Comparison of Krylov Subspace Methods for Solving Linear Systems

Project Status Update

A comparative performance and cost analysis of the RAPtor codebase implementations of conjugate gradient and biconjugate gradient stabilized methods for sparse matrices.

Tanner J. Evans

tannerjevans@unm.edu

Final Project for CS491-001, Parallel Numerical Algorithms

Professor Amanda Bienz

University of New Mexico, Fall 2021

Table of Contents

1. Project Background

Krylov subspace methods use iterative refinement to seek an answer to the linear equation $Ax = b$. These methods exploit functions which, instead of providing an exact solution, tell us how to modify the values of the current iteration to shift the system towards a (hopefully) less inexact result. The methods iterate until a certain tolerance level is reached, at which point the algorithm is said to have converged. If a defined number of iterations is exceeded before this point, the algorithm failed to converge. The tolerance is generally some measure of the change in the result between successive iterations, and it is supposed to indicate an acceptable proximity of the result to the exact solution.

The most popular Krylov subspace method is the conjugate gradient method (CG), which minimizes the residual vector. For well-conditioned matrices, it tends to converge very quickly. Unfortunately, the set of matrices which CG can solve in its most efficient form is limited to either symmetric positive definite or Hermitian[1] positive definite matrices. For matrices which do not satisfy these limitations, we must find another tactic, though these are generally more expensive. For this project, I will be focusing on comparing CG with the biconjugate gradient stabilized method (BiCGStab), which can solve non-symmetric matrices. I will be using the implementations for both of these algorithms found in the RAPtor codebase[2].

I will utilize Tau[3] and its jumpshot utility to compare CD and BiCGStab given consistent tolerance and a variety of matrices and processor counts. Since they have different domains, I will compare the performance of CG and BiCGStab on a set of matrices for which both are effective, so as to establish if BiCGStab's increase in cost over CG is consistent, and therefore reasonably extrapolatable. I had planned on assessing non-symmetric matrices, but I was unable to obtain any of the form I was looking for. I will obtain the matrices to use during this project from the SuiteSparse[4] resource in order to ensure that my assessment of solutions is based on real-world problems.

Sparse matrices emerge from many disciplines. For many, but not all, symmetry is rote. Though in computer science we tend to dismiss as trivial an algorithmic increase in cost of some constant multiple of the optimal cost, in high performance computing, even a small constant factor can be problematic. Where 100 milliseconds instead of one millisecond is insignificant, 100 hours instead of one hour is brutal. High performance computing is not infrequently performed with data at this scale. The assessment of the relative performance of various algorithms is therefore important.

[1] Hermitian matrices involve complex numbers, and I will not be dealing with them in this project.
[2] https://github.com/raptor-library
[3] https://www.cs.uoregon.edu/research/tau/home.php
[4] https://suitesparse-collection-website.herokuapp.com/

2. The Methods

The following algorithms are taken from a lecture note on the BiCGStab algorithm written by Xianyi Zeng for a class at UTEP[5].

2.1. Conjugate Gradient Method (CG)

Algorithm 2.1 Conjugate Gradient (CG)

Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$, $\boldsymbol{p}_0 = \boldsymbol{r}_0$
for $j = 0, 1, \cdots$ **do**
  $\alpha_j = (\boldsymbol{r}_j \cdot \boldsymbol{r}_j)/((A\boldsymbol{p}_j)\cdot \boldsymbol{p}_j)$
  $\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \alpha_j \boldsymbol{p}_j$
  $\boldsymbol{r}_{j+1} = \boldsymbol{r}_j - \alpha_j A\boldsymbol{p}_j$
  if $\|\boldsymbol{r}_{j+1}\| < \varepsilon_0$ **then**
    Break;
  end if
  $\beta_j = (\boldsymbol{r}_{j+1} \cdot \boldsymbol{r}_{j+1})/(\boldsymbol{r}_j \cdot \boldsymbol{r}_j)$
  $\boldsymbol{p}_{j+1} = \boldsymbol{r}_{j+1} + \beta_j \boldsymbol{p}_j$
end for
Set $\boldsymbol{x} = \boldsymbol{x}_{j+1}$

CG uses the residual to seek the solution to a system. On each iteration, it determines the direction and distance to travel in a single dimension to minimize the error in that dimension. Because of this, it is guaranteed to converge in as many steps as there are dimensions in the system, as minimizing the error in each dimension yields the solution, with some rounding error. Its superior speed is thanks to the fact that it never travels along the same dimension twice, and the fact that the calculation used to ensure that this is the case is accomplished in a three-step recurrence relation, which requires far less calculation than A-orthogonalization processes which can handle asymmetry. CG requires only two inner-product (IP) calculations and one sparse matrix-vector (SpMV) calculation per iteration.

2.2. Biconjugate Gradient Stabilized Method (BiCGStab)

Algorithm 2.3 Biconjugate Gradient Stabilized (BICGSTAB)

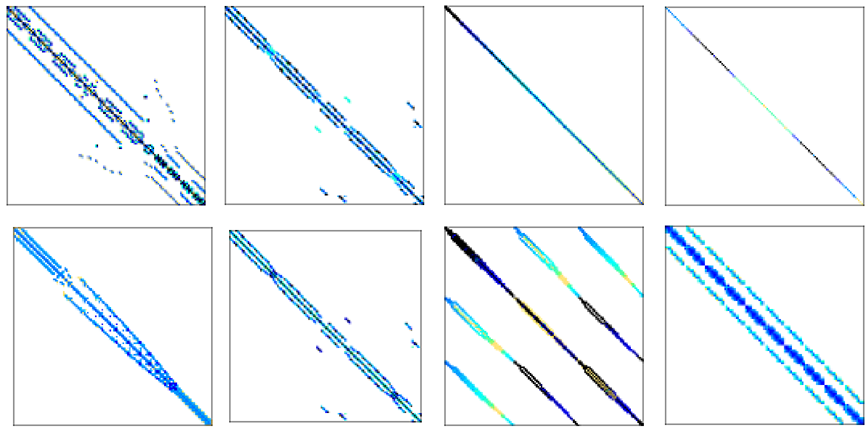
Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$, choose $\boldsymbol{r}'_0$ such that $\boldsymbol{r}_0 \cdot \boldsymbol{r}'_0 \neq 0$
Set $\boldsymbol{p}_0 = \boldsymbol{r}_0$
for $j = 0, 1, \cdots$ **do**
  $\alpha_j = (\boldsymbol{r}_j \cdot \boldsymbol{r}'_0)/((A\boldsymbol{p}_j)\cdot \boldsymbol{r}'_0)$
  $\boldsymbol{s}_j = \boldsymbol{r}_j - \alpha_j A\boldsymbol{p}_j$
  $\omega_j = ((A\boldsymbol{s}_j)\cdot \boldsymbol{s}_j)/((A\boldsymbol{s}_j)\cdot(A\boldsymbol{s}_j))$
  $\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \alpha_j \boldsymbol{p}_j + \omega_j \boldsymbol{s}_j$
  $\boldsymbol{r}_{j+1} = \boldsymbol{s}_j - \omega_j A\boldsymbol{s}_j$
  if $\|\boldsymbol{r}_{j+1}\| < \varepsilon_0$ **then**
    Break;
  end if
  $\beta_j = (\alpha_j/\omega_j) \times (\boldsymbol{r}_{j+1} \cdot \boldsymbol{r}'_0)/(\boldsymbol{r}_j \cdot \boldsymbol{r}'_0)$
  $\boldsymbol{p}_{j+1} = \boldsymbol{r}_{j+1} + \beta_j(\boldsymbol{p}_j - \omega_j A\boldsymbol{p}_j)$
end for
Set $\boldsymbol{x} = \boldsymbol{x}_{j+1}$

BiCGStab works in a fairly similar way, with some additions. It is an improvement on the BiCG algorithm, which calculates each iteration the residual and the residual of the transpose of A, and orthogonalizing the true residual with respect to the residual of A transpose[6]. This method has

[5] Found at https://utminers.utep.edu/xzeng/2017spring_math5330/MATH_5330_Computational_Methods_of_Linear_Algebra_files/ln07.pdf
[6] https://etna.math.kent.edu/vol.1.1993/pp11-32.dir/pp11-32.pdf

some flaws, some of which are resolved by the stabilization introduced in BiCGStab, which smooths and speeds convergence. Unfortunately, BiCGStab requires 4 IP and 2 SpMV per iteration.

3. Source Matrices



1. nos5 | https://suitesparse-collection-website.herokuapp.com/HB/nos5
    a. 5,172 Nonzeros
    b. 468x468
    c. Structural Problem
2. bcsstm12 | https://suitesparse-collection-website.herokuapp.com/HB/bcsstm12
    a. 19,659 Nonzeros
    b. 1,473x1,473
    c. Structural Problem
3. mhd3200b | https://suitesparse-collection-website.herokuapp.com/Bai/mhd3200b
    a. 18,316 Nonzeros
    b. 3,200x3,200
    c. Electromagnetics Problem
4. t3dl_e | https://suitesparse-collection-website.herokuapp.com/Oberwolfach/t3dl_e
    a. 20,360 Nonzeros
    b. 20,360x20,360
    c. Structural Problem
5. ex33 | https://suitesparse-collection-website.herokuapp.com/FIDAP/ex33
    a. 22,189 Nonzeros
    b. 1,733x1,733
    c. Computational Fluid Dynamics Problem
6. bcsstk11 | https://suitesparse-collection-website.herokuapp.com/HB/bcsstk11
    a. 32,241 Nonzeros
    b. 1,473x1,473
    c. Structural Problem
7. plat1919 | https://suitesparse-collection-website.herokuapp.com/HB/plat1919
    a. 32,399 Nonzeros
    b. 1,919x1,919
    c. 2D/3D Problem
8. bscctk06 | https://suitesparse-collection-website.herokuapp.com/HB/bcsstk06
    a. 7,860 Nonzeros
    b. 420x420
    c. Structural Problem

4. Approach

4.1. Processor Counts

I performed runs with 1, 4, 8, 16, 32, and 64 processors.

4.2. Code

```
#!/bin/bash

#PBS -q default
#PBS -l nodes=8:ppn=8
#PBS -l walltime=02:00:00


cd /users/tevans/cs_491_21/sparse_linear_algebra/project
module load netlib-lapack-3.6.1-gcc-4.8.5-x3vu6o3
module load mpich-3.2-gcc-4.8.5-7ebkszx

export PATH=$PATH:/users/tevans/tau-2.30.2/x86_64/bin

CGFILES=$(ls suitesparse/cg_compat/*.pm | xargs -n 1 basename)

PROCS="1 4 8 16 32 64"

LOG="runs/runlog.txt"

export TAU_TRACE=0

echo "" >> $LOG
echo "NEW RUN" >> $LOG

echo "        UNTRACED" >> $LOG

for f in $CGFILES
do
        for n in $PROCS
        do
                echo "" >> $LOG
                echo "STARTING" $f $n "cg" >> $LOG
                echo "start:" >> $LOG
                date +"%Y-%m-%d %H:%M:%S,%N" >> $LOG
                mpirun -n $n ./cg suitesparse/cg_compat/$f >> $LOG
                echo "end:" >> $LOG
                date +"%Y-%m-%d %H:%M:%S,%N" >> $LOG

                echo $f $n "bicgstab" >> $LOG
                echo "start:" >> $LOG
                date +"%Y-%m-%d %H:%M:%S,%N" >> $LOG
                mpirun -n $n ./bicgstab suitesparse/cg_compat/$f >> $LOG
                echo "end:" >> $LOG
                date +"%Y-%m-%d %H:%M:%S,%N" >> $LOG
        done
done

echo "        TRACED" >> $LOG

export TAU_TRACE=1

for f in $CGFILES
do
        for n in $PROCS
        do
                DIR1="runs/$f.dir/cg.$f.$n.procs/"
                DIR2="runs/$f.dir/bicgstab.$f.$n.procs/"
                rm -r $DIR1 &> /dev/null
                mkdir -p $DIR1 &> /dev/null

                echo "" >> $LOG
                echo "STARTING" $f $n "cg" >> $LOG
                echo "start:" >> $LOG
                date +"%Y-%m-%d %H:%M:%S,%N" >> $LOG
                mpirun -n $n tau_exec ./cg suitesparse/cg_compat/$f >> $LOG
                echo "end:" >> $LOG
                date +"%Y-%m-%d %H:%M:%S,%N" >> $LOG

                tau_treemerge.pl > /dev/null
                tau2slog2 tau.trc tau.edf -o tau.cg.$f.$n.slog2 > /dev/null
                mv *.edf *.trc *.slog2 $DIR1

                rm -r $DIR2 &> /dev/null
                mkdir -p $DIR2 &> /dev/null

                echo $f $n "bicgstab" >> $LOG
                echo "start:" >> $LOG
                date +"%Y-%m-%d %H:%M:%S,%N" >> $LOG
                mpirun -n $n tau_exec ./bicgstab suitesparse/cg_compat/$f >> $LOG
                echo "end:" >> $LOG
                date +"%Y-%m-%d %H:%M:%S,%N" >> $LOG

                tau_treemerge.pl > /dev/null
                tau2slog2 tau.trc tau.edf -o tau.noncg.$f.$n.slog2 > /dev/null
                mv *.edf *.trc *.slog2 $DIR2
        done
done
```

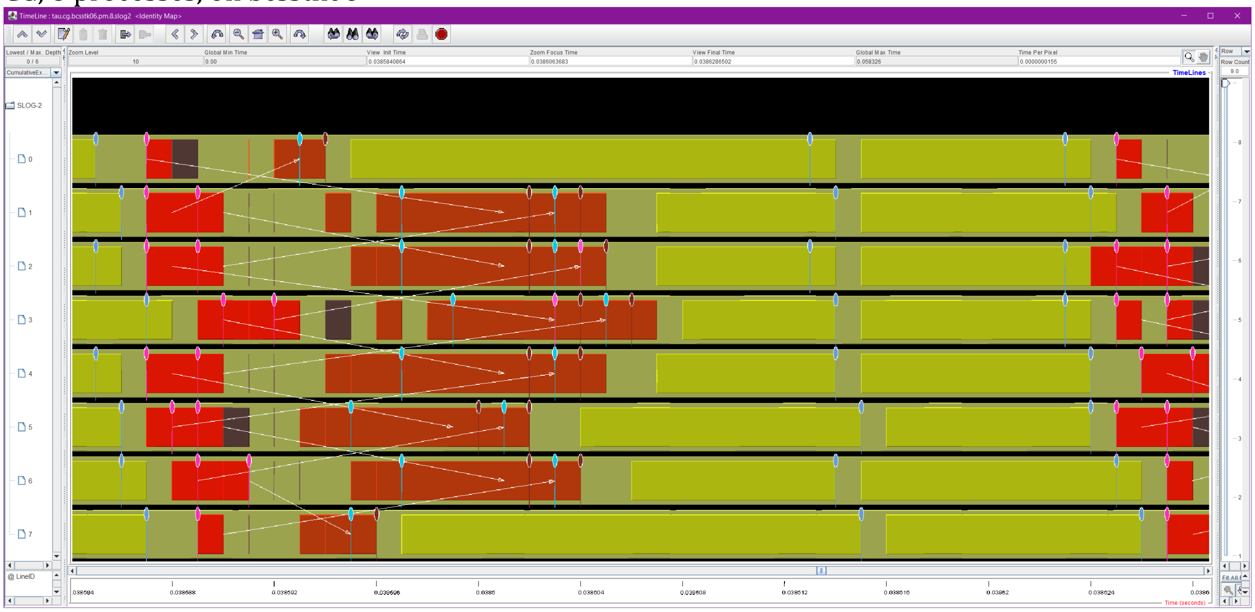
Fortunately, the BiCGStab RAPtor implementation was easily swappable with the CG implementation found in HW5 code. I built a run-script that performed many of the processes dynamically, given that I was running the calculations both traced and untraced on both CG and BiCGStab methods over 6 different processor counts, for a total of 192 different runs, half of which had profiling data that needed to be slogged and moved to organizational folders.

In order to improve outputs to both Wheeler and to my log files, I also diverted unnecessary prints to the void.

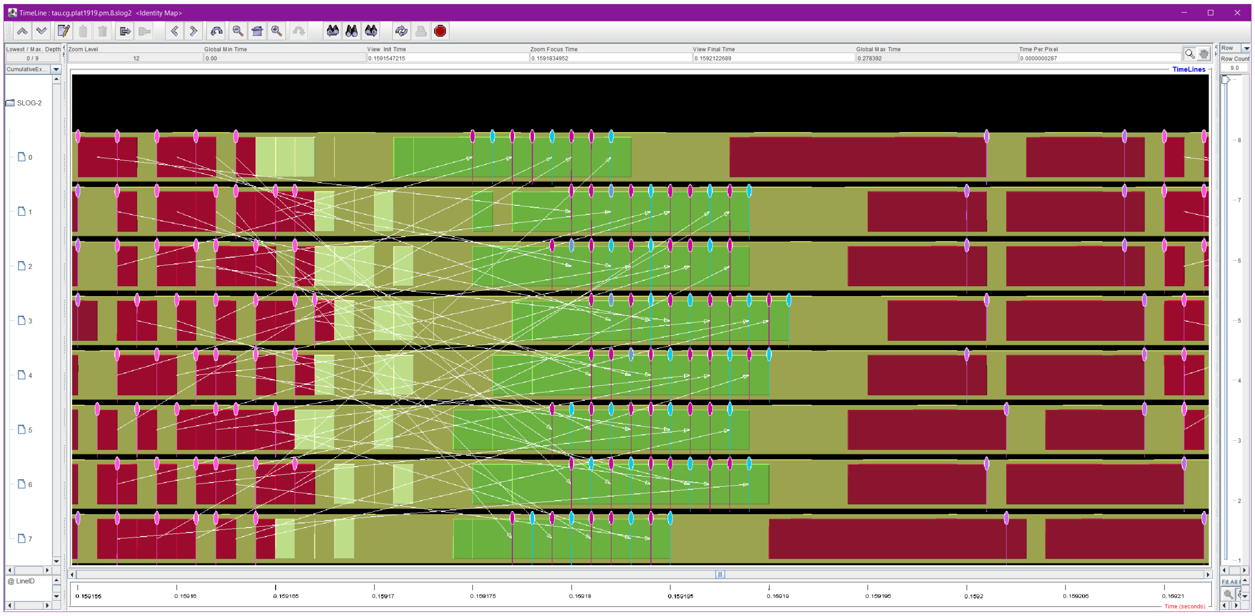
5. Results

I was very ambitious when setting up my code, and I generated absurdly more data than I have the time or resources to tabularize, let alone analyze. I have pulled up the Jumpshots for 8 processes for a couple of the matrices below.

CG, 8 processes, on bcsstk06



BiCGStab, 8 processes, on bcsstk06



BiCGStab, 8 processes, on plat1919

In both sets of profiles, when looking at the per-iteration pattern, we can see the patterns of communication that denote SpMVs and IPs.

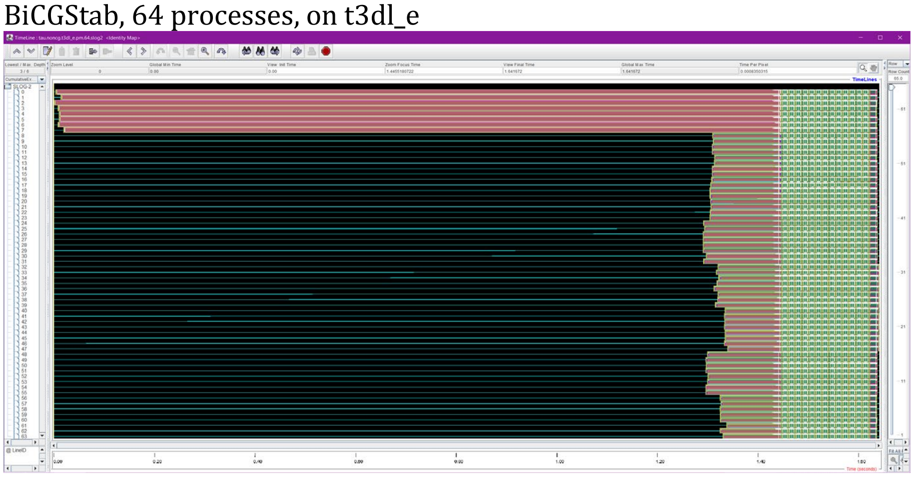
For the CG runs, there is a burst of communication between processes as the SpMV is calculated, followed by two all-reduces as part of the inner product calculations. The delay in completion of the first inner product is caused by the delay in completion of the SpMV due to load imbalances. The second inner product runs much better.

I have shown the BiCGStab split on what seems most likely to be a single iteration to me, based on the algorithm provided above: in line 4, we calculate the inner product of the residual and the shadow residual, then the SpMV of A and p_j. We then find the inner product of this and the shadow residual. On line six, we SpMV to obtain As_j, which we use in an inner product calculation with s_j and then with itself. Finally, on line 12, we calculate the inner product of the next iteration's residual and the current iteration's shadow residual. All other dot and inner products are duplicates of calculations already made during the iteration.

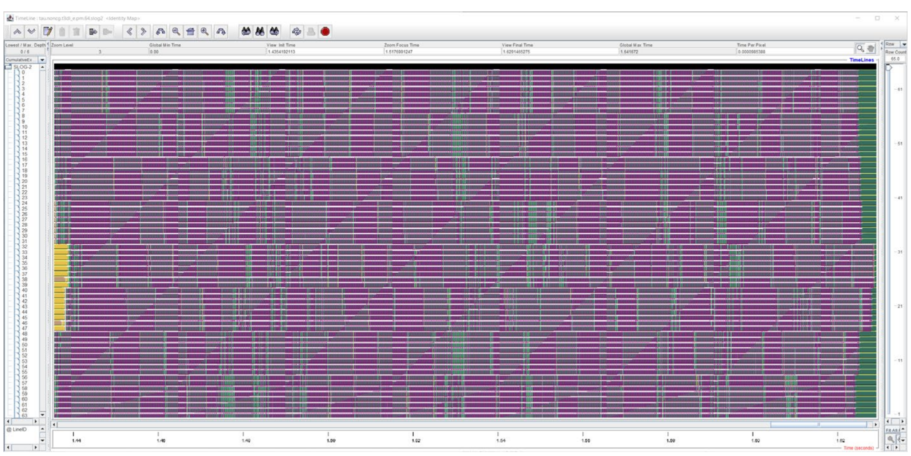
Unfortunately, when I was speaking with the professor, I was told that BiCGStab should have 4 inner products and 2 sparse matrix-vector calculations. I can not see a way to resolve that here, where I consistently seem to see 5 inner products. It is possible that there is another form of the BiCGStab which eliminates one of these calculations.

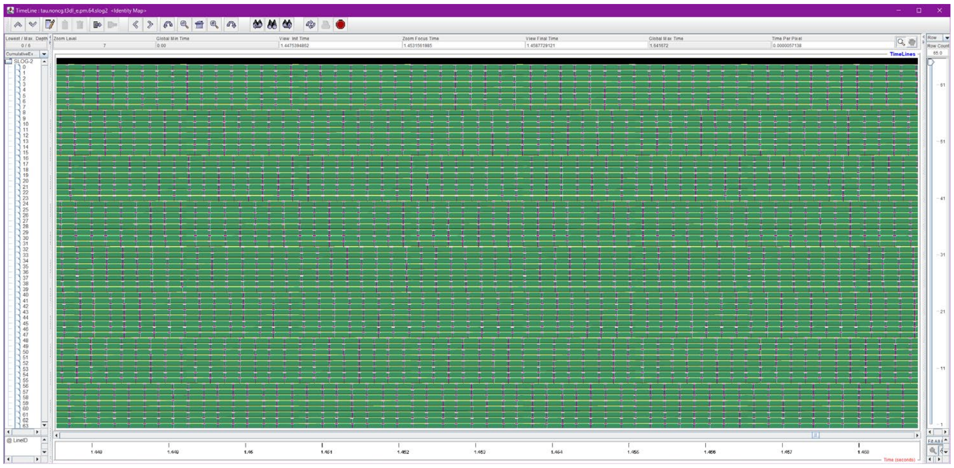
I chose the illustrative Jumpshots with 8 processes because Wheeler features 8 processes per node, and when more than 1 node is used, communication can get very interesting, and Tau seems to start making errors in timekeeping. Most Tau traces fail on runs with 64 processors due to memory overflow, but a couple did run:

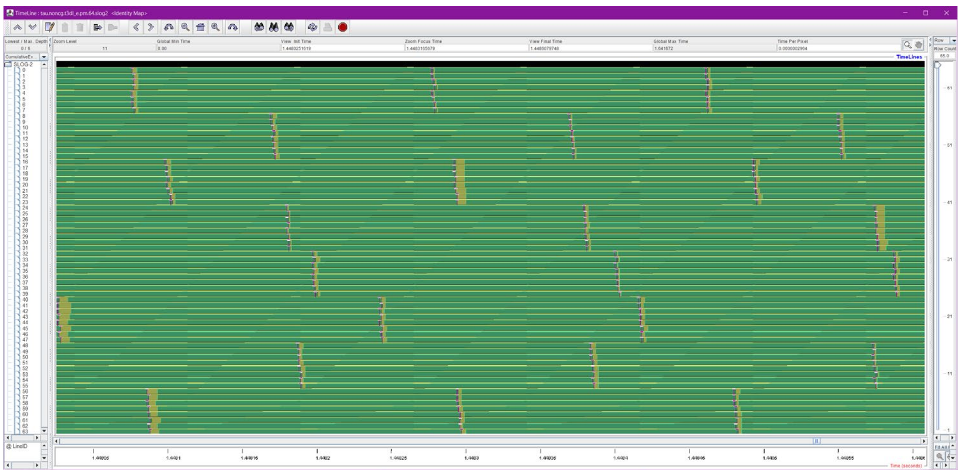
BiCGStab, 64 processes, on t3dl_e



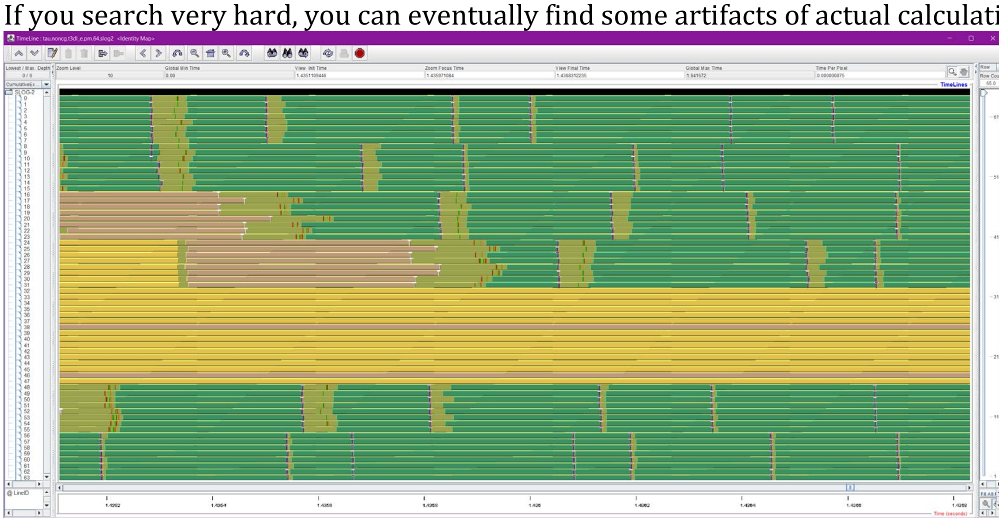
The topmost node is ready almost 1.3 seconds before any other node, a veritable eternity in these situations. The remaining .3 seconds or so of the run are basically entirely all-reduces (the following screen shots are successive zoom-ins):





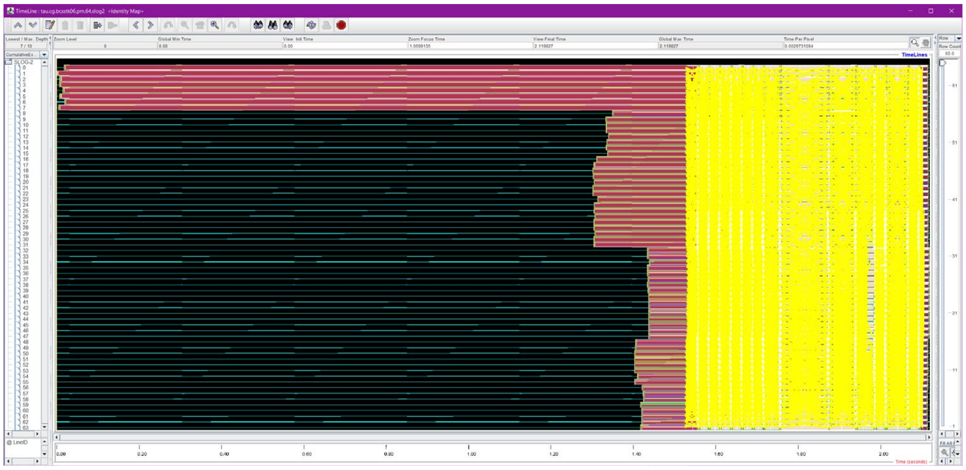


If you search very hard, you can eventually find some artifacts of actual calculation:

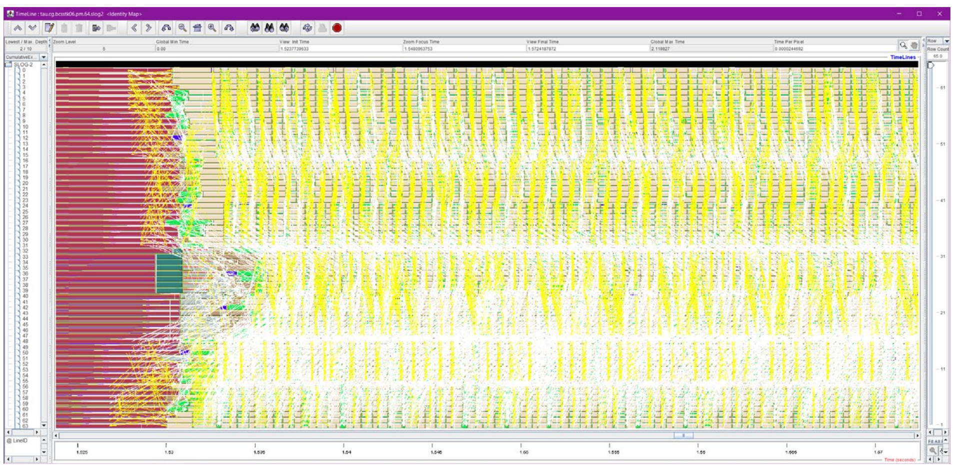


Overall, I suspect that the communication requirements destroyed any hope of this algorithm performing reasonable.

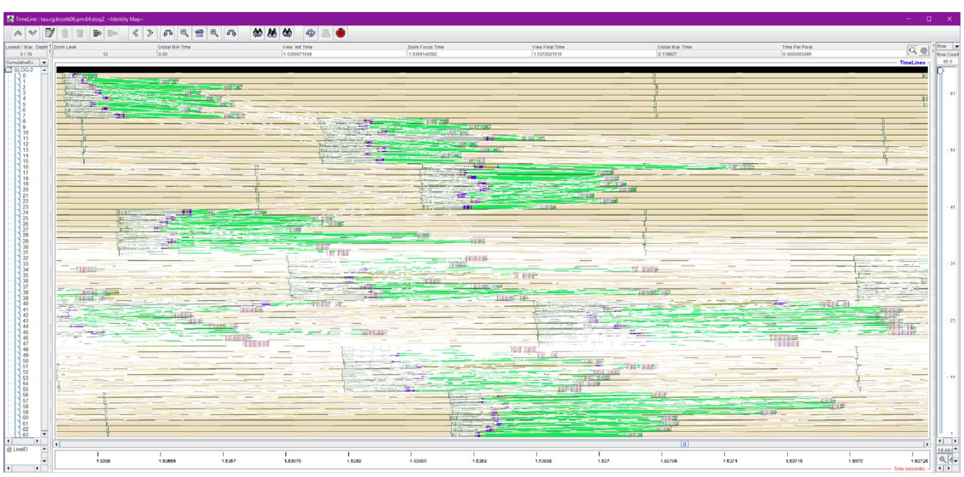
CG features similarly massive delays:



But it has recognizable calculation patterns:



But a single iteration does not yield entirely sensible results:



Costs of Algorithms on Various Matrices Using 8 Processes					
	Iterations	**Total Time (seconds)**	**Time/Iter (microseconds)**	**Total Cost Ratio**	**Per Iter Cost Ratio**
nos5, CG	372	0.081327100	218.62		
nos5, BiCGStab	276	0.059987387	217.35	0.74	0.99
bcsstm12, CG	1728	0.098899249	57.23		
bcsstm12, BiCGStab	1864	0.127937312	68.64	1.29	1.20
mhd3200b, CG	4162	0.137513475	33.04		
mhd3200b, BiCGStab	3203	0.195325097	60.98	1.42	1.85
t3dl_e, CG	198	0.052752405	266.43		
t3dl_e, BiCGStab	171	0.067823326	396.63	1.29	1.49
ex33, CG	303	0.046970703	155.02		
ex33, BiCGStab	296	0.063928360	215.97	1.36	1.39
bcsstk11, CG	1916	0.079138220	41.30		
bcsstk11, BiCGStab	1916	0.133310455	69.58	1.68	1.68
plat1919, CG	2496	0.116543394	46.69		
plat1919, BiCGStab	2496	0.179752219	72.02	1.54	1.54
bcsstk06, CG	548	0.051396295	93.79		
bcsstk06, BiCGStab	548	0.056614649	103.31	1.10	1.10
			Average Total Cost Ratio:		**1.30**
			Average Per Iter Cost Ratio:		**1.41**

Table 1

I did not have time to tabularize all results, but I did have time to compare number of iterations and total time for both algorithms on all matrices using 8 processes.

I find these results slightly surprising. I expected the overall difference in cost to be much greater for large matrices. The matrices with the largest number of nonzeroes, bcsstk11 and plat1919 did show a markedly greater difference in time, but they both required the same number of iterations. I am particularly surprised by the cost ratio with nos5. I can't think of why BiCGStab would be faster than CG in this case, especially not by such a wide margin.