SANDIA REPORT

SAND2011-6036
Unlimited Release
Printed August 2011

Optimizing Tpetra's Sparse Matrix-Matrix Multiplication Routine

Kurtis L. Nusbaum

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2011-6036
Unlimited Release
Printed August 2011

Optimizing Tpetra's Sparse Matrix-Matrix Multiplication Routine

Kurtis L. Nusbaum
Scalable Algorithms
Sandia National Laboratories
Mailstop 1318
Albuquerque, NM 87185-1318

Abstract

Over the course of the last year, a sparse matrix-matrix multiplication routine has been developed for the Tpetra package. This routine is based on the same algorithm that is used in EpetraExt with heavy modifications. Since it achieved a working state, several major optimizations have been made in an effort to speed up the routine. This report will discuss the optimizations made to the routine, its current state, and where future work needs to be done.

Acknowledgments

Thanks to all of those at Sandia National Labs who helped provide the tools needed to compile this report. A special thanks to Dr. Chris Baker, Dr. Chris Siefert, Dr. Mark Hoemmen, and Dr. Jonathan Hu whose guidance was invaluable. Thanks to Dr. Mike Heroux for funding this project. Thank you to the entire Trilinos community whose help is always appreciated. Thanks to St. John's University for providing resources which helped create the material that this report evaluates.

This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

The format of this report is based on information found in [5].

Contents

Appendix

Figures

Tables

1 Introduction

Over the course of the last year, a sparse matrix-matrix multiplication routine has been developed for the Tpetra [3] package. This routine is based on the same algorithm that is used in EpetraExt [1] with heavy modifications. Since it achieved a working state, several major optimizations have been made in an effort to speed up the routine. This report will discuss the optimizations made to the routine, its current state, and where future work needs to be done.

2 Basic Outline of the Algorithm

The Tpetra sparse matrix-matrix multiply algorithm allows two matrices (A and B) to be multiplied. The result of this multiplication is then placed in a third matrix (C). The basic algorithm is as follows:

1. A' and B' are created from A and B, respectively. If it has been specified that A should be transposed, an actual transpose of the matrix is created and assigned to A'. Otherwise A' is simply set equal to A. The same is done for creating B'.

2. A "view" of A' and a "view" of B' are created. These views simply provide fast access to information that will be needed later in the algorithm. In addition, any imports of off-processor elements are done. Namely, all the rows in B' that contain columns needed by the local copy of A' are imported.

3. The sparsity pattern of C is determined by doing a symbolic multiplication of A' and B', i.e., no actual value computations are done. In this step, column indices for C are computed and used to construct a graph.

4. The actual multiplication of A' and B' is done by iterating through each row of A'. For each row in A', every row in B' is looped through and the appropriate calculations are done.

5. Unless indicated otherwise by the user, `fillComplete` is called on matrix C.

3 Optimizations

3.1 Fixing FillComplete's Sort

As part of its algorithm, the `fillComplete` function relies on a function called `sort2`. This function performs a sort on two arrays by sorting the first array and concurrently doing the same permutations on the second array, i.e., both arrays are sorted according to the ordering of the first array. The main use case for this function is sorting an indices array and moving the values in a values array so that they stay matched up with their associated index.

Up until recently, the `sort2` function relied on an insertion sort algorithm. We modified the function so that it first checks to see if the arrays are already sorted (which happens quite often) and returns right away if they are. If the arrays are not sorted, a quicksort is performed on the arrays. This means that for the section of our sparse matrix-matrix multiply routine where we call `fillComplete` we went from running on average $O(n^2)$ operations to running on average $O(n \log(n))$ operations. It's also worth noting that since our `sort2` function checks first to see if the arrays are already in order, we avoid the worst case runtime for the quicksort routine.

3.2 Removal of Specific Transpose Mode Kernels

In the original ExpetraExt algorithm there were separate multiplication kernels for each possible transpose combination (e.g., $A^T B$, AB^T, and $A^T B^T$). Some of these kernels relied on a function called `find_rows_containing_columns`. The relevant trait to know about this function is that during execution it created an array that was of size $N_c + 2N_p + N_p N_r$, where N_c, N_p, and N_r are the number of local columns, processors, and local rows, respectively. Obviously this is not scalable. At anything but the lowest processor counts, this array quickly balloons to a size that won't fit in memory. We decided to remove this function and the specific transpose kernels. Instead, we explicitly transpose the matrices using Tpetra's `RowMatrixTransposer` if needed and use the $A \times B$ kernel on those matrices. This has *significant*[1] performance benefits when doing operations like $A^T \times B$.

3.3 Streamlining the Graph Building Routine

The original algorithm from EpetraExt used the same function for both building the graph of matrix C and calculating its values by doing two passes through the same function with different data. As a result, on the first pass through the function when the graph was being calculated, the values for matrix C were also calculated, but thrown out. It's not until the second pass through the function that the values calculated are actually inserted into matrix C. We modified this function so that it takes an argument which indicates whether or not

[1] See Section 4 for details

we're just calculating the graph for matrix C. If we're just calculating the graph, all the value calculation is skipped. This saves the routine $O(Numrows \times nnzMax(A) \times nnzMax(B))$ operations where $nnzMax$ is the maximum number of non zeros in the rows of a given matrix.

4 Performance

We conducted a series of weak scaling studies on the Hopper NERSC machine. Hopper is a Cray XE6 with 24 cores per node[2]. We define weak scaling as increasing the problem size while holding the amount of work done on each *node* constant. We define weak scaling efficiency as $Time_{serial}/Time_{parallel}$, where $Time_{serial}$ is the amount of time required to run on a single node[3] and $Time_{parallel}$ is the amount of time to run on N number of nodes.

4.1 Comparison of Development Stages

Figure 1 and Figure 2 show the test results of comparing the Tpetra sparse matrix-matrix multiply routine at various stages of its development. The tests were as such: Two matrices were constructed, each using a 3D Laplace stencil. Each node was assigned 110^3 number of rows from each matrix (this means each core has approximately 55500 rows). We then timed the cost of multiplying these two matrices together. We ran these experiments three times for each node count, and then averaged the results. The red line represents the routine as it was when it first achieved a working state. The blue line represents the routine after the `sort2` algorithm problem had been fixed and we removed the specific transpose mode kernels. And the purple line represents the routine after all optimizations had been applied. As can be clearly seen, the optimizations we applied helped the algorithm substantially. After all of our optimizations were applied, the sparse matrix-matrix multiply routine was running in roughly half the time it was originally.

4.2 Tpetra vs. EpetraExt vs. ML

Figure 3 and Figure 4 show the test results of comparing the Tpetra sparse matrix-matrix multiply routine (with all of its optimizations) to other sparse matrix-matrix multiply routines in Trilinos [4] (namely the original EpetraExt routine and a slightly modified version ML's Epetra sparse matrix-matrix multiply routine[4]). These tests were conducted in the same manner as above. The encouraging thing about these results is that the Tpetra algorithm is as good as if not better than the EpetraExt algorithm at scale. It should be noted that the ML lines are cut short because at the next problem size in our testing, they would fail due to a memory error. Consequently, we could not run tests for greater problem sizes with ML.

[2]For more machine information regarding Hopper, please visit `http://www.nersc.gov/systems/hopper-cray-xe6`

[3]Strictly speaking, $Time_{serial}$ is not actually serial runtime in our tests because a single node on hopper has 24 cores. That said, it's worth noting that none of the testing done below involving Tpetra utilizes any Kokkos [2] kernels. This means that we're not taking advantage of any node-level parallelism.

[4]The only difference between the ML's `Epetra_MatMat_Mult` and the one we used in our test is that we modified the routine to use ML's matrix storage when assembling matrix C rather than Epetra's

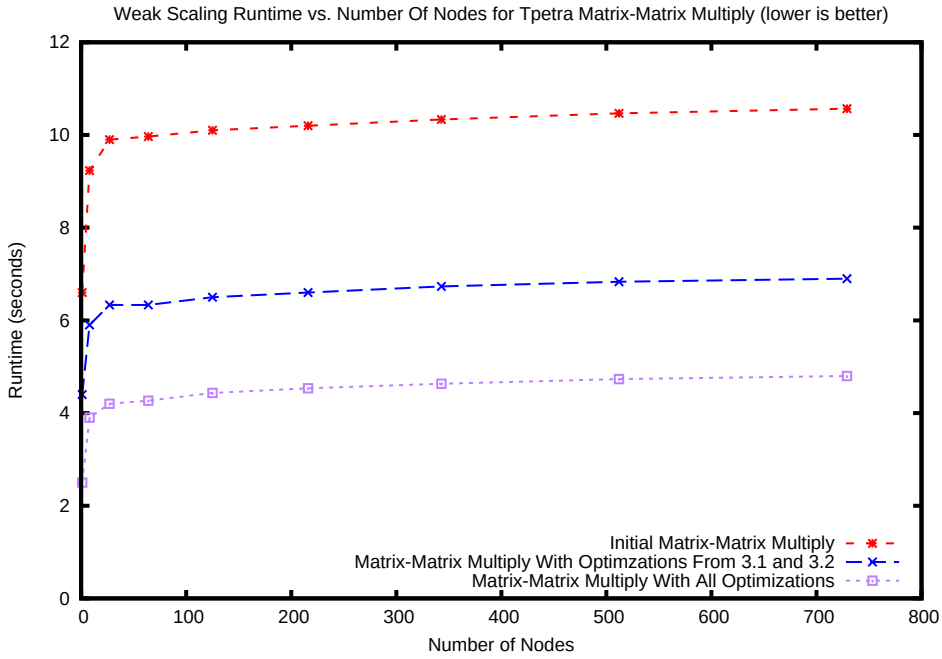

Figure 1. A comparison of the Tpetra sparse matrix-matrix multiply routine's runtime throughout various stages of its development. Note this is a simple $C = A \times B$; so we are running in non-transpose mode.

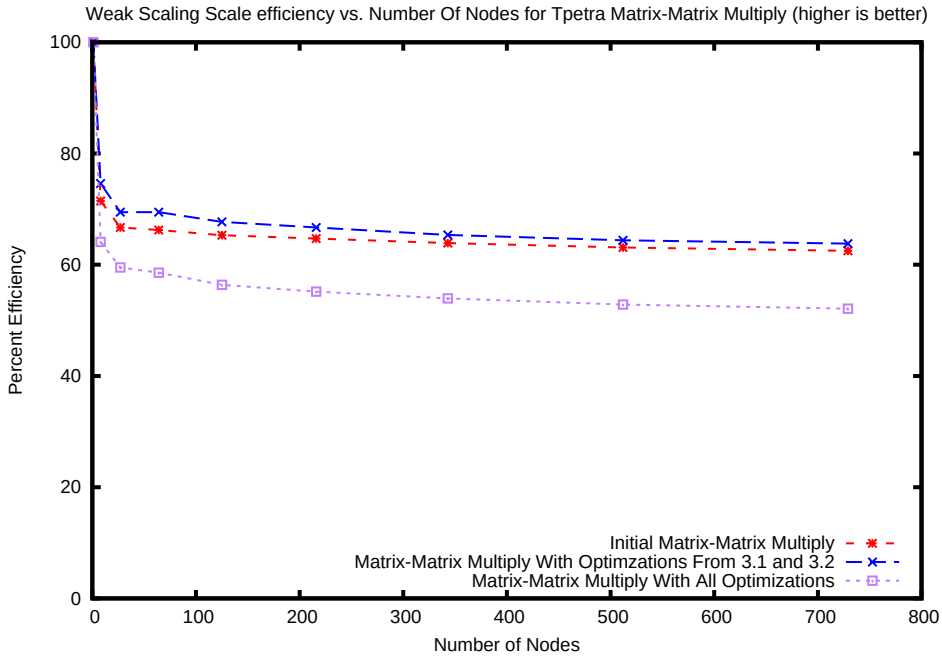

Figure 2. A comparison of the Tpetra sparse matrix-matrix multiply routine's scaling efficiency throughout various stages of its development. Note this is a simple $C = A \times B$; so we are running in non-transpose mode.

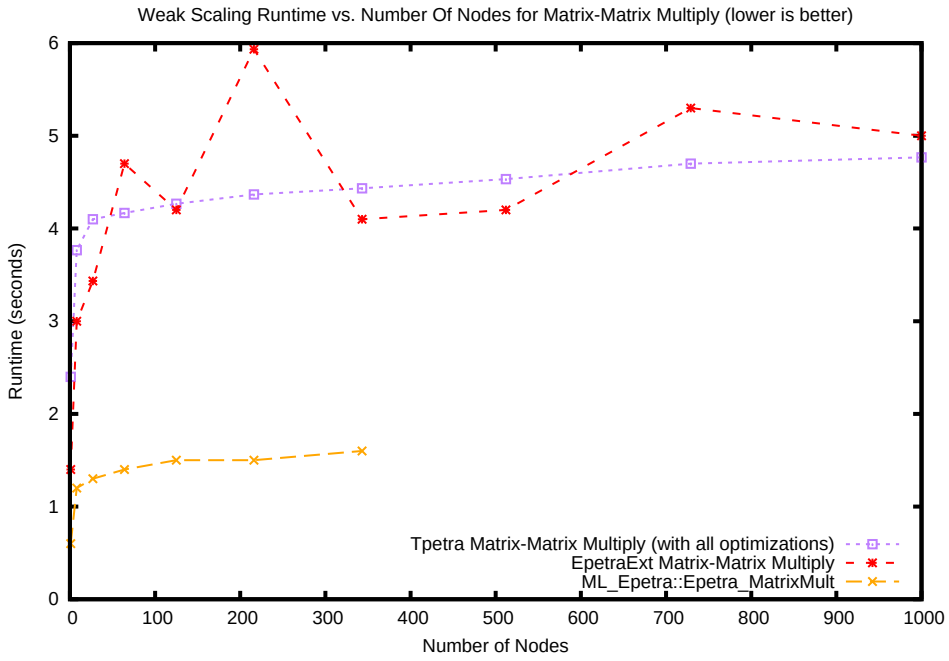

Figure 3. A comparison of various Trilinos sparse matrix-matrix multiply routines' runtime. The ML line is cut short due to a memory error that prevented testing at higher node counts.

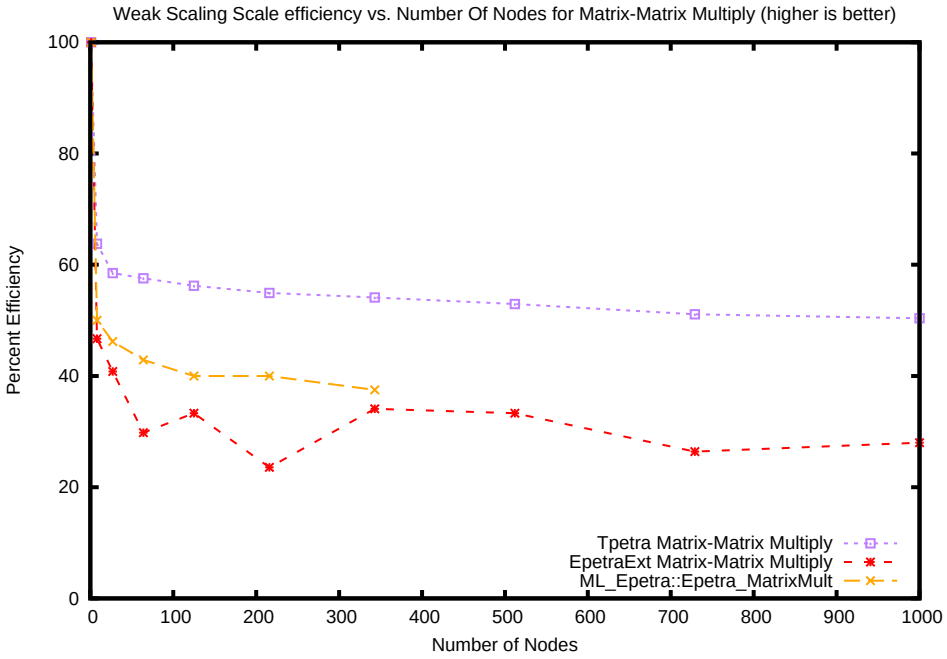

Figure 4. A comparison of various Trilinos sparse matrix-matrix multiply routines' scaling efficiency. The ML line is cut short due to a memory error that prevented testing at higher node counts.

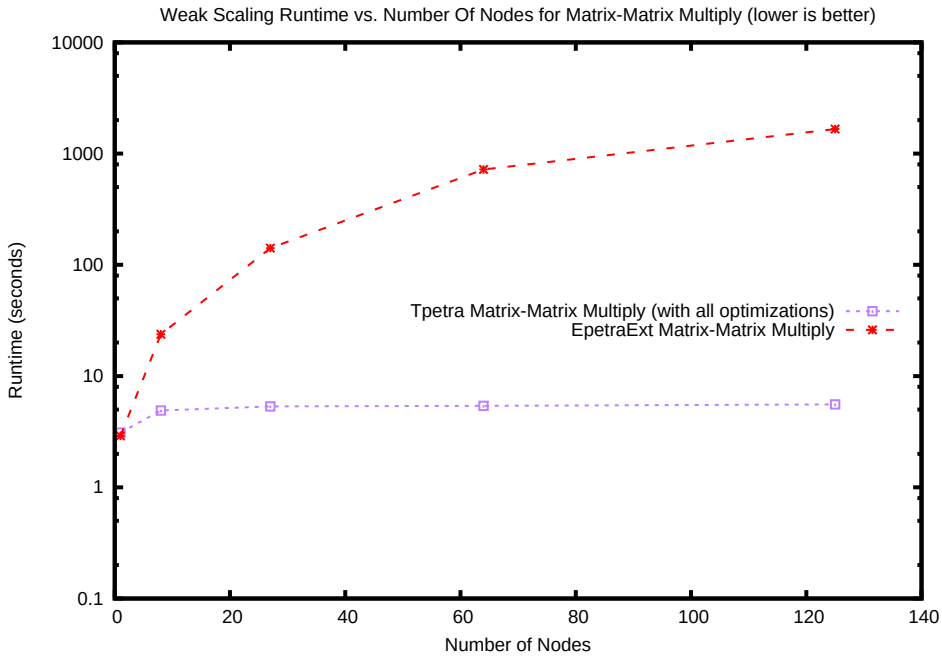

Figure 5. A comparison of sparse matrix-matrix multiple runtime in Tpetra and EpetraExt using transpose mode

4.3 Transpose Mode Tests

Since Tpetra and EpetraExt differ wildly when it comes to transpose modes, we decided to compare the two. We did the same testing procedure as above, except that we requested A be transposed[5]. Figure 5 and Figure 6 show the results. To say that the Tpetra algorithm scales better than the EpetraExt algorithm would be a gross understatement. The Tpetra algorithm levels off at about a 5 second runtime, where as EpetraExt algorithm's runtime seems to grow quadratically. We attempted to conduct a similar experiment where we transposed B instead of A, but the EpetraExt algorithm was taking even longer (over ten minutes on a single node of Hopper) and we didn't want to waste our limited computing resources.

[5]One could first explicitly transpose a matrix and then give it to the ML algorithm which does not have a transpose mode. We did not do this because the point of this test was to compare EpetraExt and Tpetra.

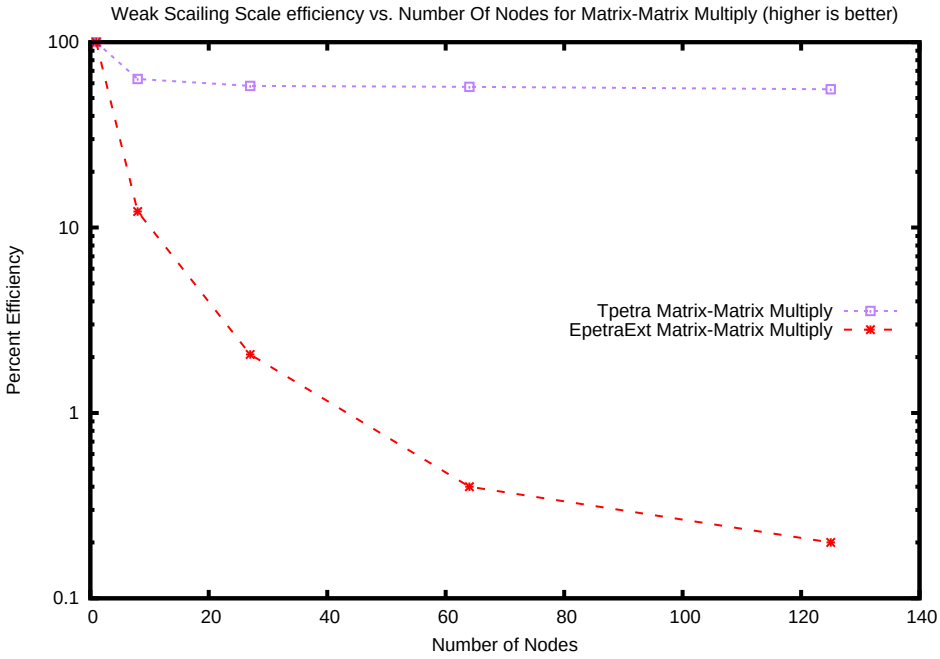

Figure 6. A comparison of sparse matrix-matrix multiple scaling efficiencies in Tpetra and EpetraExt using transpose mode

	Tpetra	EpetraExt	ML
Time (Seconds)	1.4	0.9	0.3

Table 1. Serial Matrix-Matrix Multiplication Run Times

5 Areas for Future Improvement

5.1 Serial Test Motivations

Table 1 shows the results of running three Trilinos-based sparse matrix-matrix multiply algorithms in serial. These tests were conducted identically to the ones in section 4 with the exception of being run on a single *core* of Hopper. Since this test was in serial, all run times should be void of any time that might be attributed to communication. The discrepancy between the EpetraExt and ML has been known for quite some time and is believed by several scientists at Sandia National Labs to be attributed to a hashing scheme employed by the ML algorithm. We believe the discrepancy between Tpetra's and EpetraExt's time has do to with Tpetra's `fillComplete` function since most everything else in this simple test is relatively the same between the Tpetra algorithm and the EpetraExt algorithm. This leads us to believe that by improving Tpetra's underlying architecture (i.e. refining `fillComplete`) and implementing ML's hashing scheme, we should be able to make Tpetra's sparse matrix-matrix multiplication routine run at the same level as ML's.

5.2 Improvement of Underlying Tpetra Architecture

The most important thing that will help the sparse matrix-matrix multiply routine is to improve the implementation of the underlying Tpetra architecture. Undoubtedly the inefficient `sort2` routine we found is not the only problem with Tpetra at large. There have been reports from other scientists at Sandia National Labs that things like the Tpetra Import/Export classes are not running as nearly as fast as their Epetra counterparts. In addition, Tpetra has no performance tests. Consequently, the performance of almost all of tpetra is unknown. These things need to be investigated, and fixed if necessary.

5.3 Possible Implementation of ML's Algorithm

ML's sparse matrix-matrix multiply routine uses a complex hashing scheme in order to speed lookup times for column indices. Overall, ML's sparse matrix-matrix multiplication algorithm is fast, and several scientist's at Sandia National Labs believe this is mainly due to the hashing scheme. Figure 7 outlines, in pseudo code, the algorithm as current Tpetra developers understand it. Implementing this algorithm in the Tpetra sparse matrix-matrix

multiply routine should not be all that difficult and could potentially provide great speed improvements.

5.4 Kokkos Kernel

While not necessarily indicated by the serial tests, writing a Kokkos kernel for the matrix-multiply routine could improve performance. The implementation of such a kernel should be fairly trivial and provide significant speedups by taking advantage of node-level parallelism.

```
1. For matrix B only, create a hashtable where given a globalid,
we get a unique hashtag. i.e. hash[gid] = hashtag
Note that in Serial we don't actually need a "hash", local id's should suffice
2.Create a "reverse" map, one where we can do rmap[hashtag] = gid
3. Allocate an array called acc_index with the same size as the hash table
4. Allocate two arrays called acc_col and acc_val whose size is
equal to the maximum number of row entries in matrix A.
5. For each row i in A{
  acc_index.fill(-1)
  ArrayView cur_A_cols;
  ArrayView cur_A_vals;
  A->getRowView(i, cur_A_cols, cur_A_vals);
  curr_acc_ptr=0;
  for each column k in row A[i]{
    ArrayView cur_B_cols;
    ArrayView cur_B_vals;
    (B or Bimport)->getRowView(k, cur_B_cols, cur_B_vals);
    for(j=0; j< cur_B_cols.size(); ++j){
      cur_acc_index = hashtable(getGlobalElement(cur_B_cols[j])) //precomputing this might be useful
      if(acc_index(cur_acc_index) == -1){
        acc_col[cur_acc_ptr] = cur_acc_index  //Probably should just put in gid actually
        acc_val[cur_acc_ptr] = cur_B_vals[j]*cur_A_vals[k]
        acc_index[curr_acc_index]=cur_acc_ptr++
      }
      else{
        acc_val[acc_index(cur_acc_index)] += cur_B_vals[j]*cur_A_vals[k]
      }
    }
  }
  c.insertGlobalVals(i, (Global_ids_of_hashes(acc_col))(0, curr_acc_ptr), acc_val(0,cur_acc_ptr))
}
```

Figure 7. ML's hash based algorithm for sparse matrix-matrix multiply

6 Contributions

All matrix multiplication code was adapted from EpetraExt for use in Tpetra by Kurtis Nusbaum. Test code was written by Jonathan Hu, Chris Siefert, Jeremie Gaidamour, and Kurtis Nusbaum.

References

[1] M. Heroux. EpetraExt: Linear algebra services package. `http://trilinos.sandia.gov/trilinos/packages/epetraext/`, 2001.

[2] M. Heroux and C. Baker. Kokkos: Core kernels package. `http://trilinos.sandia.gov/trilinos/packages/kokkos`.

[3] M. Heroux, C. Baker, and A. Williams. Tpetra: Next-generation templated Petra. `http://trilinos.sandia.gov/trilinos/packages/tpetra`.

[4] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[5] Tamara K. Locke. Guide to preparing SAND reports. Technical report SAND98-0730, Sandia National Laboratories, Albuquerque, New Mexico 87185 and Livermore, California 94550, May 1998.

A Trilinos Configure Script

Figure A.1 shows the script that was used to configure the build of Trilinos on Hopper that was used for testing.

```
#!/bin/bash
#
# This configure worked for me (Kurtis Nusbaum <klnusbaum@gmail.com>) on
# NERSC's hopper machine as of 21/07/2011.

if [ $# != 0 ]
then
echo ""
echo "NOTE: Before running this configure, I needed to run the "
echo "following module commands. You might need to as well. "
echo "module swap PrgEng-pgi PrgEnv-gnu"
echo "module load git"
echo "module load cmake"
echo "This configure script is also designed for an \"out of source\""
echo "build. You should create a build directory within your TRILINOS_HOME"
echo "and run this script from in there."
echo ""
else
cmake \
-D MPI_CXX_COMPILER="CC" \
-D MPI_C_COMPILER="cc" \
-D MPI_Fortran_COMPILER="ftn" \
-D Teuchos_ENABLE_STACKTRACE:BOOL=OFF \
-D Teuchos_ENABLE_LONG_LONG_INT:BOOL=ON \
-D Trilinos_ENABLE_Tpetra:BOOL=ON \
-D Tpetra_ENABLE_TESTS:BOOL=ON \
-D Tpetra_ENABLE_EXAMPLES:BOOL=ON \
-D Tpetra_ENABLE_EXPLICIT_INSTANTIATION:BOOL=ON \
-D Teuchos_ENABLE_EXPLICIT_INSTANTIATION:BOOL=ON \
-D TPL_ENABLE_MPI:BOOL=ON \
-D CMAKE_INSTALL_PREFIX:PATH="$HOME/opt/Trilinos/tpetraEval" \
-D BLAS_LIBRARY_DIRS="$LIBSCI_BASE_DIR/gnu/lib" \
-D BLAS_LIBRARY_NAMES="sci" \
-D LAPACK_LIBRARY_DIRS="$LIBSCI_BASE_DIR/gnu/lib" \
-D LAPACK_LIBRARY_NAMES="sci" \
-D CMAKE_CXX_FLAGS="-O3 -ffast-math -funroll-loops" \
\
..
fi
```

Figure A.1. Configure script used for Trilinos

DISTRIBUTION:

1	MS 1318	Kurtis Nusbaum, 01426
1	MS 1320	Dr. Michael Heroux, 01426
1	MS 0378	Dr. Christopher Siefert, 01443
1	MS 9159	Dr. Jonathan Hu, 01426
1	MS 1320	Dr. Jeremie Gaidamour, 01426
1	MS 0899	Technical Library, 9536 (electronic copy)

Sandia National Laboratories