea of segmenting becomes complicated when we perform matrix vector computations. Because our data is evenly divided among the various regions of memory, the multiplicative vector must be carefully constructed so as to minimize foreign memory accesses for any given thread. Because accessing the multiplicative vector is the most expensive part of the computation, several designs to mitigate this expense are examined.
# Table of Contents

Paralle vs. Sequential Computing ................................................................. 8  
High Performance Computing Hardware Environments .................................. 10  
Applications of HPC ................................................................................ 11  

Our Problem ................................................................................................ 12  

The Conjugate Gradient Method .................................................................. 12  
The Three Functions of the CG ................................................................. 13  
Compressed Row Storage for Sparse Matrices ........................................... 15  
Order of Complexity ............................................................................ 16  
Matrix Vector Computations Make for Ideal Parallel Computations .......... 17  

Modeling NUMA ....................................................................................... 18  

Intro to NUMA ......................................................................................... 18  
Target Machine Specifications .................................................................. 19  
Beast’s NUMA Topology ....................................................................... 20  
The NUMA Effect Demonstrated ............................................................... 21  
NUMA Machine Model ........................................................................... 22  

Mapping Our Problem to NUMA .............................................................. 24  

Segmentation ............................................................................................. 24  
Segmenting Our Data Structures .............................................................. 24  
Matrix/Vector Multiplication .................................................................... 26  
Global Arrays ............................................................................................ 26  
Computing threadID and Offset Locally .................................................... 28  
Region Vector ......................................................................................... 29  
2D Pointer Array ..................................................................................... 29  
Thread Pinning ........................................................................................ 30  

Testing and Results .................................................................................. 31  

Hypothesis ................................................................................................ 31  
Testing Methods ................................................................................... 31  
Results and Analysis ............................................................................ 32  

Conclusions and Future Work ................................................................... 36  

Conclusions ............................................................................................... 36  
Future Work ............................................................................................. 36  

Appendices ............................................................................................... 37  

References ............................................................................................... 37  
Source Code ............................................................................................. 38  

1.0: Best Matrix ....................................................................................... 38  
1.1: best_segMatrix.cpp ........................................................................... 38  
1.2: best_segMatrix.h ............................................................................. 40  
1.3: computeStartStop.cpp ................................................................. 41  
1.4: computeStartStop.hpp ................................................................. 42  
1.5: dot.cpp ............................................................................................ 42
List of Figures

FIGURE 1: CLOCK SPEED VS. YEAR ................................................................. 9
FIGURE 2: CRS EXAMPLE ............................................................................. 16
FIGURE 3: AMD OPTERON 6168 .................................................................. 20
FIGURE 4: NUMA MACHINE MODEL .............................................................. 23
FIGURE 5: SEGMENTATION .......................................................................... 25
FIGURE 6: GLOBAL ARRAYS ......................................................................... 27
FIGURE 7: PROBLEM SIZE 100 ..................................................................... 33
FIGURE 8: PROBLEM SIZE 200 ..................................................................... 34
FIGURE 9: PROBLEM SIZE 300 ..................................................................... 35

List of Tables

TABLE 1: BANDWIDTH NUMA TEST THE Y-AXIS REPRESENTS A REGION DOING THE ACCESSING. THE X-AXIS REPRESENTS A REGION WHERE THE DATA IS BEING HOSTED. .................................................. 21
TABLE 2: LATENCY NUMA TEST THE Y-AXIS REPRESENTS A REGION DOING THE ACCESSING. THE X-AXIS REPRESENTS A REGION THAT IS BEING ACCESSED. .................................................. 22
TABLE 3: MAXIMUM SPEED-UP FOR EACH SEGMENTED IMPLEMENTATION ON PROBLEM SIZE OF 100 .......... 33
TABLE 4: MAXIMUM SPEED-UP FOR EACH SEGMENTED IMPLEMENTATION ON PROBLEM SIZE OF 200 .......... 34
TABLE 5: MAXIMUM SPEED-UP FOR EACH SEGMENTED IMPLEMENTATION ON PROBLEM SIZE OF 300 .......... 35
Parallel vs. Sequential Computing

Parallel computing is a trend in the computing field that is becoming increasingly important and ubiquitous. The idea behind parallel computing is to use multiple computing cores to have a computer do many tasks at the same time. This is in contrast to sequential computing where a computer only has a single core, and thus is only able to do one thing at a time. When we add more processors to a computer, we are essentially giving the computer the ability to multitask. Instead of computing one thing at a time, it now has the capability to compute many things at once. This is parallel computing.

Moore’s Law (which is not really a law, but more of an observation) states that the number of transistors on an integrated circuit will double approximately every eighteen months. Due to the limited physical space on a computer chip, there is only so much of this “doubling” that can happen before transistors can no longer shrink, and the maximum possible amount of transistors are crammed onto a chip. At this point, processors will no longer continue to gain large increases in speed. One method to circumvent this inevitable end of Moore’s Law is to add more processors together. So, instead of trying to make single processors faster and faster, the idea is to add multiple processors together to compute some job. We can see this trend of ceasing the production of faster CPUs in Figure 1 below. CPU clock speed has stagnated in the last few years because the new trend is to utilize many slower, cheaper CPUs. This trend has become so ubiquitous that almost all computers sold today have more than one core. This is how supercomputers are designed as well. Instead of one very fast chip, they consist of lots of chips and together achieve speeds that no single processor ever could. This trend has been growing rapidly for several years, and it is what has allowed us to continue to produce faster computers. Parallel computing has become a necessity in the computing realm.
While parallel computing has allowed us to sustain the core idea behind Moore’s Law (faster computing over time), it should be noted that parallel computing does have its own set of
challenges to overcome. The idea of parallel computing seems simple at first (just keep adding more processors!), but very quickly many complications come up that must be addressed whenever one utilizes more than one core. Possible issues that may arise include locality, granularity, scalability, dependencies, and performance loss due to overhead, non-parallelizable code, contention, and idle cores. Some of these topics will be touched on later in this thesis, but an exhaustive list with explanations of the pitfalls of parallel computing is outside the scope of this paper.

High Performance Computing Hardware Environments

High performance computing (HPC) hardware can vary just as much as the software tools used to implement the parallel applications that are run on the hardware. One such type of HPC hardware is known as symmetric multiprocessing (SMP). This is essentially where there is more than one of the same processing unit present in a system. These processing units will typically have one shared access to memory. This is the type of setup the machine we did our tests on was like, except there are several of these SMP nodes in the system. So each SMP node will have its own access to memory. This idea, and the machine our tests were conducted on, will be discussed at length below. Other notable mentions of HPC hardware environments include GPU computing and distributed computing, though these types of environments are not explored in this research.

If processor technology is half of HPC hardware, the other half would be memory technology. While there are a lot of options for memory and communication in the field of HPC, the two that bear the most relevance to this research are uniform memory access (UMA) and non-uniform memory access (NUMA). If we imagine a machine with many cores, we can either set up memory for these cores in a shared (UMA) or distributed (NUMA) manner. While sharing memory means each element of memory will have equal latency and bandwidth contention for
every core, it will also be generally slower to retrieve data due to bus contention. NUMA is the memory architecture that we looked at and it will be discussed in depth below.

Applications of HPC

Before we proceed any further with the discussion of our research, just as with any scientific endeavor, it is important to understand the relevance of what is being studied. Parallel computing and HPC are similar fields that this research is immersed in. This area of computer science has become ubiquitous over the last few decades in everything including scientific research, industry, and commercial markets. This computing revolution has brought with it unprecedented advances in every field it encounters.

One of the many great things HPC is able to do is operate on and process very large quantities of data, and it does this very quickly (relative to sequential computing). The sort of scientific and industrial projects that HPC is used for today that utilize this large data processing trait could not have even been a consideration just two or three decades ago. Climate research, protein-folding, encryption cracking, simulated nuclear weapons detonation, and supercollider tests are all examples of data heavy projects that are only possible with the power of HPC.

Due to the extreme speeds that parallel computing works at, applications are able to process data much faster and, in a lot of cases, allow users real-time interaction with their programs that they might otherwise not have. This sort of technology exists in almost every newly manufactured computing device. These high computing speeds have literally transformed the way the world functions. Almost all home computers have multiple cores to increase performance, and even the newest mobile devices can come equipped with a dual-core processor chip. The great benefits HPC and parallel computing have to offer are not restrained to academia and industry, they are commonplace in almost all home computing devices as well.
Our Problem

The Conjugate Gradient Method

For this research study, we will be examining the iterative method known as the Conjugate Gradient (CG) method to solve the linear system of $Ax = y$, where $y$ is a known vector, $A$ is a known, sparse matrix, and $x$ is an unknown vector. We are using this computation as a testing ground for our research because this computation is widely used in many applications of HPC.

A thorough proof and understanding of the CG is outside the scope of this thesis, but it is still necessary to understand the operations happening at each step of the algorithm. Below are the steps to the CG method. Note that $A$ is a sparse matrix; $r$, $x$, $p$, and $q$ are vectors; and $\alpha$, $\beta$, $\gamma$, and $\rho$ are scalars. Also, note that $\langle \nu, w \rangle$ is the notation used for the dot product of two vectors.

We begin the CG with an initial guess of $x^{(0)}$ where $r$ is a residual vector.

$$r^{(0)} = y - Ax^{(0)} \quad (1)$$

Then for $i = 1, 2, 3, ...$

$$\rho_{i-1} = \langle r_{(i-1)}, r_{(i-1)} \rangle \quad (2)$$

$$\beta_{i-1} = \frac{\rho_{i-1}}{\rho_{i-2}} \quad (3)$$

$$p_{(i)} = r_{(i-1)} + \beta_{i-1} p_{(i-1)} \quad (4)$$

$$q_{(i)} = Ap_{(i)} \quad (5)$$

$$\gamma_{i} = \langle p_{(i)}, q_{(i)} \rangle \quad (6)$$

$$\alpha_{i} = \frac{\rho_{i-1}}{\gamma_{i}} \quad (7)$$
\[
\begin{align*}
    x_{(i)} &= x_{(i-1)} + \alpha_i p_{(i)} \\ 
    r_{(i)} &= r_{(i-1)} - \alpha_i q_{(i)}
\end{align*}
\]  

(8)  

(9)  

Check if \( r \) is sufficiently small (i.e. \( 1.0\times10^{-6} > r > 0.0 \)), repeat if needed by incrementing \( i \), else end.

This algorithm works by taking an initial guess for \( x \) and then returning a value of \( x \) that is closer to the true \( x \) after each iteration. We continue this process until our residual vector is sufficiently small.

We will be using the Mantevo miniapplication named HPCCG from Sandia National Laboratories for our study. The Mantevo project is a robust software suite designed for HPC, and the miniapp, HPCCG, is an application for the analysis of the Conjugate Gradient on some data set. We will be using much of the source code found in HPCCG, and adding our own as well.

**The Three Functions of the CG**

While finding the solution of our linear system, \( Ax = y \) via the Conjugate Gradient, requires a lot of intermediate steps, our actual computations can be divided into three separate functions. They are the dot product, WAXPBY, and sparse matrix/vector multiplication.

The dot product is just the standard linear algebra computation where we take the sum of the products of two equal length vectors. So, from equation (10) below, to take the dot product of the \( x \) and \( y \) vectors, we multiply each corresponding element from each vector and add the terms into a scalar value \( s \).
\[
\begin{bmatrix}
x_1 & x_2 & \ldots & x_n
\end{bmatrix}
\begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_n
\end{bmatrix}
= x_1y_1 + x_2y_2 + \ldots + x_ny_n = s \quad (10)
\]

WAXPBY is a function that does the linear combination of two vectors \(x\) and \(y\), which are scaled by \(\alpha\) and \(\beta\) respectively. By equation \((11)\), if \(\alpha\) and \(\beta\) are some scalar values, \(x\) and \(y\) are two known vectors, and \(w\) is an unknown vector, then we first scale vector \(x\) by \(\alpha\), and vector \(y\) by \(\beta\). Then, we add our two newly scaled vectors together with vector addition and store the result in \(w\) to give us our resulting vector.

\[
\alpha \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} \alpha x_1 + \beta y_1 \\ \alpha x_2 + \beta y_2 \\ \vdots \\ \alpha x_n + \beta y_n \end{bmatrix} = w \quad (11)
\]

The matrix/vector multiplication operation is by far the most important operation in this method, not only because it is the most computationally heavy, but it is also where many interesting considerations happen during implementation, which will be covered below. For now, we will give a brief overview of matrix/vector multiplication.

We have the matrix \(M\), the known vector \(v\), and the unknown vector \(w\). To multiply \(M\) and \(v\), we multiply each element of \(v\) with the corresponding element in each row of \(M\) and then add each term. So in essence, we perform a dot product of each row of \(M\) with \(v\). This gives us a new resulting vector \(w\).
Notice now that equations (1) and (5) are matrix/vector multiplication functions; (4), (8), and (9) are WAXPBY functions; and (2) and (6) are dot product functions. With this in mind, implementing such a complex algorithm will be much easier. Categorizing almost every step into one of our three core functions also allows us to measure where certain bottlenecks are occurring in the computation.

**Compressed Row Storage for Sparse Matrices**

Because we are working with sparse matrices (matrices with enough zeroes where it becomes a hindrance to keep track of the zeroes), we are able to employ a storage technique called Compressed Row Storage (CRS). This allows us to only store the non-zero values of a sparse matrix, so we don’t need to potentially store thousands of zeros in large matrices. To do this, we only need three arrays to represent our non-zero values: `values`, `indices`, and `numEntries`. The `values` array will hold the actual non-zero values of the matrix. `indices` will hold the column index of each corresponding non-zero value. `numEntries` will store the number of non-zero elements per row. With the combination of these three arrays, one can access every non-zero element in a sparse matrix. An example is given below in Figure 2. Note that `values` and `indices` will be an array of arrays, and `numEntries` will simply be an array of integers.
Though CRS may seem like we are storing a lot more information than if we had simply stored the original sparse matrix, it does indeed give us significant memory storage gains (and it is still quite simple to navigate through the values during computation). Without CRS, we are forced to store all the elements of a matrix, or $n^2$ elements. With CRS, we store $2z+n$ elements, where $z$ is the number of non-zeros and $n$ is the number of rows. To simplify, CRS has linear storage complexity, whereas non-CRS has squared storage complexity. It is quite clear then that CRS can give us huge memory savings for sparse matrices of sufficient size.

**Order of Complexity**

It is also useful for us to reason about the computational complexity of our three core operations. For the dot product, we will have one multiplication per element, or $n$ multiplications. We will also have a total of $n-1$ additions. So, the dot product will have a total of $2n-1$ operations. For WAXPBY, we have a total of $n$ multiplications to scale each vector, or $2n$ multiplications. We then need to add each element of the two vectors together, so $n$ additions. This gives us $3n$ operations. For matrix/vector multiplication we are only concerned with the non-zero values of the matrix because those are the only values we are storing. We will represent these values as $z$. There are $z$ multiplications, and to err on more additions than might be
necessary we will say \( z \) additions. This gives matrix/vector multiplication an approximate total of \( 2z \) operations.

It is also important to reason about the complexity of memory operations. That is, the number of times a process must access an element in memory. The dot product will have a total of \( 2n \) memory accesses; \( n \) for each vector. WAXPBY will have \( n \) reads for each vector and another \( n \) writes for the results vector for a total of \( 3n \) memory operations. Matrix/vector multiplication will have \( z \) reads for each of the non-zero values of the matrix, and \( z \) reads for the multiplicative vector. There will also be \( n \) writes to the results vector. This gives us a total of \( 2z+n \) memory operations. The matrices being used in our computations have roughly 27 non-zero values per row. So for an \( nxn \) matrix, \( 2(27n)+n \) or \( 55n \) memory operations. It is clear then that matrix/vector multiplication will require the most computation time of our functions.

**Matrix Vector Computations Make for Ideal Parallel Computations**

These sorts of computations, matrix and vector manipulations, are ideal structures for parallel computations because they have many independent parts that can be given to the various threads that are running. For example, during matrix/vector multiplication, each row of the matrix is independent from what happens in other rows. Because of this, one could, for example, assign each thread to be responsible for entire rows of the matrix. So, instead of one large matrix that needs to be multiplied, there are now lots of smaller matrices that are all being computed at the same time. In this way, solving linear systems becomes quite fast in parallel, which makes it an ideal tool for parallel computation study.
**Modeling NUMA**

**Intro to NUMA**

NUMA is a design for the main memory of a computing system that attempts to address the issue of bus bandwidth contention to main memory in a parallel system. That is, when there is more than one core present in an UMA system, typically only one of these cores can address memory at a time, leading to the other cores becoming data starved while they wait for the memory bus to clear. NUMA addresses this by dividing memory into regions that are closer to certain cores, and farther from others. This division of memory into regions causes memory regions to become “local” to a core or group of cores. In doing this, different areas of memory can be addressed at the same time; effectively “parallelizing” memory just as adding cores parallelizes computation. This division of memory into regions causes each group of cores to have a different access time to different memory regions, thus the non-uniformity. So for example, if core 1 belongs to NUMA region 1 we can say that core 1 has a unit time of 1 to access data from its own local memory. Now, if core 1 wants data that lives in a remote memory region, or non-local memory, it will have an access time that is greater than a unit time of 1. This is because it is more costly to retrieve data from memory that is non-local. There are several reasons for this increase in cost when going to remote memory such as contention for bandwidth and greater physical distance for the data to travel.

We wish to conduct our tests on a NUMA aware machine because NUMA machines are commonplace. Memory in most everyday computing environments is typically structured in a similar way that was outlined above. It is important for us to fully understand not only the abstract idea of a NUMA memory system, but also the actual topology of a system in one’s testing environment in order to fully capitalize on the performance gains it has to offer.
Target Machine Specifications

Before we can talk about the specific NUMA system we studied, we must first have a rough understanding of the machine and environment we used. We used C++ for all coding. We chose to use the GCC version 4.5.1 compiler. All of our parallel code was performed with OpenMP version 3.0. The Linux kernel version of our target platform was 2.6.35.14-95.fc14.x86_64.

Our target machine that we conducted all of our tests on was the machine named Beast at St. John’s University, Collegeville, Minnesota. Beast is a 48 core, single image machine with eight NUMA domains. Beast has four chip sockets on its motherboard. Each socket holds an AMD Opteron 6168 chip; each of these chips houses 12 physical cores. Each core has its own L1 and L2 cache levels (128KB of L1 cache and 512KB of L2 cache). There are two 6 MB L3 cache banks per chip and each of the L3 caches is shared among 6 of the chip’s cores. A HyperTransport and memory controller manages the transfer of data between the two sides of each chip, and neighboring chips. Figure 3 depicts the topology of the AMD Opteron 6168. Note that the ordering of the cores is accurate. That is, P0 – P5 share an L3 cache as do P6 – P12.
Beast’s NUMA Topology

With the topology of each individual chip in place, we can now describe the more important NUMA regions and topology of Beast. The memory of Beast is divided into 8 different regions, or NUMA domains. The 48 cores of Beast are distributed evenly among each of these regions. For example, from Figure 3, cores P0 - P5 belong to NUMA region 0, cores P6 - P12 belong to NUMA region 1, and so on. This means each grouping of six cores shares an L3 cache, and a NUMA region of main memory. Each region has 8 GB of memory giving Beast a total of 64 GB of memory.
The NUMA Effect Demonstrated

In order to demonstrate this non-uniformity of memory, we conducted several small tests to demonstrate how local memory accesses are faster than remote memory accesses.

Our first test looks at bandwidth contention and how this global resource can severely impact performance when retrieving non-local data. We first create a data structure on a NUMA region, and then have all 48 cores of Beast attempt to access the same data structure at the same time. The result is when the data is hosted in a region local to a given thread the access time is always faster than when trying to compete for bandwidth among the other remote cores. Below in Table 1 are ratios for access time of each region. The ratios are row-relative, so for example, the time it takes Region0 to access data from Region1 is 1.7 times more than the time it takes Region0 to access data from its own local region. Observe that accessing one’s own region of memory is always faster than accessing remote memory. Also worth mentioning is the lack of symmetry here. It is unclear to us why this is the case, and further research will need to be done to understand this.

<table>
<thead>
<tr>
<th></th>
<th>Host 0</th>
<th>Host 1</th>
<th>Host 2</th>
<th>Host 3</th>
<th>Host 4</th>
<th>Host 5</th>
<th>Host 6</th>
<th>Host 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region0</td>
<td>1</td>
<td>1.7</td>
<td>1.4</td>
<td>2.0</td>
<td>1.8</td>
<td>2.1</td>
<td>1.4</td>
<td>2.0</td>
</tr>
<tr>
<td>Region1</td>
<td>1.5</td>
<td>1</td>
<td>1.8</td>
<td>1.6</td>
<td>1.8</td>
<td>1.2</td>
<td>1.8</td>
<td>1.6</td>
</tr>
<tr>
<td>Region2</td>
<td>1.5</td>
<td>1.8</td>
<td>1</td>
<td>1.5</td>
<td>1.2</td>
<td>1.7</td>
<td>1.6</td>
<td>1.8</td>
</tr>
<tr>
<td>Region3</td>
<td>1.7</td>
<td>1.6</td>
<td>1.6</td>
<td>1</td>
<td>1.8</td>
<td>1.6</td>
<td>1.6</td>
<td>1.8</td>
</tr>
<tr>
<td>Region4</td>
<td>1.1</td>
<td>1.7</td>
<td>1.6</td>
<td>1.8</td>
<td>1</td>
<td>1.5</td>
<td>1.6</td>
<td>1.8</td>
</tr>
<tr>
<td>Region5</td>
<td>1.7</td>
<td>1.2</td>
<td>1.8</td>
<td>1.6</td>
<td>1.5</td>
<td>1</td>
<td>1.7</td>
<td>1.2</td>
</tr>
<tr>
<td>Region6</td>
<td>1.5</td>
<td>1.8</td>
<td>1.6</td>
<td>1.8</td>
<td>1.6</td>
<td>1.8</td>
<td>1</td>
<td>1.5</td>
</tr>
<tr>
<td>Region7</td>
<td>1.7</td>
<td>1.6</td>
<td>1.8</td>
<td>1.2</td>
<td>1.8</td>
<td>1.6</td>
<td>1.5</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 1: Bandwidth NUMA Test

The rows represent a region doing the accessing. The columns represent a region where the data is being hosted.
The second test we looked at dealt with the issue of latency. To examine this, we first initialized a data structure on each NUMA region. This time, instead of having every thread access the same data at once, we had each region of cores access a neighboring region exclusively. This way we can more accurately measure the time it takes to get an element from remote memory, without the contention of other cores accessing the same element. While the results are not as stark as the bandwidth test above, it is still clear that accessing your own local memory is faster than accessing remote memory.

<table>
<thead>
<tr>
<th>Region</th>
<th>Region +1</th>
<th>Region +2</th>
<th>Region +3</th>
<th>Region +4</th>
<th>Region +5</th>
<th>Region +6</th>
<th>Region +7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region 0</td>
<td>1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Region 1</td>
<td>1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Region 2</td>
<td>1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Region 3</td>
<td>1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Region 4</td>
<td>1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Region 5</td>
<td>1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Region 6</td>
<td>1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>Region 7</td>
<td>1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
<td>1.1</td>
</tr>
</tbody>
</table>

**Table 2: Latency NUMA Test**

The rows represent a region doing the accessing. The columns represent a region that is being accessed. (Note that the max region is Region 7, so when a region’s cores are accessing another region, there is a modulo 8 operation happening.)

**NUMA Machine Model**

The above tests highlight the “NUMA effect” and give us the basis for the machine model we are working with. Observe that the worst case scenario is a doubling of access time to remote memory. We will assume this is the case for any remote memory access. With this, we can build an abstract machine model for our environment. Figure 4 depicts two nodes, or regions, each with a core and section of main memory that is local to the cores of that region. We
represent unit time with the $\lambda$ character. It costs $1\lambda$ for P0 to access its own local memory, and $2\lambda$ for it to access a remote memory location.

---

**Figure 4: NUMA Machine Model**

Accessing non-local memory is always slower than accessing your own local memory. Because of this fact, one of our primary goals will be to minimize the number of $\lambda$’s incurred from remote memory accesses during computation, and increase local memory accesses. This is a very important concept to understand when working in a parallel, NUMA aware environment, as it has the potential to allow the user to capture performance gains not available to UMA systems. Failure to acknowledge the effects of a NUMA system can also be detrimental to one’s computation.

The question is now, given our problem of the CG outlined above and our awareness of the above NUMA machine model, how can we take advantage of a NUMA parallel system?
Mapping Our Problem to NUMA

Segmentation

With our understanding of an abstract parallel NUMA environment, our goal then is to minimize remote memory accesses and increase local accesses in the hope that this will increase the performance of our computation. With our problem of the CG algorithm in mind, and sparse matrix/vector multiplication being the dominant computation involved, we would like some way of distributing the data so as to maximize local memory accesses and decrease remote memory accesses.

The simplest and most straightforward way to distribute data in such a way would be to give each region of memory the same amount of data. Or stated another way, we don’t want any one region to have a disproportionate amount of data compared to the other regions. Evenly distributed data is not the default method of data distribution for the current HPCCG implementation. The current default distribution of the matrix and vector structures is the “first-touch” rule of OpenMP. In OpenMP, data will live on the memory region corresponding to the thread that “touched” the initialization code for that data first. Put another way, data will live on the NUMA region closest to the thread that did the initialization of that data. Because of this, data can become unevenly distributed across the various regions of memory. This will lead to cores that own less data processing their data the fastest. These cores will either need to retrieve data from non-local memory in order to continue computing or become data starved. To address this, we implement segmented data structures.

Segmenting Our Data Structures

Segmented matrices and vectors are just like normal matrices and vectors except in their initialization and storage. Instead of letting OpenMP distribute these structures by the first-touch principle, we will control their creation and memory allocation so as to evenly distribute the data
to each region. To do this, we will specify the size of our matrix and vectors, and then break them into even chunks for each core to initialize. This way, each core will initialize an equal amount of the data, so all memory regions will hold an equal amount of data. Note that in cases where the number of elements in our matrices and vectors is not evenly divisible by the number of NUMA regions then some regions will have one more element than other regions. This difference is miniscule, but is still an important consideration during implementation.

We start our segmentation by dividing the matrix and vertices into chunks of rows. The row numbers each thread is assigned will be the same across the matrix and vectors. Using Figure 5, we can imagine \( A \) to be a 4x4 matrix and \( x \) and \( y \) to have length 4. Thread 0 will receive rows 0 - 1 of matrix \( A \) and vector \( x \). Thread 0 will also then be responsible for calculating the same rows of \( y \). Thread 1 will receive rows 2 - 3 of \( A \) and \( x \) and will likewise be responsible for the same rows of \( y \). This segmentation will maintain data locality in all our computations except matrix/vector multiplication.

![Figure 5: Segmentation](image-url)
Matrix/Vector Multiplication

When we compute matrix/vector multiplication with the segmentation example outlined above, thread0 will be able to compute coordinates \([0, 0], [0, 1], [1, 0], \text{ and } [1, 1]\) of \(A\) locally because it owns the corresponding values of \(x\). However, for thread0 to compute the other coordinates of \(A\) it is responsible for \(([0, 2], [0, 3], [1, 2], [1, 3])\), it will need to go to thread1’s memory to retrieve the corresponding values of \(x\). This results in remote memory accesses which slow down our overall performance. Our challenge then is how to deal with this. How do we minimize the amount of remote memory accesses? And how can we best cope with necessary remote memory accesses. Below are the four different implementations we looked at to manage the segmented vector \(x\).

Global Arrays

Because we are working with parallel threads, any data that one thread creates will only be accessible to that thread. Threads have no knowledge of what other threads are doing. In order to give all threads access to all elements of the \(x\) vector, a few globally visible arrays will be necessary. We will need a \textit{threadID} array, an \textit{offsets} array, and a \textit{pointers} array. Each of these arrays will be \(n\) elements long where \(n\) is the length of \(x\) except for pointer which will be \textit{numThreads} long (\textit{numThreads} being the total number of threads).
ThreadID will hold the thread number of the thread that owns that corresponding element. So for example, in Figure 6, thread0 will own values 0 – 2 of $x$. Thread0 will then populate the first three values of threadID with its thread number of 0. All threads will do this for their section of threadID.

Offsets will be populated in a similar manner. Offsets will hold the local offset for each element from the local starting element. For example, $x[0]$ has a local offset of 0, $x[4]$ has a local offset of 1, $x[8]$ has a local offset of 2, etc. Again, each thread will populate its own section of offsets.

Pointers will contain pointers to the starting element of each local array of values. Each thread will assign a pointer that points to the first element of that thread’s local section of $x$ to $\text{pointers}[$my_thread_number$]$. 

Figure 6: Global Arrays
Now, let’s suppose thread0 we would like to access \( x[5] \). Thread0 begins by determining what thread that element belongs to. We see threadID[5] returns 1, so \( x[5] \) is not local. Next we use this to grab the first element that thread1 has. This is done with pointers[1]. This gives us the address of \( x[3] \). We then use offsets[5], which returns 2, to increment \( x \) to the correct value of \( x[5] \). We can then access the element at this position.

**Computing threadID and Offset locally**

The global arrays technique of accessing remote memory elements incurs a bit of non-local memory accesses. In an attempt to improve upon this and reduce these non-local accesses, we developed a method for computing the threadID and offset locally. Essentially, by knowing the number of threads involved in the computation and the problem size, it should be possible for each thread to calculate which thread owns any given element, and what that thread local offset is for the element being accessed. This leaves us with a single non-local memory access, the *pointers* array, for any thread attempting to access elements in remote memory.

While this approach seems like it would be an improvement from the three global arrays approach, initial testing revealed it to be much slower than using no segmentation at all. We believe that this is due to the increase in computational complexity. Computing which thread owns a given element, and what that element’s thread local offset will be, involves many more calculations and comparisons than using the three global array approach. Because of bad performance of this implementation, we did not test this method extensively, and have no significant data to report on it.
Region Vector

Our next attempt to reduce the number of remote memory accesses involved some pre-computation. Because each thread would not necessarily be using every value of \( x \) (due to the sparse nature of our matrix), we attempted to build a segmented vector where each thread would precompute the values of \( x \) that they would require for their computation, grab them from remote memory regions, and store them locally for later use. This way, each thread will only be accessing remote memory regions once for each element it will need, instead of many times due to the iterative process of the CG algorithm.

Unfortunately, this approach proved unsuccessful as well. Going back to the steps of the CG algorithm, we see that equation (8) requires us to update \( x \). This means that threads will be required to somehow communicate with each other their new values for \( x \) when \( x \) gets updated locally. But this is in contradiction to our motivation for this regional vector approach. Our goal in this segmented vector design was to minimize remote memory accesses, yet now we are forced to update \( x \) with remote memory accesses during each iteration of the CG algorithm. Because of these extra remote memory accesses required to update \( x \), one can expect this approach to be slow. Initial tests revealed just this. Because of this, there is no relevant data for this segmented vector approach.

2D Pointer Array

For our final approach of managing the placement of data through segmentation, we again aimed to eliminate unnecessary remote memory accesses. This time, we made use of the fact that while our multiplicative vector values are changing, where they live in memory does not. That is, they retain their same memory address throughout the computation.
To take advantage of this, we implement a segmented matrix class that manages a 2D pointer array to all elements. After passing a segmented vector (the multiplicative vector), to our segmented matrix class, each thread is then able to save the values of $\mathbf{x}$ it will need in this 2D pointer array. So, instead of accessing multiple global arrays to retrieve elements from non-local memory, each thread is now accessing one global pointer array during computation.

**Thread Pinning**

One last point should be made before results can be analyzed, which is the idea of thread pinning or thread affinity. For various reasons, operating systems shuffle threads and processes around the various cores of a computer. This thread behavior, known as thread migration, is very detrimental to the performance of most parallel computations. When a thread becomes reassigned to a new core, not only does the relocation itself cost the computation precious time, but all the memory that used to be local to the migrated thread could now be remote resulting in cache misses, and non-local memory accesses. So the data that used to be very close to the migrated thread could now be quite distant, and the thread must now spend time accessing remote memory to localize it in its cache. This is completely contrary to our intentions of keeping segmented data local to the threads that created it. Luckily, the GNU implementation of OpenMP that we used provides us with an environment variable called `GOMP_CPU_AFFINITY` which binds threads to their respective cores. With thread pinning in effect, threads are able to take full advantage of spatial locality (local NUMA regions, various cache levels).
Testing and Results

Hypothesis
To make our expectation explicit, we claim our hypothesis of the following tests to be that explicitly controlling the placement of data via segmentation to increase spatial locality of data for all threads will result in better performance than ignoring data placement and allowing OpenMP to place data through its “first touch” policy.

Testing Methods
To test our various segmentation methods, we utilized the current implementation of HPCCG as the control. As was stated above, the current implementation uses OpenMP’s the first-touch policy of data placement.

We ran several problem sizes for each method. Each tested size correlates to the dimensions of a three dimensional cube. So for a given problem size \( n \), after mapping this three dimensional cube to a two dimensional matrix, the matrix will be of size \( n^3 \times n^3 \). Our vectors then will be of size \( n^3 \). The problem sizes we looked at were 100, 200, and 300. We then varied the number of NUMA regions being used in each problem size by incrementing the number of cores by 6. So for example, the problem size of 100 on the control had 8 tests, each one with 6 more cores than the previous, up to a maximum of 48 cores. This was done for the control, the global arrays (named Standard Matrix), a best case scenario of the global arrays (named Best Matrix), and the 2D pointer array (named Vector Pointer). (The best case test consisted of all local computations. That is, the matrix was constructed in a way so as to have only local computations, and no remote memory accesses.) To achieve good statistical significance, we ran each test ten times to measure the level of uncertainty using the standard deviation.
Results and Analysis

Below are the figures and tables of our results. The figures depict our different implementations on a MegaFLOPS vs. Number of Cores graph. The tables give us the maximum speed-up of each of our segmentation solutions against the control.

Notice that each implementation performs relatively the same compared to the control. In the 100 and 200 problem size cases, the control does slightly worse as we scale up the number of cores used. Our segmented implementations have similar linear speed-up as we increase the cores used.

The only outlier in these figures comes from the Control in the 300 problem size. At 12 cores, we see a dramatic increase in performance. This could be due to the way data is fitting in each memory region. At this problem size, we are working with a lot of data, and it’s possible that some of the data does not completely fit in the first memory region and spills over into the next. This second region could then be taking advantage of this newfound local data and the increased speeds of accessing it.

Also note the degradation of our maximum speed-ups in the 300 problem size. This idea of scalability is a typical problem in parallel computing. As problem sizes increase, it becomes more and more difficult to achieve the same levels of speed-up of smaller problem sizes.
Problem Size 100

Figure 7: Problem Size 100

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Maximum Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Matrix</td>
<td>9.6</td>
</tr>
<tr>
<td>Best Matrix</td>
<td>11.1</td>
</tr>
<tr>
<td>Vector Pointer</td>
<td>9.8</td>
</tr>
</tbody>
</table>

Table 3: Maximum Speed-Up for each Segmented Implementation on Problem Size of 100
Figure 8: Problem Size 200

<table>
<thead>
<tr>
<th></th>
<th>Maximum Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Matrix</td>
<td>10.3</td>
</tr>
<tr>
<td>Best Matrix</td>
<td>10.8</td>
</tr>
<tr>
<td>Vector Pointer</td>
<td>9.9</td>
</tr>
</tbody>
</table>

Table 4: Maximum Speed-Up for each Segmented Implementation on Problem Size of 200
Figure 9: Problem Size 300

Table 5: Maximum Speed-Up for each Segmented Implementation on Problem Size of 300

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Maximum Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Matrix</td>
<td>2.84</td>
</tr>
<tr>
<td>Best Matrix</td>
<td>2.83</td>
</tr>
<tr>
<td>Vector Pointer</td>
<td>3.00</td>
</tr>
</tbody>
</table>
Conclusions and Future Work

Conclusions
Our goal when starting this research was to look at NUMA environments in parallel systems and devise a model of NUMA memory. With this model we could then explore the possibilities of taking advantage of the potential performance gains that come with explicitly controlling the placement of data. Our solution of evenly segmenting the data across each memory region, and the tests we have conducted with our various implementations of this solution, has yielded the following conclusion for this research:

1. Managing the placement of data in a NUMA aware system to maximize data locality for threads via evenly segmenting data across all NUMA regions yields better performance than remaining ignorant to data placement.
2. With the exception of our two failed implementations, the segmentation implementations we developed yielded little performance difference.
3. The process of developing controlled data management solutions and implementations is non-trivial.

Future Work
Further study can still be done on this subject. Because the system we used had an intricate cache topology, it would be worthwhile to develop a way to exclude caching effects from our results. This would give us a much purer look at the NUMA effects we studied. Further study could be conducted on different problems. The HPCCG application we looked at is only one problem, and it would be interesting to see how well segmentation holds up under different condition.
Appendices

References


Source Code

1.0: Best Matrix

1.1: best_segMatrix.cpp

```
#include "best_segMatrix.h"

best_segMatrix::best_segMatrix(const HPC_Sparse_Matrix &input_matrix, const int num_threads_)
{
    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 addresses of each thread local 1D array of row nonzeros
    nonzeros_ptr = new int*[num_threads];

    total_nnz = 0;
}

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

    // Iterates across my rows
    for(int i = 0; i < row_chunk_size; ++i){
        // Count number of non-zeros in columns from my_start to my_stop
        int nnz_in_row = 0;
        for(int j = 0; j < input_matrix.nnz_in_row[my_start+i]; ++j){
            if(input_matrix.ptr_to_inds_in_row[my_start+i][j] >= my_start &
                input_matrix.ptr_to_inds_in_row[my_start+i][j] < my_stop)
                {nnz_in_row++;}
        }
        my_row_nonzeros[i] = nnz_in_row;
    }
```
// Prevent race conditions on all threads trying to update this value
#pragma omp atomic
total_nnz+=nnz_in_row;
}

// 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 non-zero values in row i
    my_row_values[i] = new double[my_row_nonzeros[i]];
    int k = 0;
    for(int j = 0; j < input_matrix.nnz_in_row[my_start+i]; ++j){
        if(input_matrix.ptr_to_inds_in_row[my_start+i][j] >= my_start && input_matrix.ptr_to_inds_in_row[my_start+i][j] < my_stop)
            {
            my_row_values[i][k]=input_matrix.ptr_to_vals_in_row[my_start+i][j];
            k++;
            }
    }
}

// 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){
    my_row_indices[i] = new int[my_row_nonzeros[i]];
    int k = 0;
    for(int j = 0; j < input_matrix.nnz_in_row[my_start+i]; ++j){
        if(input_matrix.ptr_to_inds_in_row[my_start+i][j] >= my_start && input_matrix.ptr_to_inds_in_row[my_start+i][j] < my_stop)
            {
            my_row_indices[i][k]=input_matrix.ptr_to_inds_in_row[my_start+i][j];
            k++;
            }
    }
}

// Stores pointers to thread local data on a global structure
values_ptr[my_thread_num] = my_row_values;
indices_ptr[my_thread_num] = my_row_indices;
nonzeros_ptr[my_thread_num] = my_row_nonzeros;
}

int best_segMatrix::getNumRows() const{
    return num_rows;
}

int best_segMatrix::getNumCols() const{
    return num_cols;
```cpp
int best_segMatrix::getNumThreads() const{
    return num_threads;
}

int best_segMatrix::getTotalNNZ() const{
    return total_nnz;
}

double** best_segMatrix::getValuesArray(const int thread_num_) const{
    assert(thread_num_ >= 0 && thread_num_ < num_threads);
    return values_ptr[thread_num_];
}

int** best_segMatrix::getIndicesArray(const int thread_num_) const{
    assert(thread_num_ >= 0 && thread_num_ < num_threads);
    return indices_ptr[thread_num_];
}

int* best_segMatrix::getNonZerosArray(const int thread_num_) const{
    assert(thread_num_ >= 0 && thread_num_ < num_threads);
    return nonzeros_ptr[thread_num_];
}
```

### 1.2: best_segMatrix.h

```cpp
#ifndef BEST_SEGMATRIX_H
#define BEST_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 <stdio.h>

class best_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 addresses of each thread local 1D array of row nonzeros
int ** nonzeros_ptr;

int num_threads;

// How many non-zeros in the entire matrix
int total_nnz;

public:
best_segMatrix(const HPC_Sparse_Matrix &input_matrix, const int num_threads_);
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;
int getTotalNNZ() const;
};
#endif

1.3: computeStartStop.cpp

#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;
}
1.4: computeStartStop.hpp

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

1.5: dot.cpp

```cpp
#include "dot.h"

double dot(const segVector &x_, const segVector &y_){

double result = 0.0;

assert (x_.getLength()==y_.getLength());
assert (x_.getNumThreads()==y_.getNumThreads());

#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;
}
```

1.6: dot.h

```cpp
#ifndef DOT_H
#define DOT_H
#include "omp.h"
#include "computeStartStop.hpp"
#include <assert.h>
#include "segVector.h"

double dot(const segVector &x_, const segVector &y_);
#endif
```
1.7: generate_matrix.cpp

```cpp
//@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)
//@
//@ ************************************************************************
//@
//@ROUTE gen_matrix:read sparse matrix, right hand side, initial guess,
//@ and exact solution (as computed by a direct solver).
//@******************************************************************************/

#include <iostream>
#include <cstdlib>
#include <cstdio>
#include <cassert>
#include "generate_matrix.hpp"

#using std::cout;
#using std::cerr;
#using std::endl;
#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)
{
    #if defined 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;
1.8: generate_matrix.hpp

```cpp
//@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
```

1.9: HPC_Sparse_Matrix.hpp

```cpp
//@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.
//@**************************************************************************
//@HEADER
```
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;
#endif

#endif
```c
int *neighbors;
int *recv_length;
int *send_length;
double *send_buffer;

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

1.10: HPCCG_seg.cpp

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

#define TICK() t0 = mytimer() // Use TICK and TOCK to time a code section
#define TOCK(t) t += mytimer() - t0

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);
best_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); 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);
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);

nites=k;
}
t0=mytimer()-t_begin;

double fniters = nites;
double fnrow = sparse_matrix->total_nrow;
double fnnz = seg_A.getTotalNNZ();
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[8];

std::cout << "Number of iterations = " << nites << ",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("best_matrix",nx,num_threads,results);

return 0;
}

1.11: Makefile

#@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

Simple hand-tuned makefile. Modify as necessary for your environment.
Questions? Contact Mike Heroux (maherou@sandia.gov).

0) Specify compiler and linker:

CXX=icpc -openmp -w -pg
LINKER=icpc -openmp -pg

CXX=g++ -fopenmp -w
LINKER=g++ -fopenmp

1) Specify C++ compiler optimization flags (if any)
Typically some reasonably high level of optimization should be used to enhance performance.

IA32 with GCC:
CPP_OPT_FLAGS = -O3 -funroll-all-loops -malign-double

CPP_OPT_FLAGS = -DWall -O3 -funroll-all-loops -malign-double

2) System libraries: (May need to add -lg2c before -lm)
SYS_LIB = -lm
# 3) Specify name if executable (optional):

TARGET = HPCCG

Derived Quantities (no modification required)

CXXFLAGS = $(CPP_OPT_FLAGS)

LIB_PATHS = $(SYS_LIB)

TEST_CPP = segVector.cpp dot.cpp computeStartStop.cpp waxpby.cpp best_segMatrix.cpp segMatrix.cpp matvec.cpp mytimer.cpp results_log.cpp generate_matrix.cpp HPCCG_seg.cpp

TEST_OBJ = $(TEST_CPP:.cpp=.o)

$(TARGET): $(TEST_OBJ)
   $(LINKER) $(CFLAGS) $(TEST_OBJ) $(LIB_PATHS) -o $(TARGET)

clean:
   @rm -f *.o *~ $(TARGET) $(TARGET).exe

1.12: matvec.cpp

#include "matvec.h"

void matvec(const best_segMatrix &A, const segVector &x, segVector &y)
{
    assert(x.getLength() == y.getLength());
    assert(A.getNumCols() == x.getLength());

    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;

        // Get pointers to thread local data
        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);
```c
int row_index;
int local_vector_index;
double row_result;
int row_nonzeros;

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){
        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;
}

1.13: matvec.h

#ifndef MATVEC_H
#define MATVEC_H
#include "best_segMatrix.h"
#include "segVector.h"
#include "assert.h"
#include <omp.h>
#include "computeStartStop.hpp"
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <ctime>
#include <iostream>
#include <cmath>
#include "mytimer.hpp"
void matvec(const best_segMatrix &A, const segVector &x, segVector &y);
#endif

1.14: mytimer.cpp

//@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.
   *
   * @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.
   *
   * @HEADER
   * ifndef 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);
}

#if 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 );
}
#endif

#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
1.15: mytimer.hpp

```cpp
//@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
#include <cassert>

#ifndef MYTIMER_H
#define MYTIMER_H

double mytimer(void);

#endif // MYTIMER_H
```

1.16: results_log.cpp

```cpp
#include "results_log.h"

int results_log(const char *implementation, int problem_size, int num_threads, double results[]) {

    // Format the results file name
    char filename[256];
    sprintf(filename, "Results/%s_%d_%d.csv", implementation, problem_size, num_threads);
    double total_time = results[0];
    double total_mflops = results[1];
    double ddot_time = results[2];
    double ddot_mflops = results[3];
    double waxpby_time = results[4];
```
double waxpby_mflops = results[5];
double sparsemv_time = results[6];
double sparsemv_mflops = results[7];

// Get date and time
char *time_stamp;
time_t time_now = time(NULL);
time_stamp = ctime(&time_now);
int i = 0;
while(time_stamp[i]!=\0)
i++;
// Remove newline
time_stamp[i-1]=\0;

// Open file for writing
ofstream myfile (filename,ios::app);
if (myfile.is_open()){
    myfile << time_stamp
    << "," << "TOTAL TIME: " << total_time << "," <<"Total MFLOPS : " << "," << total_mflops
    << "," << "DDOT TIME:" << "," << ddot_time << "," << "DDOT MFLOPS : " << "," << ddot_mflops
    << "," << "WAXPBY TIME: " << "," << waxpby_time << "," << "WAXPBY MFLOPS : " << "," " << waxpby_mflops
    << "," << "SPARSEMV TIME: " << "," << sparsemv_time << "," << "SPARSEMV MFLOPS : " << "," << sparsemv_mflops
    << endl;
    myfile.close();
}else{
    cout << "Unable to open file: " << filename << endl;
}
return 0;

1.17: results_log.h

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string>
using namespace std;
#ifndef RESULTS_LOG_H
#define RESULTS_LOG_H

int results_log(const char * implementation, int problem_size, int num_threads, double results[]);

#endif
```cpp
#include <omp.h>
#include <iostream>
#include <cstdio>
#include "computeStartStop.hpp"
#include "segVector.h"
#include "segMatrix.h"
#include <assert.h>
#include "HPC_Sparse_Matrix.hpp"
#include <sched.h>

#define _GNU_SOURCE

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 addresses of each thread local 1D array of row nonzeros
    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);

#pragma omp critical
{
//std::cout << "Thread " << my_thread_num << " up and running." << std::endl;
//std::cout << "my_start " << my_start << " my_stop " << my_stop << std::endl;
//}

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

// 2D array holds thread local row values
double ** my_row_values = new double*[row_chunk_size];

// 2D array holds thread local indices of non-zeros
int ** my_row_indices = new int*[row_chunk_size];

// 1D array holds # non-zeros per row
int * my_row_nonzeros = new int[row_chunk_size];

// Iterates across my rows
for(int i=0; i < row_chunk_size; ++i){

// Number of nonzero entries in row i
my_row_nonzeros[i] = input_matrix.nnz_in_row[my_start+i];

// Allocates storage for all non-zero values in row i
my_row_values[i] = new double[my_row_nonzeros[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){
    my_row_values[i][j] = input_matrix.ptr_to_vals_in_row[my_start+i][j];
    my_row_indices[i][j] = input_matrix.ptr_to_inds_in_row[my_start+i][j];
}
}

// Stores pointers to thread local data on a global structure
values_ptr[my_thread_num] = my_row_values;
indices_ptr[my_thread_num] = my_row_indices;
nonzeros_ptr[my_thread_num] = my_row_nonzeros;

// 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_];
}

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){

1.19: segMatrix.h

```c++
#include "HPC_Sparse_Matrix.hpp"
#ifndef SEGMATRIX_H
#define SEGMATRIX_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 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;
    void printMatrix() const;
};
#endif
```
```cpp
#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 irrelevant 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;
}

//double& segVector::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 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;
}

---

1.21: segVector.h

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

class segVector
{
private:
    int length;
    int chunk_size;

#}
```cpp
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* getThreadId(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

1.22: waxpby.cpp

#include "waxpby.h"
void waxpby (const segVector &x, const segVector &y, const double &alpha, const double &beta, const segVector &w){
    assert(x.getLength() == y.getLength());
    assert(x.getLength() == w.getLength());
    assert(x.getNumThreads() == y.getNumThreads());
    assert(x.getNumThreads() == w.getNumThreads());
    int num_threads = x.getNumThreads();
    int length = x.getLength();
```
#pragma omp parallel
{
  double alpha = alpha_;  
  double beta = beta_;  
  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);  
  double * w = w_.getThreadPointer(my_thread_num);

  computeStartStop(my_thread_num, num_threads, length, my_start, my_stop, false);
  int my_length = my_stop-my_start;

  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;
}

1.23: waxpby.h

#ifndef WAXPBY_H
#define WAXPBY_H
#include "segVector.h"
#include "omp.h"
#include "computeStartStop.hpp"
#include <cstdio>
#include <iostream>
#include <cassert>

void waxpby(const segVector &x, const segVector &y, const double alpha, const double beta, const segVector &w,);

68
#endif

2.0: Standard Matrix

2.1: computeStartStop.cpp

```cpp
#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;
}
```

2.2: computeStartStop.hpp

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

2.3: create_best_matrix.cpp

```cpp
#include "create_best_matrix.h"

// Create sparse matrix only with elements that will need be multiplied
// by NUMA region local vector elements, i.e. populate each threads rows
// from columns my_start to my_stop
void create_best_matrix(int num_rows, int num_cols, HPC_Sparse_Matrix **A, int num_threads) {
    int total_nrow = num_rows;
```
int local_ncol = num_cols;
int *nnz_in_row = new int[total_nrow];
double **ptr_to_vals_in_row = new double*[total_nrow];
int **ptr_to_inds_in_row = new int *[total_nrow];
double **dummy_diag;

#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, 0, my_start, my_stop, dummy, dummy, false);

    int chunk_size = my_stop - my_start;

    for(int i = my_start; i < my_stop; ++i){ // Rows
        nnz_in_row[i] = chunk_size;
        ptr_to_vals_in_row[i] = new double[chunk_size];
        ptr_to_inds_in_row[i] = new int[chunk_size];

        for(int j = my_start; j < my_stop; ++j){ // Columns
            ptr_to_vals_in_row[i][j] = 1.0;
            ptr_to_inds_in_row[i][j] = j;
        }
    }
}

*A = new HPC_Sparse_Matrix;
(*A)->title = 0;
(*A)->start_row = 0;
(*A)->stop_row = 0;
(*A)->total_nrow = total_nrow;
(*A)->total_nnz = 0;
(*A)->local_nrow = 0;
(*A)->local_ncol = local_ncol;
(*A)->local_nnz = 0;
(*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;
}

2.4: create_best_matrix.h

#ifndef CREATE_BEST_MATRIX_H
#define CREATE_BEST_MATRIX_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 <sched.h>

void create_best_matrix(int nrow, int ncol, HPC_Sparse_Matrix **A);

#endif

2.5: create_matrix.cpp

#include "create_matrix.h"
void create_matrix(int nrow, int ncol, HPC_Sparse_Matrix **A){
    int total_nrow = nrow;
    int local_ncol = ncol;
    int *nnz_in_row = new int[total_nrow];
    double **ptr_to_vals_in_row = new double*[total_nrow];
    int  **ptr_to_inds_in_row = new int *[total_nrow];
    double **dummy_diag;

    for(int i = 0; i < nrow; ++i){ // Rows
        nnz_in_row[i] = ncol-i;
        ptr_to_vals_in_row[i] = new double[ncol-i];
        ptr_to_inds_in_row[i] = new int[ncol-i];

        for(int j = 0; j < ncol - i; ++j){ // Columns
            ptr_to_vals_in_row[i][j] = 1.0;
            ptr_to_inds_in_row[i][j] = j;
        }
    }

    *A = new HPC_Sparse_Matrix;
    (*A)->title = 0;
    (*A)->start_row = 0;
    (*A)->stop_row = 0;
    (*A)->total_nrow = total_nrow;
    (*A)->total_nnz = 0;
    (*A)->local_nrow = 0;
    (*A)->local_ncol = local_ncol;
    (*A)->local_nnz = 0;
    (*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;
}

}
2.6: create_matrix.h

```cpp
#ifndef CREATE_MATRIX_H
#define CREATE_MATRIX_H
#include "HPC_Sparse_Matrix.hpp"

void create_matrix(int nrow, int ncol, HPC_Sparse_Matrix **A);

#endif
```

2.7: dot.cpp

```cpp
#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;
}
```

2.8: dot.h

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

2.9: generate_matrix.cpp

```cpp
//@HEADER
// ************************************************************************
```


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

```c
#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<<" 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;
}

2.10: generate_matrix.hpp

//@HEADER
//@*************************************************************************
//@
2.11: HPC_Sparse_Matrix.hpp

```cpp
#include <mpi.h>
#include "HPC_Sparse_Matrix.hpp"

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

```cpp
#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 //ifndef
```
// 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

#endif // HPC_SPARSE_MATRIX_H


2.12: HPCCG_seg.cpp

```cpp
#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);
n = n + 1;
if (n == N) break;
}

t0 = mytimer();

double fniters = n;
double fnrow = sparse_matrix->total_nrow;
double fnz = sparse_matrix->total_nnz;
double fnops_ddot = fniters * 4 * fnrow;
double fnops_waxpby = fniters * 6 * fnrow;
double fnops_sparsemv = fniters * 2 * fnz;
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 << " w  ObjectMapper 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 = " << fniters << ", " << std::endl;
std::cout << "Final residual = " << normr << ", " << std::endl;
std::cout << "************ Performance Summary (times in sec) ************" << 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;
2.13: Makefile

```plaintext
#@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

# Simple hand-tuned makefile. Modify as necessary for your environment.
# Questions? Contact Mike Heroux (maherou@sandia.gov).
#
#
# 0) Specify compiler and linker:

#CXX=icpc -openmp -w -pg
#LINKER=icpc -openmp -pg

CXX=g++ -fopenmp -w
LINKER=g++ -fopenmp

# 1) Specify C++ compiler optimization flags (if any)
#  Typically some reasonably high level of optimization should be used to
#  enhance performance.
```
#IA32 with GCC:
CPP_OPT_FLAGS = -O3 -funroll-all-loops -malign-double

#CPP_OPT_FLAGS = -DWALL -O3 -funroll-all-loops -malign-double

# 2) System libraries: (May need to add -lg2c before -lm)
SYS_LIB = -lm

# 3) Specify name if executable (optional):
TARGET = HPCCG

######################## Derived Quantities (no modification required) ########################

CXXFLAGS = $(CPP_OPT_FLAGS)
LIB_PATHS = $(SYS_LIB)

TEST_CPP = segVector.cpp dot.cpp computeStartStop.cpp waxpby.cpp segMatrix.cpp matvec.cpp mytimer.cpp generate_matrix.cpp HPCCG_seg.cpp results_log.cpp

TEST_OBJ = $(TEST_CPP:.cpp=.o)

$(TARGET):
$(TEST_OBJ)
$(LINKER) $(CFLAGS) $(TEST_OBJ) $(LIB_PATHS) -o $(TARGET)

clean:
@rm -f *.o *~ $(TARGET) $(TARGET).exe

2.14: matvec.cpp

#define _GNU_SOURCE
#include "matvec.h"
#include "assert.h"
#include <omp.h>
#include "computeStartStop.hpp"
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <sched.h>

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

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;
}

2.15: matvec.h

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

```cpp
#ifdef USING_MPI
#include <mpi.h>  // if this routine is compiled with -DUSING_MPI
    // then include mpi.h

double mytimer(void)
{
    return(MPI_Wtime());
}
#endif

#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

2.17: mytimer.hpp

//@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

2.18: results_log.cpp

#include "results_log.h"
int results_log(const char *implementation, int problem_size, int num_threads, double results[]){

// Format the results file name
char filename[256];
sprintf(filename, "Results/%s_%d_%d.csv", implementation, problem_size, num_threads);
double total_time = results[0];
double total_mflops = results[1];
double ddot_time = results[2];
double ddot_mflops = results[3];
double waxpby_time = results[4];
double waxpby_mflops = results[5];
double sparsemv_time = results[6];
double sparsemv_mflops = results[7];

//@ Get date and time
char *time_stamp;
time_t time_now = time(NULL);
time_stamp = ctime(&time_now);
int i = 0;
while (time_stamp[i]!="\0")
i++;
//@ Remove newline
time_stamp[i-1]="\0";

//@ Open file for writing
ofstream myfile (filename, ios::app);
if (myfile.is_open()){
myfile << time_stamp << "TOTAL TIME: " << total_time << "," << "Total MFLOPS : " << total_mflops;
myfile << "DDOT TIME:" << ddot_time << "," << "DDOT MFLOPS : " << ddot_mflops;
myfile << "WAXPBY TIME: " << waxpby_time << "," << "WAXPBY MFLOPS : " << waxpby_mflops;
myfile << "SPARSEMV TIME: " << sparsemv_time << "," << "SPARSEMV MFLOPS : " << sparsemv_mflops;
myfile << endl;
myfile.close();
}else{
  cout << "Unable to open file: " << filename << endl;
}
return 0;

2.19: results_log.h

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string>
using namespace std;
ifndef RESULTS_LOG_H
ifndef RESULTS_LOG_H
endif
endif

#define RESULTS_LOG_H

int results_log(const char * implementation, int problem_size, int num_threads, double results[]);

#endif

2.20: runtest.pbs

#!/bin/sh

#

# This is an example script for the qsub command

# Usage examples:

#

# Run test_HPCCG 50*50*50

# qsub -o qsub_files -e qsub_files -m n -v NXYZ=200,NUM_THREADS=48 runtest.pbs

#

# Torque configuration parameters

# To get email when your job starts and stops, delete ONE of the # marks below and enter your email address

###PBS -M youraddress@csbsju.edu

###PBS -m abe

# Issue your qsub command from the directory where you want the job to execute.

# This script will automatically cd to that directory.

# cd to run directory

cd $PBS_O_WORKDIR

# Load module for correct compilation environment

source /etc/profile.d/modules.sh

# Choices are openmpi-x86_64 or intel-x86_64
source /opt/intel/bin/iccvars.sh intel64

module load openmpi-x86_64

# Make executable from scratch.
# If you have already created the executable, you can skip this. BUT MAKE SURE YOU HAVE THE RIGHT BUILD!
make clean
make

#run command: Pick the serial version or MPI version
# This is the serial version (uncomment by removing the ### characters)
### echo ============= Running program test_HPCCG $NXYZ $NXYZ $NXYZ =============
### ./test_HPCCG $NXYZ $NXYZ $NXYZ

# This is the OpenMP version
export GOMP_CPU_AFFINITY="0-47"
# echo $GOMP_CPU_AFFINITY
echo ============= Running program HPCCG $NXYZ with $NUM_THREADS threads =============
./HPCCG $NXYZ $NUM_THREADS

# This is the MPI version
### echo ============= Running program test_HPCCG $NUM_MPI_PROCS $NXYZ $NXYZ $NXYZ =============
### mpirun -np $MPI_NP ./test_HPCCG $NXYZ $NXYZ $NXYZ

2.21: script.sh

#!/bin/sh
echo BE SURE TO DO A MAKE CLEAN/MAKE BEFORE HAND.
for (( i = 1 ; i <= 10 ; i=i+1 ))
do
  for (( size=50 ; size <= 300 ; size=size+50 ))
  do

91
if (size == 50); then
  s=10
fi
if (size == 100); then
  s=20
fi
if (size == 150); then
  s=30
fi
if (size == 200); then
  s=60
fi
for (( threads=6; threads <= 48; threads=threads+2 ))
do
  qsub -o qsub_files -e qsub_files -m n -v NXYZ=${size},NUM_THREADS=${threads} runtest.pbs
  sleep $s
done
done
done

2.22: segMatrix.cpp

#define _GNU_SOURCE
#include "segMatrix.h"

#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
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_vals_in_row[my_start+i][j];
  }
}
indices_ptr[my_thread_num] = my_row_indices;
// 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{
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("%.3f", get(i,j));
        }
        std::cout << std::endl;
    }
    std::cout << "-----------------------------------------" << std::endl;
}

2.23: segMatrix.h

#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

2.24: segMatrix_test.cpp

#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <omp.h>
#include "segVector.h"
#include "dot.h"
#include "waxpby.h"
#include "create_matrix.h"
#include "segMatrix.h"
#include "HPC_Sparse_Matrix.hpp"
int main(int argc, char * argv[]) {

    if (argc!=4) {
        std::cout << "Usage: " << argv[0] << " length numThreads option(0 or 1 or 2)" << std::endl;
        exit(1);
    }
    int num_threads = atoi(argv[2]);
    int length = atoi(argv[1]);
    int option = atoi(argv[3]);
    int num_chunks = num_threads;
    omp_set_num_threads(num_threads);

    if(option==3){// Tests segMatrix
        HPC_Sparse_Matrix *sparse_matrix;
        create_matrix(4, 4, &sparse_matrix);
    }
    return 0;
}

2.25: segVector.cpp

#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 irrelevant 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();

    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
}
}
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 << 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;
}
2.26: segVector.h

```c
#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) 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;
        }
        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
```
```cpp
#define _GNU_SOURCE

#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <omp.h>
#include "segVector.h"
#include "dot.h"
#include "waxpby.h"
#include "create_matrix.h"
#include "segMatrix.h"
#include "HPC_Sparse_Matrix.hpp"
#include "matvec.h"
#include <sched.h>

int main(int argc, char * argv[]) {

if (argc != 4) {
    std::cout << "Usage: " << argv[0] << " length numThreads option(0 or 1 or 2 or 3 or 4 or 5 or 6)" << std::endl;
    exit(1);
}

int num_threads = atoi(argv[2]);
int length = atoi(argv[1]);
int option = atoi(argv[3]);
int num_chunks = num_threads;
omp_set_num_threads(num_threads);

if (option == 0) { // Tests dot and waxpby
    std::cout << "-----------------------------------------------------------" << std::endl;
    std::cout << "Option 0: Tests dot and waxpby operations" << std::endl;
    std::cout << "-----------------------------------------------------------" << std::endl;
    segVector segX(length, num_threads);
    segVector segY(length, num_threads);
    segVector segW(length, num_threads);
    std::cout << "segX" << std::endl;
    segX.printVector();
    std::cout << "segY" << std::endl;
    segY.printVector();
    std::cout << "segW" << std::endl;
    segW.printVector();

double alpha = 3.0;
double beta = 3.0;

double dot_product = dot(segX, segY);
std::cout << "Dot product is: " << dot_product << std::endl;

std::cout << "alpha is: " << alpha << std::endl;
std::cout << "beta is: " << beta << std::endl;
```
waxpby(segX, segY, alpha, beta, segW);
std::cout<<"After waxpby of alpha*segX + beta*segY, w is: " << std::endl;
segW.printVector();
}

else if(option==1){// Tests overloaded []
std::cout << "----------------------------------------------------------" << std::endl;
std::cout << "Option 1: Tests overloaded [] of segVector" << std::endl;
std::cout << "----------------------------------------------------------" << std::endl;
segVector seg0(length, num_threads);
segVector seg1(length, num_threads);

for(int i = 0; i<length; ++i){
    std::cout << "Value of " << i << " element: " << seg0[i] << std::endl;
}

else if(option==2){// Tests input array constructor of segVector

std::cout << "----------------------------------------------------------" << std::endl;
std::cout << "Option 2: Tests input array constructor of segVector" << std::endl;
std::cout << "----------------------------------------------------------" << std::endl;

double * test_array = new double[length];

for(int i = 0; i < length; ++i){
    test_array[i]=i*2;
}

segVector seg3(test_array, length, num_threads);
for(int i=0; i<length; ++i){
    std::cout << "Value of seg3 " << i << " element: " << seg3[i] << std::endl;
}

else if(option==3){// tests create_matrix

std::cout << "----------------------------------------------------------" << std::endl;
std::cout << "Option 3: Tests segMatrix and create_matrix" << std::endl;
std::cout << "----------------------------------------------------------" << std::endl;

HPC_Sparse_Matrix *sparse_matrix;
create_matrix(length, length, &sparse_matrix);

std::cout << "New Matrix: " << std::endl;

for(int i=0; i< sparse_matrix->total_nrow; i++){
    int nnz_counter=sparse_matrix->nnz_in_row[i];
    int sparse_index_counter=0;
    for(int j=0; j< sparse_matrix->local_ncol; j++){
        if(nnz_counter>0 && j==sparse_matrix->ptr_toinds_in_row[i][sparse_index_counter]){
            std::cout << sparse_matrix->ptr_tovals_in_row[i][sparse_index_counter] << " ";
            nnz_counter--;
        }
    }
}

else if(option==4){// tests zero
std::cout << "Option 4: Tests create_matrix as Zero, sparse_matrix is: " << std::endl;
/* ...
*/

else if(option==5){
std::cout << "Option 5: Tests input array and constructor of segVector as Zero, segVector is: " << std::endl;
/* ...
*/
sparse_index_counter++;
    }
    else std::cout << "0 ";
    }
    std::cout << std::endl;
}

else if (option == 4) {// Tests segMatrix
    HPC_Sparse_Matrix *sparse_matrix;
    create_matrix(length, length, &sparse_matrix);
    segMatrix seg_matrix(*sparse_matrix, num_chunks);
    int chunk_size = length / num_chunks;
    int num_remainders = length % num_chunks;

    std::cout << "-----------------------------" " << std::endl;
    std::cout << "Option 4: Tests segMatrix" " << std::endl;
    std::cout << "-----------------------------" " << std::endl;

    for (int threads = 0; threads < num_threads; ++threads){
        std::cout << "-----------------------------" " << std::endl;
        std::cout << "Thread number: " " << threads " << std::endl;
        std::cout << "-----------------------------" " << std::endl;

        if(threads < num_remainders){//Have a remainder
            for (int rows = 0; rows < chunk_size + 1; ++rows){// Rows
                int nnz_counter = seg_matrix.getNonZerosArray(threads)[rows];
                int sparse_index_counter = 0;
                for (int cols = 0; cols < seg_matrix.getNumCols(); cols++){
                    if(nnz_counter > 0 && cols == seg_matrix.getIndicesArray(threads)[rows][sparse_index_counter]){  
                        std::cout << seg_matrix.getValuesArray(threads)[rows][sparse_index_counter] << " ";
                        nnz_counter--;
                        sparse_index_counter++;
                    }
                }
                std::cout << "0 ";
            }
            std::cout << std::endl;
        }
        else{ 
            for (int rows = 0; rows < chunk_size; ++rows){// Rows
                int nnz_counter = seg_matrix.getNonZerosArray(threads)[rows];
                int sparse_index_counter = 0;
                for (int cols = 0; cols < seg_matrix.getNumCols(); cols++){
                    if(nnz_counter > 0 && cols == seg_matrix.getIndicesArray(threads)[rows][sparse_index_counter]){  
                        std::cout << seg_matrix.getValuesArray(threads)[rows][sparse_index_counter] << " ";
                        nnz_counter--;
                        sparse_index_counter++;
                    }
                }
            }
        }
        else std::cout << "0 ";
    }
    std::cout << std::endl;
else if(option==5){ // tests get(m,n) for segMatrix
    std::cout << "-----------------------------------------------" << std::endl;
    std::cout << "Option 5: Tests get(m,n) for segMatrix" << std::endl;
    std::cout << "-----------------------------------------------" << std::endl;

    HPC_Sparse_Matrix *sparse_matrix;
    create_matrix(length, length, &sparse_matrix); // Fills HPC_Sparse_Matrix struct
    segMatrix seg_matrix(*sparse_matrix, num_chunks); // Reads in HPC_Sparse_Matrix struct and segments
    std::cout << "New Matrix: " << std::endl;

    for(int i=0; i<seg_matrix.getNumRows(); ++i){
        for(int j=0; j<seg_matrix.getNumCols(); ++j){
            std::cout << seg_matrix.get(i, j) << " ";
        }
        std::cout << std::endl;
    }
}

else if(option==6){ // test matvec
    std::cout << "-----------------------------------------------" << std::endl;
    std::cout << "Option 6: Tests matvec" << std::endl;
    std::cout << "-----------------------------------------------" << std::endl;
    std::cout << "Main thread is currently on CPU: " << sched_getcpu() << std::endl;

    HPC_Sparse_Matrix *sparse_matrix;
    create_matrix(length, length, &sparse_matrix);
    segMatrix A(*sparse_matrix, num_threads);
    //A.printMatrix();
    std::cout << "NOW CREATING segMatrix A" << std::endl;
    segMatrix A(*sparse_matrix, num_threads);
    //A.printMatrix();
    std::cout << "NOW CREATING sedVector x" << std::endl;
    segVector x(length, num_threads);
    //x.printVector();
    std::cout << "NOW CREATING segVector y" << std::endl;
    segVector y(length, num_threads);
    //y.printVector();
    std::cout << "NOW ENTERING matvec" << std::endl;
    matvec(A, x, y, 1);
    //y.printVector();
    std::cout << "Main thread is currently on CPU: " << sched_getcpu() << std::endl;
}

return 0;
#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;
}
}

2.29: waxpby.h

#ifndef WXPBY_H

#include "waxpby.hpp"

#endif
#define WAXPBY_H

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

#endif

2.30: workSteal2_segVector.cpp

#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <cmath>
#include <vector>
#include <omp.h>
#include "segVector.h"

/*
 * Benchmark for vector math operations using primitive arrays on Heap memory for 'values' and a STL vector for 'multipliers'
 */

int main(int argc, char * argv[]) {

    if (argc!=3) {
        std::cout << "Usage: " << argv[0] << " numThreads chunkSize" << std::endl;
        exit(1);
    }

    int numThreads = atoi(argv[1]);
    int chunkSize = atoi(argv[2]);
    int numChunks = numThreads;
    int vecLength = chunkSize*numChunks;
    omp_set_num_threads(numThreads);

    segVector segV0(vecLength, chunkSize, numThreads);
    segVector segV1(vecLength, chunkSize, numThreads);
    segVector segV2(vecLength, chunkSize, numThreads);

    double startTime = omp_get_wtime();
    segV0.scale(2);
    segV1.scale(3);
    segV2.scale(4);

    double stopTime = omp_get_wtime();
    double sampleTime = stopTime - startTime;
    double dataMoved = 3.0*8.0*2.0*vecLength; // 3 vectors, 8 bytes per double, 1 read, 1 write of vecLength
double gBPerSec = dataMoved/sampleTime/1.0e9;

std::cout << "Runtime(sec): " << sampleTime << " Bandwidth(GB/s): " << gBPerSec << std::endl;
return 0;
}

3.0: Vector Pointer

3.1: computeStartStop.cpp

#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;
}

3.2: computeStartStop.hpp

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

3.3: dot.cpp

#include "segVector.h"
#include "omp.h"
#include "computeStartStop.hpp"
```c
#include <assert.h>
double dot(const segVector &x_, const segVector &y_){
    double result = 0.0;
    assert (x_.getLength() == y_.getLength());
    assert (x_.getNumThreads() == y_.getNumThreads());
#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;
}
```

3.4: dot.h

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

3.5: generate_matrix.cpp

```c
//@HEADER
// ************************************************************************
// HPCCG: Simple Conjugate Gradient Benchmark Code
// Copyright (2006) Sandia Corporation
// 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
//@HEADER

#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);
  //
  //  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)

define BEGIN
#include "generate_matrix.hpp"
define END

END define
#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;
                            }
                        }
                    }
            }
        }
}

111
```cpp
*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 << " has " << local_nrow;

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

if (debug) cout << "Process " << rank << " 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_ncol;
(*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;
}

3.6: generate_matrix.hpp

//@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.
```
3.7: HPCCG: Simple Conjugate Gradient Benchmark Code

HPC_Sparse_Matrix.hpp

```c
#include <mpi.h>
#include "HPC_Sparse_Matrix.hpp"

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

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.
// 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
#include "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;

#define 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
3.8: HPCCG_seg.cpp

#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 "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);
  segVector seg_p(length,num_threads);
  //Matrix A
  segMatrix seg_A(*sparse_matrix,num_threads,seg_p);
// segVectors r, p, q, b
segVector seg_r(length,num_threads);
segVector seg_b(b,length,num_threads);
segVector seg_q(length,num_threads);
segVector seg_x(x,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();
waxpb(seg_r,seg_q,1.0,\alpha,seg_r);
TOCK(t1);

nitsers=k;
}

t0=mytimer()-t_begin;

double fniters = nitsers;
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 << " waxpb 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 = " << nitsers << ".
" << std::endl;
std::cout << "********** Performance Summary (times in sec) **********" << std::endl;
std::cout << "Total Time/FLOPS/MFLOPS = " << t0 << " /
" << fnops/t0/1.0E6 << " " << std::endl;
std::cout << " DDOT Time/FLOPS/MFLOPS = " << t3 << " /
" << fnops_ddot/t3/1.0E6 << " " << std::endl;
std::cout << " WAXPBY Time/FLOPS/MFLOPS = " << t1 << " /
" << fnops_waxpby/t1/1.0E6 << " " << std::endl;
std::cout << " SPARSEMV Time/FLOPS/MFLOPS = " << t2 << " /
" << 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("pointer_vector_matrix",nx,num_threads,results);
3.9: Makefile

```plaintext
//@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)
#
# Simple hand-tuned makefile. Modify as necessary for your environment.
# Questions? Contact Mike Heroux (maherou@sandia.gov).
#
# 0) Specify compiler and linker:

#CXX=icpc -openmp -w -pg
#LINKER=icpc -openmp -pg

CXX=g++ -fopenmp -w
LINKER=g++ -fopenmp

# 1) Specify C++ compiler optimization flags (if any)
# Typically some reasonably high level of optimization should be used to
# enhance performance.
```

```
#IA32 with GCC:

```bash
CPP_OPT_FLAGS = -O3 -funroll-all-loops -malign-double
```

# CPP_OPT_FLAGS = -DWALL -O3 -funroll-all-loops -malign-double

# 2) System libraries: (May need to add -lg2c before -lm)

```bash
SYS_LIB = -lm
```

# 3) Specify name if executable (optional):

```bash
TARGET = HPCCG
```

### Derived Quantities (no modification required)

```bash
CXXFLAGS = $(CPP_OPT_FLAGS)
LIB_PATHS = $(SYS_LIB)
TEST_CPP = segVector.cpp dot.cpp computeStartStop.cpp waxpby.cpp segMatrix.cpp matvec.cpp mytimer.cpp generate_matrix.cpp HPCCG_seg.cpp results_log.cpp
TEST_OBJ = $(TEST_CPP:.cpp=.o)

$(TARGET): $(TEST_OBJ) $(LINKER) $(CFLAGS) $(TEST_OBJ) $(LIB_PATHS) -o $(TARGET)
```

```bash
clean:
    @rm -f *.o *~ $(TARGET) $(TARGET).exe
```

### 3.10: matvec.cpp

```bash
#define _GNU_SOURCE
#include "matvec.h"
#include "assert.h"
#include <omp.h>
#include "computeStartStop.hpp"
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <sched.h>
#include "mytimer.hpp"

#define TICK() t0 = mytimer() // Use TICK and TOCK to time a code section
#define TOCK(t) t += mytimer() - t0

double t0, t1, t2, t3;
```
// 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();

            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;

            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;
            }
        }

    } else if (option == 2) {

    } // 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();

```c
#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_vec_ptrs = A.getVecPtrArray(my_thread_num);
    double * my_y_values = y.getThreadPointer(my_thread_num);
    double row_result;
    int row_nonzeros;
    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){
            row_result += my_row_values[i][j] * (*my_vec_ptrs[i][j][j]);
        }
        my_y_values[i]=row_result;
    }
}
```

3.11: matvec.h

```c
#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
```

3.12: mytimer.cpp

```c
//@HEADER
//************************************************************************
//    HPCCG:  Simple Conjugate Gradient Benchmark Code
//    Copyright (2006) Sandia Corporation
//    Under terms of Contract DE-AC04-94AL85000, there is a non-exclusive
```
// Function to return time in seconds.
// If compiled with no flags, return CPU time (user and system).
// If compiled with -DWALL, returns elapsed time.

#include <mpi.h> // if this routine is compiled with -DUSING_MPI
// then include mpi.h
double mytimer(void)
{
    return(MPI_Wtime());
}

#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 );
}

#endif
3.13: mytimer.hpp

```cpp
//@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)
//@
//@ ************************************************************************
//@
#ifndef MYTIMER_H
#define MYTIMER_H

double mytimer(void);
#endif // MYTIMER_H
```

3.14: results_log.cpp

```cpp
#include "results_log.h"
int results_log(const char *implementation, int problem_size, int num_threads, double results[])
{
    // Format the results file name
    char filename[256];
    sprintf(filename, "Results/%s_%d_%d.csv", implementation, problem_size, num_threads);
    double total_time = results[0];
    double total_mflops = results[1];
    double ddot_time = results[2];
    double ddot_mflops = results[3];
    double waxpby_time = results[4];
    double waxpby_mflops = results[5];
    double sparsemv_time = results[6];
    double sparsemv_mflops = results[7];
```
// Get date and time
char *time_stamp;
time_t time_now = time(NULL);
time_stamp = ctime(&time_now);
int i = 0;
while(time_stamp[i]!="\0")
i++;
// Remove newline
time_stamp[i-1]="\0";

// Open file for writing
ofstream myfile (filename, ios::app);
if (myfile.is_open()){
    myfile << time_stamp << "," << "TOTAL TIME: " << total_time << " Total MFLOPS : " << total_mflops
    << "," << "DDOT TIME: " << ddot_time << " DDOT MFLOPS: " << ddot_mflops
    << "," << "WAXPBY TIME: " << waxpby_time << " WAXPBY MFLOPS: " << waxpby_mflops
    << "," << "SPARSEMV TIME: " << sparsemv_time << " SPARSEMV MFLOPS: " << sparsemv_mflops
    << endl;
    myfile.close();
} else{
    cout << "Unable to open file: " << filename << endl;
}
return 0;
}

3.15: results_log.h

#include <iostream>
#include <fstream>
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string>
using namespace std;
#ifndef RESULTS_LOG_H
#define RESULTS_LOG_H

int results_log(const char * implementation, int problem_size, int num_threads, double results[]);

#endif

3.16: segMatrix.cpp
#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;

        // 2D array holds thread local row values
        double ** my_row_values = new double*[row_chunk_size];

        // 2D array holds thread local indices of non-zeros
int ** my_row_indices = new int *[row_chunk_size];

// 1D array holds # non-zeros per row
int * my_row_nonzeros = new int [row_chunk_size];

// Iterates across my rows
for(int i = 0; i < row_chunk_size; ++i){
  // Number of nonzero entries in row i
  my_row_nonzeros[i] = input_matrix.nnz_in_row[my_start+i];

  // Allocates storage for all non-zero values in row i
  my_row_values[i] = new double [my_row_nonzeros[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){
    my_row_values[i][j] = input_matrix.ptr_to_vals_in_row[my_start+i][j];
    my_row_indices[i][j] = input_matrix.ptr_to_inds_in_row[my_start+i][j];
  }
}

// Stores pointers to thread local data on a global structure
values_ptr[my_thread_num] = my_row_values;
indices_ptr[my_thread_num] = my_row_indices;
onzeros_ptr[my_thread_num] = my_row_nonzeros;

// 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++;
}

segMatrix::segMatrix(const HPC_Sparse_Matrix &input_matrix, const int num_threads_, const segVector &p){
  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 addresses of each thread local 1D array of row nonzeros
nonzeros_ptr = new int * [num_threads];

// Holds addresses of each thread local 2D array of pointers to needed
// x vector values
vec_ptr = new double *** [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
//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;

// 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){
        int matrix_column_index = my_row_indices[i][j];
        // Store pointer needed for this row's non-zero column multiplied by
        // vector value
        my_row_vec_ptrs[i][j] = (p.get_pointer(matrix_column_index));
    }
}
vec_ptr[my_thread_num] = my_row_vec_ptrs;

// 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 < 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 << "-" * 80 << 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 << "-" * 80 << std::endl;
3.17: segMatrix.h

```cpp
#ifndef SEGMATRIX_H
#define SEGMATRIX_H
#include "HPC_Sparse_Matrix.hpp"
#include <omp.h>
#include <iostream>
#include <stdio>
#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_);
        segMatrix(const HPC_Sparse_Matrix &input_matrix, const int num_threads_, const segVector &x);
        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;
};
```

132
double *** getVecPtrArray(const int thread_num_) const;
void printMatrix() const;
};
#endif

3.18: segVector.cpp

#define _GNU_SOURCE
#include <omp.h>
#include <iostream>
#include <cstdio>
#include "computeStartStop.hpp"
#include "segVector.h"
#include <cassert.h>
#include <sched.h>

//define THREAD_PIN_TEST

// Simple Constructor where n%(n/p)==0
// The bool value is irrelevant 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

133
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 << "-"*70 << 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;
}

// Returns pointer to vector value n
double* segVector::get_pointer(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;
}

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;
}

3.19: segVector.h

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

#define SEGVECTOR_H

#include <iostream>
#include <assert.h>
#include <vector>
using namespace std;

class segVector{
public:
    segVector()
    {
        // Constructor
    }

    ~segVector()
    {
        // Destructor
    }

    double* get_pointer(int n) const;
    double* getThreadPointer(int thread_num) const;
    int getNumThreads() const;
    int getChunkSize() const;

private:
    int length;
    int num_threads;
    int chunk_size;
    int* offsets;
    int* thread_id;
    double** ptr_array;

    void printVector() const;
};

int main()
{
    segVector s;
    // Use s
    return 0;
}
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;  
    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;
    }
    double * get_pointer(const int n) const;
};

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

/*
*/

void waxpby (const segVector &x_, const segVector &y_, const double alpha_, const double beta_, const segVector &w_)
{
    assert (x_.getLength() == y_.getLength());
    assert (x_.getLength() == w_.getLength());
    assert (x_.getNumThreads() == y_.getNumThreads());
    assert (x_.getNumThreads() == w_.getNumThreads());

    #pragma omp parallel
    {
        double alpha = alpha_;  
        double beta  = beta_;  
        int my_thread_num = omp_get_thread_num();
        int my_start, my_stop, dummy;
        double * x = x_.getThreadPointer(my_thread_num);
        double * y = y_.getThreadPointer(my_thread_num);
        double * w = w_.getThreadPointer(my_thread_num);

        computeStartStop(my_thread_num, x_.getNumThreads(), x_.getLength(), my_start, my_stop, false);
        int my_length = my_stop-my_start;

        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 if (beta == -1.0)
            for(int i=0; i<my_length; ++i)
                w[i] = x[i]*alpha - y[i];

        else if(alpha == -1.0)
            for(int i=0; i<my_length; ++i)
                w[i] = y[i]*beta - x[i];
    }
else
    for(int i=0; i<my_length; ++i)
        w[i] = x[i]*alpha + y[i]*beta;
}

3.21: waxpby.h

#ifndef WAXPBY_H
#define WAXPBY_H

void waxpby(const segVector &x, const segVector &y, const double alpha, const double beta, const segVector &w);

#endif