A Chapel Parallelisation of the Singular Value Decomposition

Damian G McGuckin, Peter L Harding, Donald E K Carpenter

(Corresponding Author's Email: damianm@esi.com.au)

Pacific ESI

1. What are we trying to do (with Chapel)

We have two software refactoring projects, these being

a) Finite Element software tool for Structural Mechanics & Dynamics
    a) Including fluid-structure interaction
b) Data Filtering software tool for Airborne Survey Analysis
    a) The authors are not the experts in this science

The former is an exercise in the Solution of

a) Partial/Ordinary Differential Equations in Space/Time
b) Large Scale implicit/explicit Non-linear equations

The latter is an exercise in Space/Time filtering of data.

Both are dominated by Linear Algebra operations in real (number) space.

2. How do can we best do it (with Chapel)

a) We wanted guidelines on how best to use Chapel for the tasks mentioned
b) We considered various learning experiences from which to create (a)
c) We ignored Dense Linear Equation Solution as too trivial
d) We ignored Dense Eigen Analysis as too complex
e) We also avoided Sparse Linear Algebra due to its complexity
f) We chose Dense Singular Value Decomposition (SVD) that we already used in
    - formulating individual finite element matrix factors in Fortran
    - airborne data sampling, data filtering and reduction in C++

We up-scaled the size of the problem in (f) deliberately for performance testing.

3. Classical SVD of Golub & Reinsch 1972 (Algol60)

a) Decomposes an *m x n* matrix into a triple matrix product

b) Is the basis of many clones including implementations in
   a) LINPACK software toolkit (1976) in Fortran IV
   b) Numerical Recipes (in C), and others including Python

c) Is both computationally intensive and mathematically complex

d) Was deemed the best candidate for this exercise

- We want to learn how best to use Chapel, **not** how to write a better SVD.
- This is an exercise in creating parallelisation and programming guidelines
- Newer “better” algorithms were ignored as they brought nothing extra

4. Objectives, Constraints and Assumptions

Guidelines for exploiting Chapel's parallel and generic programming features are wanted!

Accuracy and readability will not be sacrificed for performance, but

Performance should be competitive with Fortran XY or C/C++ whatever.

Mathematics, not programming, would dominate the parallelisation and implementation

5. What Chapel Features Are Likely to be Useful

a) Data Parallelism – really only the **forall**
b) Task Parallelism – **NONE**
c) Generic programming to support **real**(32) and **real**(64), or **real**(w)
d) Array operations including data handling (slicing) and mathematics
e) Row-Major storage (Chapel's default **array** storage scheme)

The last of these has major implications in terms of needing to program the mathematics to avoid cache misses, or conversely, to ensure cache hits.

Because Fortran uses column-major storage, the re-factorization of old Fortran code is not just a transliteration of those programs

6. SVD Overview

Given an *m x n* **real***(w)* matrix A, its SVD is the decomposition such that

$$\mathbf{A = U\ W\ V^T}$$

Where

$$W = \text{diag}(w_1 , w_2 \mathbf{, \ldots . , w_n)}$$

Is a diagonal matrix with the singular values w_i along the diagonal, and both U and V are orthogonal with respect to themselves. Three tasks make up the Golub and Reinsch algorithm:

a) A bi-diagonalization of A using Householder reflections to form W';

b) An accumulation of those reflections to form an estimate of U and V; &

c) An iterative diagonalization of W' using a shifted QR strategy to form W.

Operations in the QR strategy are also applied to the estimates of U and V.

7. Dominant Computations within the SVD

The bi-diagonalisation and accumulation comprise the following algebra:

$$Z \mathrel{+}= x * y^T$$

$$y = Z * x$$

$$y = Z^T * x$$

while the final diagonalization is multiple pairs of sets of Givens' rotations. The first set is through angles θ_i and each applied to a pair of consecutive columns of the estimates of U and can be written as

$$U_{1..m,i..i+1} \mathrel{*}= G(\theta_i)$$

The second set involves different rotation angles which are applied to the estimates of V. The RHS of that last equation is a 2x2 Givens' rotation matrix.

That two column matrix on the LHS is problematic and causes cache misses with Chapel's row-major ordering (but not Fortran's column-major ordering).

8. Faster Chapel-Centric QR Transformations

The Givens' rotation operation is cache hostile and involves small matrices.

The algorithm needs to be cache friendly and involve a single matrix preferably.

The mathematics of the Givens' rotation done on columns of the estimate of U can be rewritten in complex space by constructing a complex vector from the 2 columns:

$$U_{1..m,i} + \mathbf{i}\ U_{1..m,i+1} \ *= \ G^c(\theta_i)$$

where

$$G^c(\theta_i) = \cos(\theta_i) + \mathbf{i}\ \sin(\theta_i)$$

A similar formula exists for rotations done on the columns of the estimate of V.

8+. Faster Chapel-Centric QR Transformations

A QR strategy is needed for a each singular value.

Each singular value needs some unknown number of QR iterations.

At each QR iteration within the Chapel implementation, the

a) Givens rotations are applied (unfortunately) recursively to pairs of columns across a *j..k* slice of the estimate U, a bigger matrix than just one of 2 columns

b) Each G^c (θ_i) associated with an iteration is captured in *gt*[*j*. . *k*] whose base type will be **complex***(2w) re*membering that A has a base type of **real** *(w)*.

And ditto for the Given rotations applied to the *j..k* slice of the estimate V.

For a quick explanation of how that *j..k* slice is deduced, see the paper.

For a detailed explanation, carefully read the original 1972 reference.

9. Implementation Details

a) All operations were mandated to access array data across the row
   - To best achieve **data locality** in a row-major storage scheme

b) Chapel code snippets will show what was used in the basic operations
   - They assume that *Z*[?*zD*] is **real***(w)*, and *x*[?*xD*] and *y*[?*yD*] are also **real***(w)*

11. Vector Outer Product

It, i.e.

$$Z \mathrel{+}= x * y^T$$

can be parallelised as a single line of Chapel:

```
[ (r, c) in zD ] Z[r, c] += x[c] * y[r];
```

No loop unrolling was attempted.

Chapel does an phenomenal job at optimising this.

12. Matrix Vector Product I

It, i.e.

$$y = Z * x$$

can be parallelised with a single line of executable Chapel:

```
const (rows, columns) = zD.dims();
const y = [ r in rows ] inner(columns, Z[r, ..], x);
```

The routine **inner** is a naive inner product with a 4-way unrolled internal loop and its second two parameters are mandated to be 1D arrays with unit-stride.

13. Matrix Vector Product II

A variation on a Matrix Vector product

$$y = Z^T * x$$

cannot be parallelised like the previous Matrix-Vector product I without serious cache misses (and hence major performance) problems.

It is instead parallelised as the pseudo partial-reduction

```
var y = [j in columns] 0:R;
[ (r, c) in zD with (+ reduce y) ] y[c] += x[r] * Z[r, c];
```

No loop unrolling is done.

Since 1.22.0, the above's performance has been excellent. Attempts to do this or similar operations with earlier versions were nowhere near as successful.

14. Givens Rotation within the QR Strategy

This can be written naively (and as per the original reference code) as

```
for i in j .. k - 1 do
{
        const gti = gt[i];
        for r in rows do
        {
                const ug = cmplx(U[r, i+1], U[r, i]) * gti;
                (U[r, i+1], U[r, i]) = (ug.re, ug.im);
        }
}
```

The above cannot be parallelised as it stands and causes major cache misses.

15. Parallel Givens Rotation within QR Strategy

By inverting the order of the for loops, the preceding can be rewritten as

```
for r in rows do // forall now feasible as each row is independent
{
        for i in j .. k-1 do // recursive across the row – be careful!!
        {
                const ugt = cmplx(U[r, i+1], U[r, i]) * gt[i];
                (U[r, i+1], U[r, i]) = (ugt.re, ugt.im);
        }
}
```

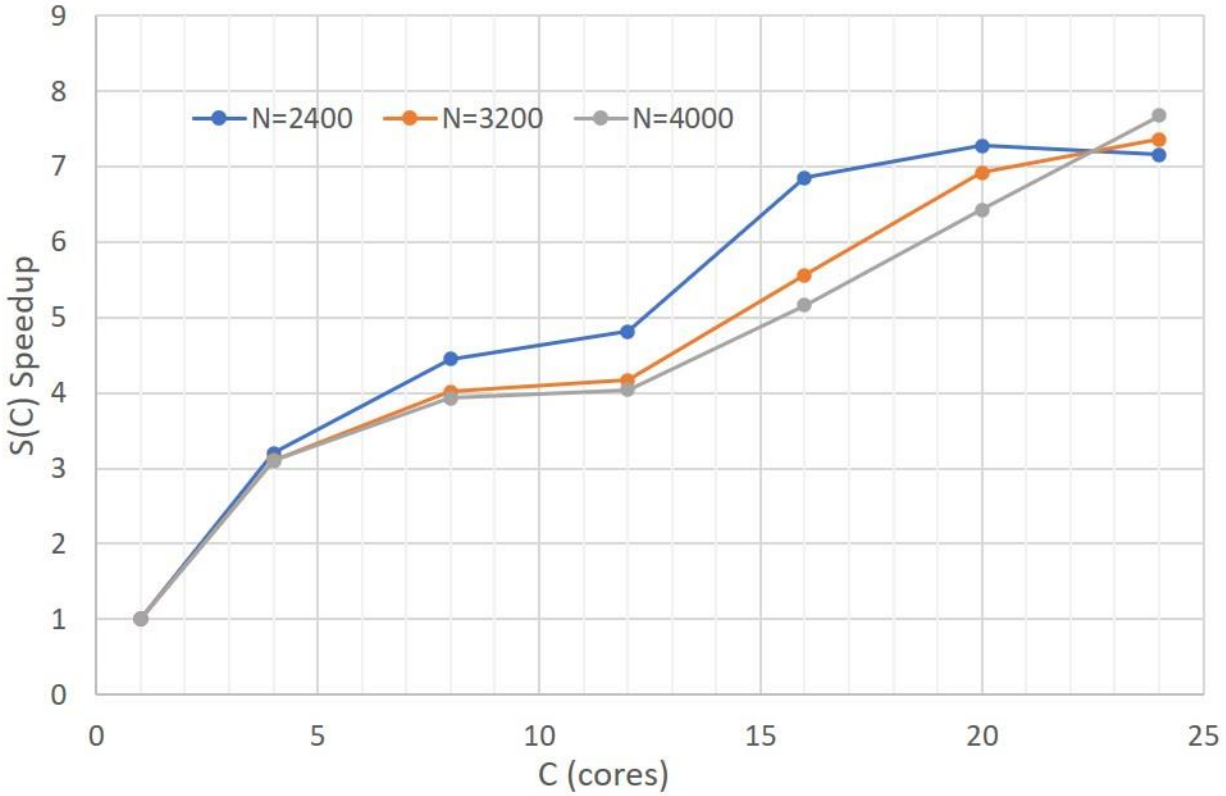
Complex numbers and the associated arithmetic are not used in practice.

The rows are processed in tiles that are unrolled 4 rows deep for performance.

Algorithm also happens to access data across the row and is cache friendly!

16. Performance on Xeon E5-2650v4

Matrices of size 2400*2400, 3200*3200 and 4000*4000 and the structure of the test case 3 matrix from the original reference of Golub and Reinsch were used for performance and accuracy testing.



17. Fortran/Chapel Comparison 1603*1603 test case

Comparing the serial performance of the Chapel with a LINPACK version of the original algorithm yielded:

14.03 seconds : **Fortran** (Linpack Code) slightly vectorised

and

13.36 seconds : **Chapel** as per this implementation

Identical versions of the GCC backend were used (4.8.5).

If the matrix dimensions were a multiple of 2, Fortran was slightly faster, a statistic assumed (but not verified) to be the speed gain from vectorisation.

18. Conclusions

a) We critically examined what we did to create clear and simple guidelines.
b) We now have parallelised SVD implementation in Chapel that works well.
c) It is performant (and comparable to Fortran)
d) The Chapel code is quite readable
e) It is as accurate as the original
f) Chapel's ease of expression made implementation straightforward
g) Improved numerical techniques within the algorithm will yield a better SVD
h) Improved algorithms should yield an even better and faster SVD
i) Vectorisation should improve performance (when it arrives in Chapel)
j) Read the paper for more details
k) Many thanks to those who assisted with this work