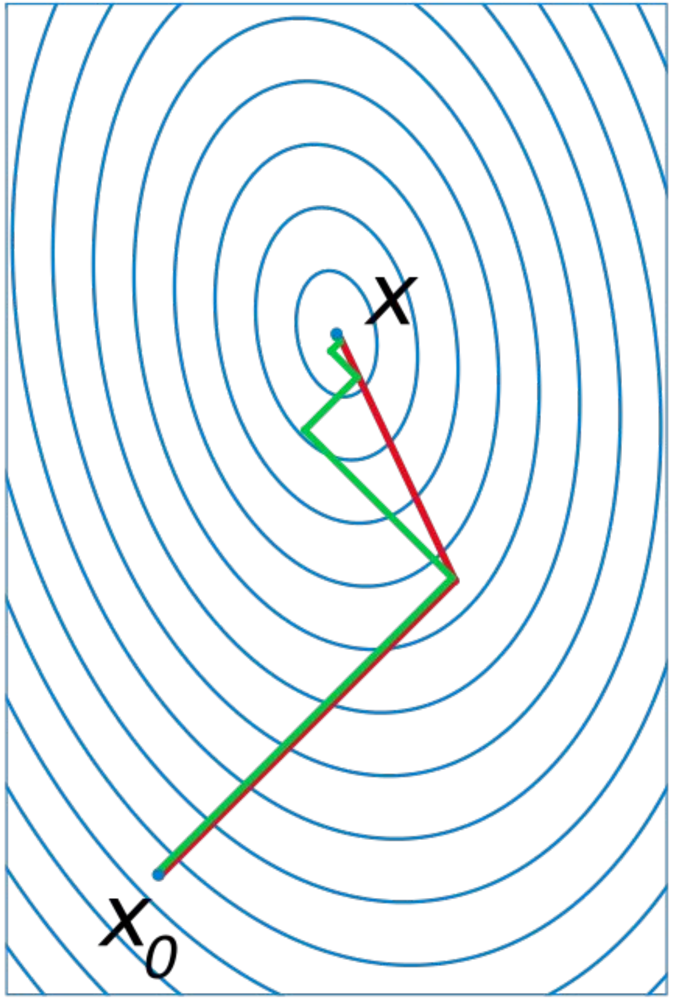
A Parallel Conjugate Gradient Routine

Noah Trupin and Xiaotao Yang

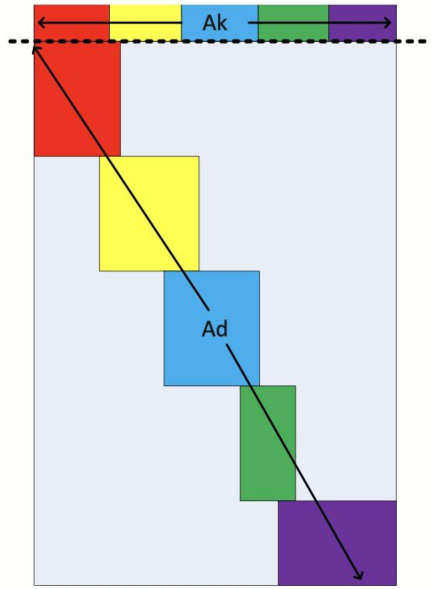
Contents



conjugate vector (red) vs. gradient descent (green) for n = 2.
from Wikipedia

Project Summary

- Implement a **memory-efficient**, **parallel routine** for solving large sparse systems in the form $Ax = b$ (or $Gm = d$).
- **Project Goals**:
  - Reduce memory usage from original LSQR routine.
  - Increase convergence speed.
  - Parallelize components. We have cores, we should use them!
- **Personal Goals**:
  - Learn more about research! (started project as a first-semester freshman)
  - Complete the project goals.



Matrix A

*image from Huang et al. Original partitioning algorithm.

What is the Conjugate Gradient Routine?

- An **iterative Krylov subspace** method for solving sparse systems of linear equations of the form $Ax = b$ (or $Gm = d$).
- Works on square systems–we can fix that using a biconjugate adaptation (dealing with transpose matrices).
- Applicable to matrices too large for solving with direct methods.
- Finding a convergent x with respect to matrix A and vector b behaves similarly to gradient descent; we can use this to backtrack and treat rounding errors.

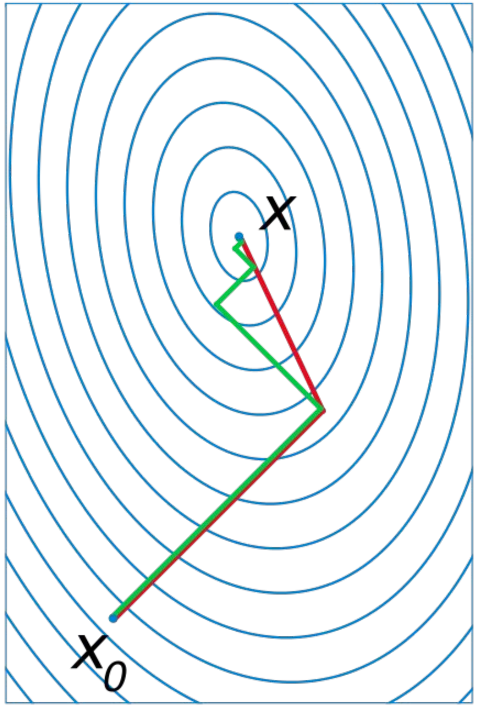


*image from Wikipedia

Krylov Vectors for Non-Square Matrices

- **Column space** of *b* under the first **r powers** of *A*.
- Useful for approximating high-dimensional linear algebra problems.
- Composable on square matrices–ours is rectangular!
- We combine our matrix and its transpose to simulate a square matrix to generate a Krylov subspace for performing the conjugate gradient method.

$$\mathcal{K}_r(A, b) = \operatorname{span}\{b, Ab, A^2 b, \ldots, A^{r-1} b\}.$$

Comparison to SPLSQR

SPLSQR

- Focused on **minimizing computational cost** via **aggressive parallelization**.
- Designed for use on non-square matrices.
- Involves **factoring** the matrix into **orthogonal** and **upper triangular** matrices.
- Not numerically stable for ill-conditioned systems. Ill-formed or unexpected inputs often cause segfaults.

Conjugate Gradient Routine

- Focused on **minimizing communication overhead** via small batch communication and a single-threaded iterative method with **nested parallel components**.
- Generalizable to non-square matrices using biconjugate methods / multiplication with transpose.
- More numerically stable through BiCGSTAB technique. Ill-formed or unexpected inputs will fail to converge without error (itcount = 30,000*)

*taken from Shen and Gao.

Naive Parallelization

- **naive parallelization**: parallelizing every loop construct without considering operation.
- loops with **order-dependent** operations **should not** be parallelized*.
- loops with **order-independent** operations **should** be parallelized.

```fortran
! apply damping
do i = 1, max_x
    coef_dp(1) = damping
    idx(1) = i
    call csm_insert_row(1, idx, coef_dp, 0.0_DP)
end do
```

a non-parallelizable loop

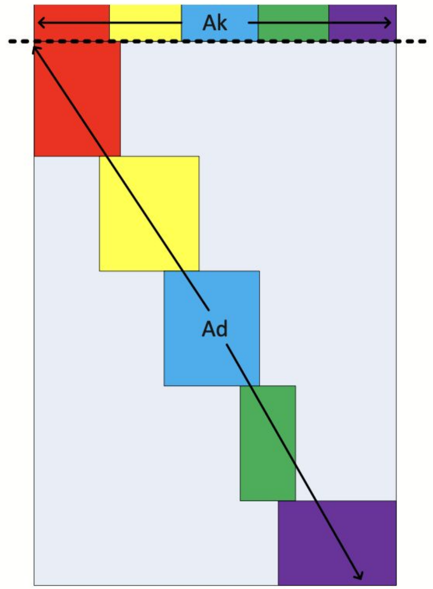
```fortran
!$omp parallel do private(tmp) shared(w) reduction(+:x,sig)
do i = 1, ncol
    tmp = w(i) / rho
    w(i) = v(i) - theta * tmp
    x(i) = x(i) + phi * tmp
    sig(i) = sig(i) + tmp ** 2
end do
!$omp end parallel do
```

a parallelizable loop

* achievable with blocking. As always, context determines application. We did not parallelize order-dependent loops in this project.

Partitioning for Multiple Tasks

- **Goal**: Find balance between communication overhead and computation time.
- **Observations**: in practice, the balance occurs when n = 1 (we do not partition the matrix) and instead perform batch multiplication on subsections of the matrix. When doing so we follow a similar structure along the band. More on this later in presentation.
- Also, partitions often appear ill-conditioned and CGR does not converge in reasonable time.



Matrix A

*image from Huang et al. Original partitioning algorithm.

Partitioning for Multiple Tasks

*From paper**:

(05) Partition matrices: *Ak*, *Ad*, and *Adt*

(06) *Aki* ← *Ak* (partitioning across columns)

(07) *Adi* ← *Ad* (partitioning across rows)

(08) *Adti* ← *Adt* (partitioning across columns)

where:

- *nk*: rows in kernel submatrix.
- *nd*: rows in damping submatrix.
- *A*: our matrix (also referred to as *G*).
- *Ak*: kernel submatrix. $Ak \in \mathbf{R}^{nk \times m}$
- *Ad*: damping submatrix. $Ad \in \mathbf{R}^{nd \times m}$
- *Adt*: transpose of *Ad*.
- *Aki*: piece of kernel submatrix on task *i*.
- *Adi*: piece of damping submatrix on task *i*.
- *Adt*: piece of transpose matrix on task *i*.

*He Huang et al.

Partitioning for Multiple Tasks

*Our implementation**:

```
(05) Partition matrices: $Ak$, $Ad$, and $Adt$
(06)     $Aki \leftarrow Ak$ (partitioning across columns)
(07)     $Adi \leftarrow Ad$ (partitioning across rows)
~~(08)     $Adti \leftarrow Adt$ (partitioning across columns)~~
```

Instead: calculate partition transpose on-the-fly.

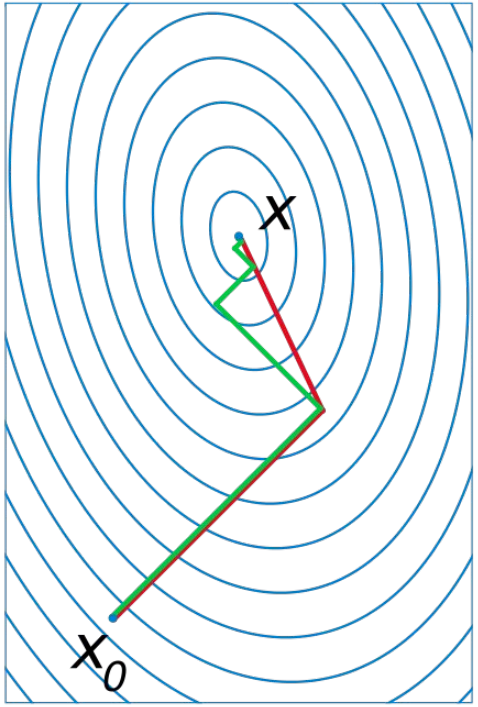
where:

- *nk*: rows in kernel submatrix.
- *nd*: rows in damping submatrix.
- *A*: our matrix (also referred to as *G*).
- *Ak*: kernel submatrix. $Ak \in \mathbf{R}^{nk \times m}$
- *Ad*: damping submatrix. $Ad \in \mathbf{R}^{nd \times m}$
- ~~*Adt*: transpose of *Ad*.~~
- *Aki*: piece of kernel submatrix on task *i*.
- *Adi*: piece of damping submatrix on task *i*.
- ~~*Adt*: piece of transpose matrix on task *i*.~~

*He Huang et al.

Calculating the Solution Vector

- We declare a variable *t = r / a(xx)* as metric of convergence. When *t* reaches specific **tolerance level*** we determine the CGR has converged.
- Utilize scalar quantities as determinants of proximity to convergence rather than checking vectors. Saves space, compute, and unneeded complexity.



*10^-7, from Paige and Saunders

Calculating the Solution Vector

*From paper**:

```
(01) Iterative until converged
(02)     Calculate: $y \leftarrow A * x + y$
(03)         Kernel component:
(04)             $yki \leftarrow Aki * xi$ (partials)
(05)             $yk \leftarrow$ sum($yki$) (sum partials)
(06)         Damping component:
(07)             Communicate: build $x'i$.
(08)             ydi $\leftarrow Adi * x'i$
(09)     Calculate: $x \leftarrow A^T * y + x$
(10)         Kernel component:
(11)             $xki \leftarrow Ak^Ti *$ yk
(12)         Damping component:
(13)             Communicate: build $yd'i$.
(14)         $xdi \leftarrow Adti * yd'i$
(15) Construct orthogonal transformation
(16) Test convergence
```

*He Huang et al.

Calculating the Solution Vector

*From paper**:

```
(01) Iterative until converged
(02)     Calculate: $y \leftarrow A * x + y$
(03)         Kernel component:
(04)             $yki \leftarrow Aki * xi$ (partials)
(05)             $yk \leftarrow \text{sum}(yki)$ (sum partials)
(06)         Damping component:
(07)             Communicate: build $x'i$.
(08)             ydi $\leftarrow Adi * x'i$
(09)     Calculate: $x \leftarrow A^T * y + x$
(10)         Kernel component:
(11)             $xki \leftarrow Ak^T i * yk$
(12)         Damping component:
(13)             Communicate: build $yd'i$.
(14)         $xdi \leftarrow Adti * yd'i$
(15) Construct orthogonal transformation
(16) Test convergence
```

*He Huang et al.

Calculating the Solution Vector

*From paper**:

(01) Iterative until converged

(02) Calculate: $y \leftarrow A * x + y$

(04) $yki \leftarrow Aki * xi$ (partials)

(05) $yk \leftarrow$ sum(yki) (sum partials)

(07) Communicate: build *x'i.*

(08) ydi $\leftarrow Adi * x'i$

(09) Calculate: $x \leftarrow A^T * y + x$

(11) $xki \leftarrow Ak^Ti$ * yk

(13) Communicate: build *yd'i*.

(14) $xdi \leftarrow Adti * yd'i$

2 rounds of communication and numerous small vecmuls per iteration, more overhead than large vecmul. Can we improve this?

*He Huang et al.

Calculating the Solution Vector

Our implementation:

```
(01) $u \leftarrow b/||b||$
(02) $v \leftarrow A^Tu/||A^Tu||$
(03) $w \leftarrow v$ (copy)
(04) Calculate initial values (α, β, etc)
(05) Iterate until converged
(06)     $u \leftarrow (Av - \alpha u)/||Av - \alpha u||$
(07)     $v \leftarrow (A^Tu - \beta v)/||A^Tu - \beta v||$
(08) Update intermediate values
(09) Update solution vectors
(10) Test for convergence
```

Calculating the Solution Vector

Our implementation:

```
(01) $u \leftarrow b/||b||$
(02) $v \leftarrow A^T u/||A^T u||$
(03) $w \leftarrow v$ (copy)
(04) Calculate initial values (α, β, etc)
(05) Iterate until converged
(06)     $u \leftarrow (Av - \alpha u)/||Av - \alpha u||$
(07)     $v \leftarrow (A^T u - \beta v)/||A^T u - \beta v||$
(08) Update intermediate values
(09) Update solution vectors
(10) Test for convergence
```

Only 2 vecmuls per iteration. No need to store transpose, normalize values to prevent overflow (problem for large systems).

Calculating the Solution Vector

Our implementation:

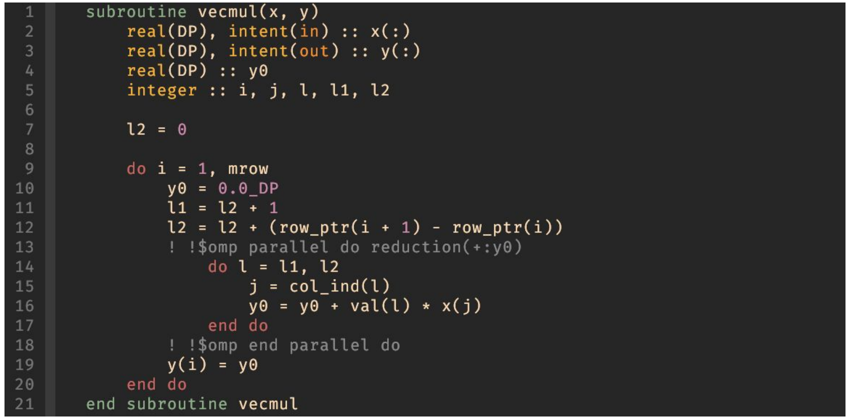
```
(01) $u \leftarrow b/||b||$
(02) $v \leftarrow A^T u/||A^T u||$
(03) $w \leftarrow v$ (copy)
(04) Calculate initial values (α, β, etc)
(05) Iterate until converged
(06)     $u \leftarrow (Av - \alpha u)/||Av - \alpha u||$
(07)     $v \leftarrow (A^T u - \beta v)/||A^T u - \beta v||$
(08) Update intermediate values
(09) Update solution vectors
(10) Test for convergence
```

We love scalar quantities. Why store a vector of multipliers when a single scalar will do?

tvecmul: O(mn) matrix transpose and vector multiplication

Original CSR vecmul routine, for comparison.

1. Loop over row_ptr array
2. Take dot product of row and input vector

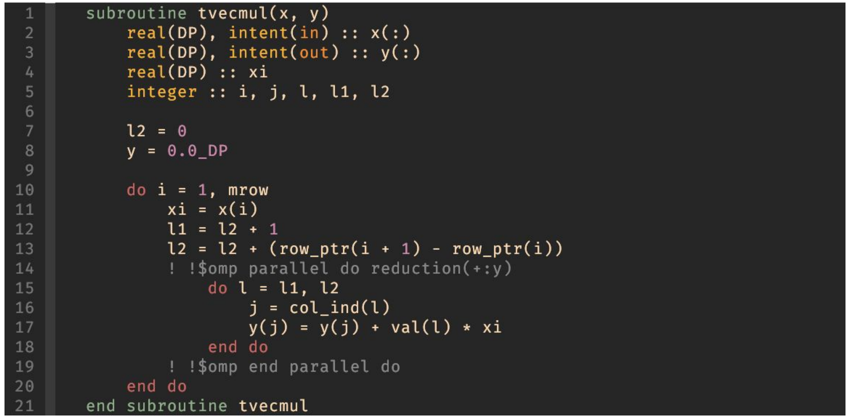


```fortran
subroutine vecmul(x, y)
    real(DP), intent(in) :: x(:)
    real(DP), intent(out) :: y(:)
    real(DP) :: y0
    integer :: i, j, l, l1, l2

    l2 = 0

    do i = 1, mrow
        y0 = 0.0_DP
        l1 = l2 + 1
        l2 = l2 + (row_ptr(i + 1) - row_ptr(i))
        ! !$omp parallel do reduction(+:y0)
            do l = l1, l2
                j = col_ind(l)
                y0 = y0 + val(l) * x(j)
            end do
        ! !$omp end parallel do
        y(i) = y0
    end do
end subroutine vecmul
```

*adapted from Paige and Saunders, 1982.

Transpose → vecmul = $O(m^2n^2)$. Not good!

1. Loop over row_ptr
2. Take columnwise dot product with input vector



```fortran
subroutine tvecmul(x, y)
    real(DP), intent(in) :: x(:)
    real(DP), intent(out) :: y(:)
    real(DP) :: xi
    integer :: i, j, l, l1, l2

    l2 = 0
    y = 0.0_DP

    do i = 1, mrow
        xi = x(i)
        l1 = l2 + 1
        l2 = l2 + (row_ptr(i + 1) - row_ptr(i))
        ! !$omp parallel do reduction(+:y)
            do l = l1, l2
                j = col_ind(l)
                y(j) = y(j) + val(l) * xi
            end do
        ! !$omp end parallel do
    end do
end subroutine tvecmul
```

Testing Process

Parameters / Flags

- Compilation: -03
- MPI Config: slots=25
- Running:
  - OMP_NUM_THREADS = 8
  - nprocs: 25
- Valgrind Tools:
  - Massif
  - Callgrind
  - Default (Valgrind)

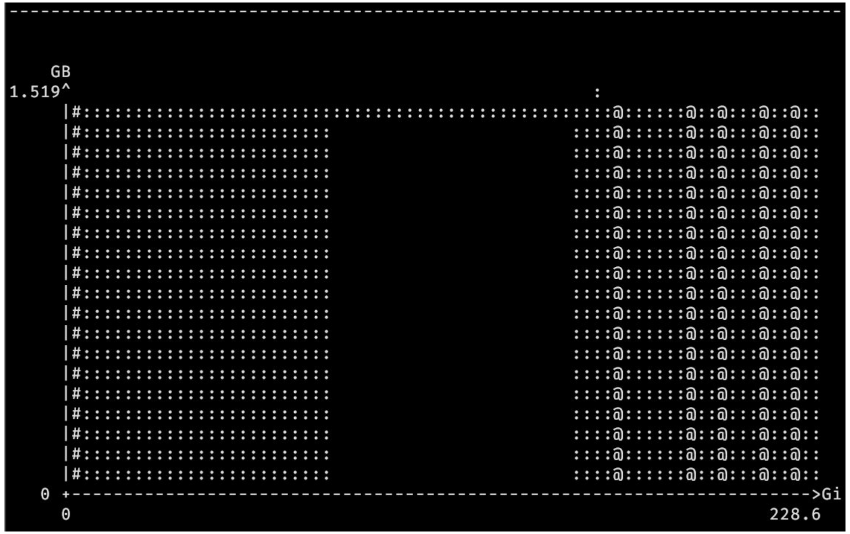
Procedure

1. Run each program 2 times. We want the OS to cache accessed files so we get “hot” access during testing*.
2. Memory Profiling
   a. Compile programs without optimization.
   b. Run with Massif.
   c. Extract graphs and reset environment.
   d. Repeat 3 times.
3. Speed Test
   a. Compile programs with optimization (-Ofast).
   b. Run with Valgrind
   c. Take user time statistics.
   d. Repeat 3 times.
4. For fun, we could test this on both Slurm and user space. For small datasets, we test on user space.

*We can do this in practice by running a low-overhead utility to read the file before execution.

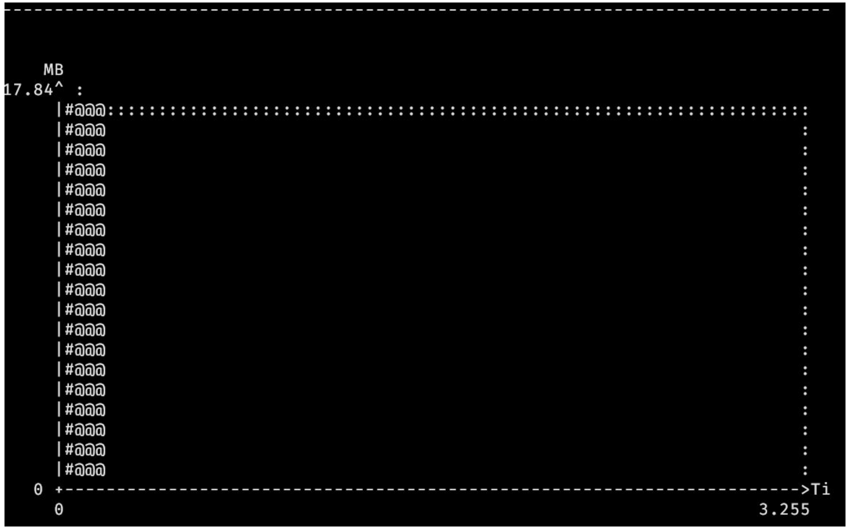
Memory Usage

*From paper**:



```
--------------------------------------------------------------------------------


    GB
1.519^                                                  :
     |#:::::::::::::::::::::::::::::::::::::::::::::::::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@::::::@::@:::@::@::
   0 +----------------------------------------------------------------------->Gi
     0                                                                   228.6
```

*Our implementation**:



```
--------------------------------------------------------------------------------


    MB
17.84^ :
     |#@@@::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
     |#@@@                                                                   :
   0 +----------------------------------------------------------------------->Ti
     0                                                                   3.255
```

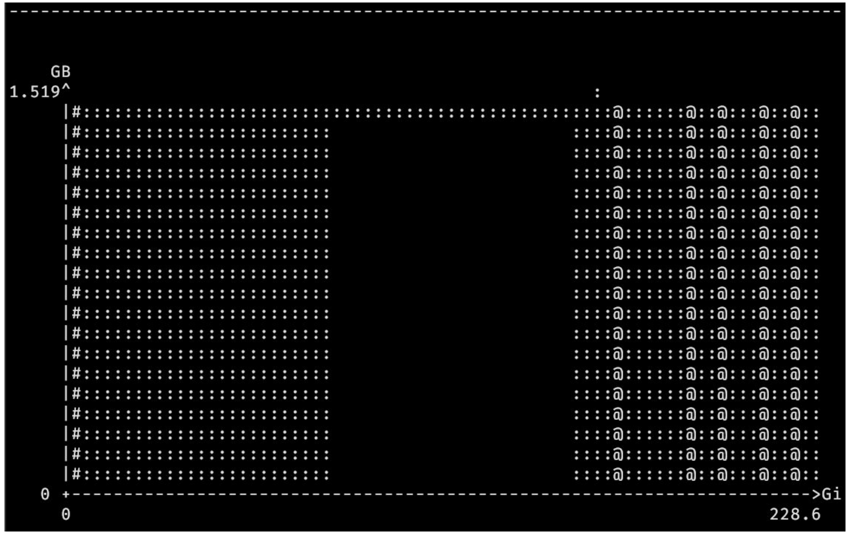
*Massif outputs.

What does this mean?

How to interpret this data: Massif takes snapshots at semi-regular intervals throughout execution. ":" denote regular snapshots, "@" detailed, "#" peak.

- One large allocation at beginning of execution (peak usage).
- Many copies made–we only need one set of data!
- Low on instructions, but high on memory.
- Detailed snapshots mid-solving–lots of data moving.

*From paper**:



```
-------------------------------------------------------------------------------


    GB
1.519^                                                  :
     |#:::::::::::::::::::::::::::::::::::::::::::::::::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
     |#::::::::::::::::::::::::                       ::::@:::::@::@:::@::@::
   0 +---------------------------------------------------------------------->Gi
     0                                                                  228.6
```

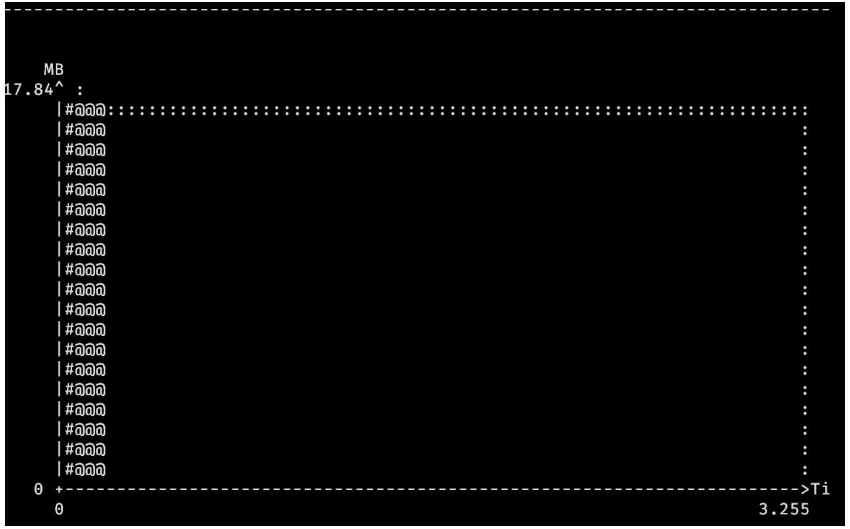
*Massif outputs.

What does this mean?

How to interpret this data: we split operations across four processes while testing. Multiply the memory usage by 25 for more appropriate figures.

- One (smaller!) allocation at the beginning.
- Detailed snapshots occur while loading, stable computation.
- Fewer allocations == less overhead.
- Rely on the stack for smaller values (let Fortran handle dynamically) and repurpose our allocation instead of making new ones.

*Our implementation**:



```
--------------------------------------------------------------------------------


    MB
17.84^ :
     |#@@@::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
     |#@@@                                                                     :
   0 +----------------------------------------------------------------------->Ti
     0                                                                     3.255
```

*Massif outputs.

Some Benchmarks (nel ≈ $1.3*10^8$, n = 10, slots = 25)

LSQR Routine

- **User Time**: 18m 45s
- **Initializing Time**: 0.6s
- **Matrix Loading Time**: ~8m
- **Time to Convergence**: ~9m
- **Peak Memory Usage**: 1.519 GB
- **Successful Runs**: 9/10, 90%

Notes: Segfaulted (somewhat randomly) during development. Numerically unstable method; overflows, division by 0, etc.

Parallel CGR

- **User Time**: 13m 12s
- **Initializing Time**: 0.03s
- **Matrix Loading Time**: ~5m
- **Time to Convergence**: ~7m
- **Peak Memory Usage**: 17.84 * 25 = 446 MB
- **Successful Runs**: 10/10, 100%

Notes: We trade off minimal operations for memory and numerical stability. Remains slightly faster.

Takeaways

On Memory Efficiency

- Fortran utilizes **pass-by-reference**. Let’s take advantage of that!
- We only use our matrices / vectors once per execution. We don’t need to make copies! **Modify in-place**.
- Paige and Saunders uses **single-precision** tolerance, but we stored **double-precision** vectors during computation–unnecessary*!
- We save around **1 single-precision** solution vector's worth of storage per iteration–amounting to ~1 GB (~60%) in test set.

On Parallelization

- If we need to pass data frequently or in large quantities, **parallelization is probably a bad idea**.
- We observed MPI latency at **3.42x** execution speed for large vectors (on the order of 100,000 rows, tested with vecmul and tvecmul).
- In short: large single-threaded vecmul with execution time of 4.52 seconds took ~15.45 second in parallel.

*Double-precision required while loading. We can shorten the variables during computation.

Takeaways

What We Achieved

- Re-implemented Paige and Saunders' **Conjugate Gradient Routine**.
- Added **parallel segments and partitioning** to code–we don't run by default on Bell, but the option exists.
- Reduced **time complexity** and **memory footprint** of matrix loading.
- Cleaned codebase, reduced redundancies, and greatly reduced overall memory usage.

What We Didn't Achieve

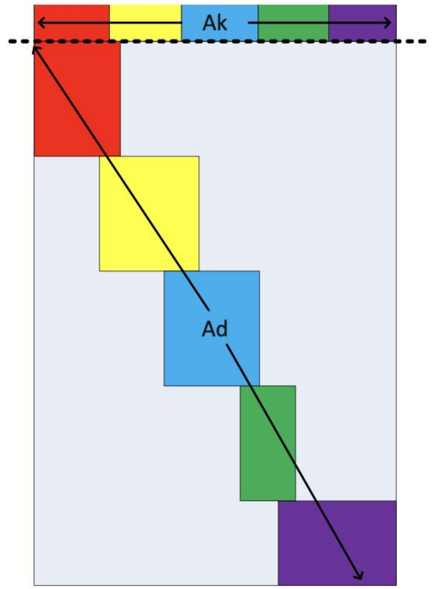
- Find a partitioning algorithm that **balances computation time and communication overhead** (though this may exist!)
- Implement an **external preconditioner** (outside scope of project, great idea for future).
- Reduce matrix loading to sub-O(n^2)–improved from O(n^3) original but still sub-optimal.

*Double-precision required while loading. We can shorten the variables during computation.

Project Summary, Revisited

- **Project Goals**:
  - ✓ Reduce memory usage from original LSQR routine.
  - ✓ Increase convergence speed.
  - ✓ Parallelize components. We have cores, we should use them!
- **Personal Goals**:
  - ✓ Learn more about research! (started project as a first-semester freshman)
  - ✓ Complete the project goals.

We received a fair amount of negative results (forced to change algorithms, block operations instead of full partitions, etc), but we did achieve each goal.



Matrix A

*image from Huang et al. Original partitioning algorithm.

Ideas for Future Improvement

On Memory Efficiency

- Other than developing a fully novel method, I do not see obvious ways of increasing memory efficiency.
- Performing further **elimination / bandwidth reduction** on the matrix leads to a lower memory footprint during computation but greater overall, may be computationally expensive.
- Running an **external preconditioner** over the data will lower net usage but not peak.

On Parallelization / Efficiency

- **GPUs** specialize in fast matrix and vector multiplication and parallel computing. Future implementations could rely on **CUDA** or **OpenGL** rather than **OpenMPI** for reduced latency and increased computation speed.
- We could **precondition** our routine to speed up convergence as seen in a **biconjugate gradient method***. Without prior knowledge of data this may be **computationally expensive**.

*this method is numerically unstable. not good!

References

A scalable parallel LSQR algorithm for solving large-scale linear system for tomographic problems: a case study in seismic tomography. Huang et al. 2013.

LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares. Paige and Saunders. 1982.

An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Shewchuk. 1994.

Full-Wave Ambient Noise Tomography, modified from Shen and Gao. 2018.