A Investigation of the Chapel Parallelisation of the Singular Value Decomposition

Damian G McGuckin
Pacific ESI
Glebe NSW, Australia
damianm@esi.com.au

Peter L Harding
PerformIQ
Caulfield South VIC, Australia
plh@performiq.com

Donald E K Carpenter
Pacific ESI
Tallai QLD, Australia
donc@esi.com.au

Keywords: Parallel Programming, Chapel Language, Numerical Linear Algebra, Singular Value Decomposition

ABSTRACT

This paper discusses the parallelisation (using the programming language Chapel) of one of the most often used algorithms for the Singular Value Decomposition of a matrix. The mathematical alterations and programming constructs used to achieve that parallelisation are examined at length. Performance and validation tests showing the increased parallel performance that occurs when run on a multi-core computer architecture are documented. Chapel's serial performance is compared against that of Fortran that implements the original algorithm. Only parallelisation in a Symmetric Multi-Processor environment is explored.

1. Introduction

During research into the mathematics behind advanced engineering software tools, several linear algebra algorithms were implemented in Chapel, a programming language designed for productive parallel computing at scale [1]. The parallelisation and implementation of one of those algorithms, which together are called **the exercise**, is discussed herein.

Five decades ago, Golub and and Reinsch published their algorithm for the Singular Value Decomposition (SVD) of a rectangular $m \times n$ matrix written using the programming language Algol60 [2]. That algorithm, called **the classical SVD** herein, is a clear and concise expression of the underlying mathematics. It has been the basis of other near clones or slight variants, the most significant of which was the implementation written in Fortran IV later that same decade as a component of the LINPACK [3] toolkit.

As an algorithm that was both compute-intensive and mathematically complex, the classical SVD was deemed to be an excellent candidate from which to learn how to parallelise other linear algebra algorithms and program them in Chapel. Alternative SVD algorithms with improved performance or accuracy (or both) that have appeared since the late 1980s were rejected as candidates because they would have added nothing to the overall learning experience.

The exercise is not really about computing the SVD because Chapel can do this today with its interface to the Fortran LAPACK [4] toolkit. The SVD is simply the example with which to illustrate how Chapel can help facilitate the parallelisation of a linear algebra algorithm.

What follows assumes that the reader has some familiarity with the Chapel language, the programming of numerical linear algebra, and floating point computations.

2. Objectives and Constraints

Rather than its programming, it is the mathematics behind any linear algebra algorithm that dominates its **parallelisation** and implementation. So, knowing how to craft that mathematics will be crucial to achieving a production-grade algorithm. That information, showing the use of Chapel's parallel processing and generic programming features, is the sole objective and output of this exercise.

Various constraints were placed on the exercise, driven by the context in which it was being done:

a. the **accuracy** of the results would not be sacrificed for the algorithm's performance, i.e. its elapsed run-time;

b. the **readability** of the Chapel code would at least match that of the Algol60 used in the original; and

c. the serial performance of the implementation should be competitive with that of either Fortran or C/C++.

Only parallelisation in a Symmetric Multi-Processor (SMP) environment, i.e. on a single computer system or what Chapel calls one **locale**, is explored. Any parallelisation in a Distributed Multi-Processor environment is ignored by design, or more correctly, by project scope (and budget).

3. Chapel Overview

Chapel is a programming language that facilitates the clear expression of algorithms for computational mathematics. It achieves this by providing features that allow a high level of abstraction.

Chapel provides explicit *data-parallel* constructs (the **forall**-expression and the **forall**-statement) and several idioms that support data parallelism implicitly (reductions and scans, whole-array assignment and arithmetic, and function and operator promotion). It also provides explicit *task-parallel* constructs, the **begin**, **cobegin**, and **coforall**. The last two sentences are far too brief to do justice to their content.

The implementation uses many of the data parallel features, including the compact form of the **forall** statement, but none of the task parallel features.

Chapel's underlying support for clear, easy-to-read and powerful *generic programming* and polymorphism was used to ensure that the code works for both **real**(32) and **real**(64) floating point types. As Chapel seeks to support processors with hardware implementations of floating point types such as **real**(16) and **real**(128) by introducing them into the language

in the near future, that same code should work unchanged.

Chapel's ability to express *array operations* concisely is particular interesting in a linear algebra context. One example of this is the concept of a *submatrix*. Consider the declaration of an $m \times n$ two-dimensional (2D) array (or matrix) Z:

var *Z : [1 .. m , 1 .. n]* **real***(64)*;

Chapel can be used to do things like concisely reference respectively a submatrix of Z or make a copy of the entirety of one of its columns using the concept of an array slice as:

```
ref Zremainder = Z [ i .. m , j .. n ]; // no copy involved
const Zcolumnj = Z [ .. , k ]; // make a copy of a column
```

While such slicing has an overhead, its judicious use in the Chapel implementation of the classical SVD algorithm proved that this overhead was minimal and the clarity of expression that it provided was most welcome.

Numerous *basic linear algebra procedures* were written for this exercise to support fundamental one-dimensional (1D) array operations. Their use helped to avoid the slowness of some whole-array arithmetic operations which were less than performant due to Chapel not supporting vectorisation at the time. As an interim measure to enhance their performance, limited manual loop unrolling was done within some of these routines, but only where this proved to be advantageous.

An additional routine written for this exercise was the *overloaded function* **cmplx** which avoided the need to use type casting when constructing a **complex***(2w)* number from arguments where w is the width in bits of the floating point numbers which comprise the real and imaginary components of the return value of that function. This utility function accepts as argument(s), one **real***(w)* number, or two **real***(w)* numbers, or one **imag***(w)* number, or even a tuple of two **real***(w)* numbers. It provides slightly more comprehensive functionality than that of the intrinsic function of the same name that has been available to Fortran users for decades.

4. Array Storage Schemes

Any parallelisation must ensure data locality when handling 2D arrays within a linear algebra algorithm so that it maximises hits (or minimises misses) on a computational processor's cache during memory access [5]. This access and hence locality is in turn intricately linked to the arrangement of those arrays within memory.

Fortran, the language which still casts a giant shadow over mathematical software programming even today, uses a **column-major** array storage scheme. Algorithms that are implemented in Fortran process arrays column-by-column to ensure that data that is next to be processed is close (or even adjacent) to data that has just been processed. This maximises the cache hits. A very long list of linear algebra algorithms have been (and are being) written to access and process data in this way, including the classical SVD.

Chapel on the other hand, uses a **row-major** storage scheme (by default) for its arrays, just like C/C++ and Pascal. Algorithms that are to be implemented in Chapel should process arrays row-by-row to maximise those same cache hits, not the arrangement for algorithms previously written to suit Fortran. This will mean that those algorithms had to be reformulated, a task that proved to be the major challenge of this exercise!

5. Algorithm Overview

The SVD decomposition of a real $m \times n$ rectangular matrix A computes the rectangular, square and diagonal matrices U, V, and Σ such that

$$A \equiv U\ \Sigma\ V^T \tag{1a}$$

where

$$\Sigma \equiv diag(\sigma_1, \sigma_2, \cdots, \sigma_n) \tag{1b}$$

and

$$U^T U \equiv V^T V \equiv V V^T \equiv I \tag{1c}$$

Three floating point intensive code segments make up the original 1970 Algol60 algorithm:

a. A **reduction** (using Householder reflections) of A to its bi-diagonal form Σ' whose respective diagonal and off-diagonal elements are w_k and e_k at column k and where $e_1 \equiv 0$;

b. An **accumulation** of those same reflections to form an estimate of V, and U; and

c. An iterative **diagonalisation** (using an initially *shift*'ed **QR** strategy) of Σ' that isolates the diagonal σ_k by eliminating the off-diagonal e_k by transforming Σ' such that $e_k \equiv 0$ which implies $w_k \equiv \sigma_k$;

after which the σ_k are **sort**ed into descending order and the column ordering of U and V adjusted appropriately to match.

From an abstract perspective, most of the floating point operations seen during the reduction (with reflections) and accumulation (of those reflections) are just the matrix operations:

$$\begin{aligned} Z &+= x \times y^T \\ y &:= Z \times x \\ y &:= Z^T \times x \end{aligned} \tag{2}$$

where x and y are intermediate vectors and Z refers to the any of the matrices A, U or V (or submatrix thereof).

The only dominant floating point operation seen during the iterative diagonalisation is the application of Givens rotation matrices to pairs of columns within U and V. During this diagonalisation, an iterative process is used to isolate (in turn), the k'th singular value, $k \in 1..n$. Each pass of every such iterative process identifies the column j closest to k ($2 \le j < k$) where w_{j-1} is small. This defines the submatrix $\Sigma'[j..k, j..k]$ through which the algorithm will *chase* zeros by applying a mix of multiple Givens rotations, in fact $(k - j)$ of them, to adjacent row and column pairs of Σ'. That Givens rotation mix is then also applied consecutively to two-column submatrices of $U[1..m, j..k]$ and $V[1..n, j..k]$, i.e.

$$\begin{aligned} U_{1..m\ ,\ i..i+1} &\times= \bar{G}(\theta_i) \\ V_{1..n\ ,\ i..i+1} &\times= \bar{G}(\psi_i) \end{aligned} \tag{3}$$

where $\bar{G}(\theta_i)$ and $\bar{G}(\psi_i)$ are compact Givens rotation matrices,

$$\begin{aligned} \bar{G}(\theta_i) &\equiv \begin{bmatrix} +\cos(\theta_i) & -\sin(\theta_i) \\ +\sin(\theta_i) & +\cos(\theta_i) \end{bmatrix} \\ \bar{G}(\psi_i) &\equiv \begin{bmatrix} +\cos(\psi_i) & -\sin(\psi_i) \\ +\sin(\psi_i) & +\cos(\psi_i) \end{bmatrix} \end{aligned} \tag{4}$$

θ_i and ψ_i are the rotation angles applied to columns $j..k$ of respectively U and V, and $i \in [j, k-1]$.

Either two-column matrix that appears on the left hand side of Eq.3 is problematic. Accessing data in that matrix will likely cause serious cache misses because its components come from columns of Chapel 2D arrays, U and V, stored in row-major order. Also, with just two columns in that matrix, it will not be large enough to see much performance gain from parallelisation. Quite clearly, there is no future from the perspective of parallelisation for Eq.3 as it stands and the underlying mathematics must be reformulated. At least one such example of this exists [6], but that approach was discarded because of its complexity.

Of later interest to this discussion is the fact that if each column of those two column submatrices is treated separately, then Eq.3 can be written in complex space as

$$\begin{aligned} U_{1..m\ ,\ i+1} + U_{1..m\ ,\ i}\ \mathbf{i}\ \times &= \bar{G}^c(\theta_i) \\ V_{1..n\ ,\ i+1} + V_{1..n\ ,\ i}\ \mathbf{i}\ \times &= \bar{G}^c(\psi_i) \end{aligned} \tag{5}$$

where $\bar{G}^c(\theta_i)$ and $\bar{G}_c(\psi_i)$ are complex compact forms of the Givens rotations of Eq.4 defined in complex space as

$$\begin{aligned} \bar{G}^c(\theta_i) &= \cos(\theta_i) + \sin(\theta_i)\ \mathbf{i} = \mathbf{cmplx}(\cos(\theta_i), \sin(\theta_i)) \\ \bar{G}^c(\psi_i) &= \cos(\psi_i) + \sin(\psi_i)\ \mathbf{i} = \mathbf{cmplx}(\cos(\psi_i), \sin(\psi_i)) \end{aligned} \tag{6}$$

with **cmplx** being the Chapel function mentioned in §3.

6. Implementation Considerations

Refinements and enhancements must be applied to the algorithm of the classical SVD to yield a parallel Chapel implementation. These mainly relate to data locality issues, accuracy, and importantly, how Givens rotations are applied.

6.1 Achieving Data Locality Implicitly

The first technique that helps achieve data locality is optimal memory access when reducing A to its bi-diagonal form (with Householder reflections) and accumulating those reflections on U and V. The intensive computations identified in Eq.3 are an outer product and two matrix-vector product variants. The library routine **dot** was rejected for these tasks because of observed poor performance. With the project's scope not extending to investigating this problem (and rectifying it), the code snippets shown below were used, explained assuming a matrix Z[?zD] of some type R, and two vectors, x[?xD] and y[?yD] of the same type. Unless explicitly noted in these snippets, the quest for performance did not exploit loop unrolling, this being relegated to a future improvement.

6.1.1 Vector Outer Product

This operation is

$$Z\ +=\ x \times y^T$$

It can be parallelised in an cache-friendly fashion as:

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

No loop unrolling has been attempted.

6.1.2 Matrix-Vector Product I

This operation is

$$y = Z \times x$$

It can be parallelised in a row-major fashion as:

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

where the *inner* routine is currently a 4-way unrolled (serial) inner product **inline proc**edure which is not vectorised. An unroll beyond a 4-way was found to be counterproductive.

6.1.3 Matrix-Vector Product II

This operation is

$$x = Z^T \times y$$

There are serious cache misses in a parallelisation like:

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

Rewriting this to access memory in a row-major fashion demands the use of a pseudo partial-reduction, an alternative which is fractionally less (but still sufficiently) accurate

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

Again, no loop unrolling has been attempted.

6.2 Achieving Data Locality Explicitly

The second technique that helps achieve data locality is the judicious use of a cached copy of data that is referenced more than once within a single pass of a loop. In particular,

a. at the i'th stage of the bi-diagonalisation, an easily cached local copy is made of the i'th diagonal column of A and used during that stage, and

b. at the i'th stage of the left accumulation on U, an easily cached local copy is made of that same column from (a), and used during that stage.

Doing this, as opposed to working with a **ref**-erence to that column, can often dramatically improve the cache hit rate and hence the algorithm's performance.

The above approaches mitigate the cache miss rate problem. The memory penalty during the algorithm totals an additional $m + n$ floating point numbers, a small overhead for better algorithm performance. These unit-stride copies are also a pre-requirement for vectorisation to occur.

6.3 Applying Givens Rotations

A Chapel implementation of the first line of the algebra in Eq.5 for every i rotations within one iteration pass during the iterative diagonalisation stage is shown in Fig. 1. It demands that every $\bar{G}^c(\theta_i)$ associated with that iteration has been captured in a **complex***(2w)* 1D array called (say) $gt[\,j\,..\,k\,]$.

```
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);
  }
}
```

Figure 1. Serial Givens Rotation - Simple Version

With a recursive relationship between successive columns beyond the second in Fig.1, there is no independence between the columns at the outer loop making its parallelisation impossible. The overheads of working with the inner loop in parallel is horrendous so its parallelisation is impractical from any perspective. Deeper inspection shows that the operations within each row are independent of the operations of any other rows. Those rows are now obvious candidates for parallelisation and there are certainly more than enough computations per row to make that parallelisation worthwhile.

Reversing the order of the **for** loops in Fig.1 reveals an algorithm variant with the elements of each inner loop independent of each other which can be parallelised. Each of these can run in parallel by just replacing the outside loop with a **forall**. The (now) parallel Chapel implementation of the algebra of that first line in Eq.5 is shown in Fig.2.

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

Figure 2. Parallel Givens Rotation - Simple Version

Testing showed that the complex number arithmetic ran much slower than a longhand version using real number arithmetic, probably caused by the Chapel compiler being understandably conscious about its accuracy. It would handle complex arithmetic by subroutine calls that do complex number range reduction to avoid overflow and underflow. These two floating point exceptions are most unlikely because the sum of the squares of the elements of $\bar{G}^c(\theta_i)$ will always equate to **1.0**. On that basis, a longhand version of the complex arithmetic of Fig.2 was used in the production algorithm.

Further testing revealed a register stall in the (recursive) inner loop of Fig.2. The loop was unrolled to force more computations into each pass through this loop. This seemed to minimise the impact of this stall on the performance of the code. This meant that rows had to be processed in tiles, suitably handling any remainder when the number of rows was not an exact multiple of the tile size. Tiling two rows at a time reduced the impact of the stall, four rows at a time more so, but tiling of more than four rows at a time resulted in too many cache misses for even medium sized matrices. So, the production version uses four row tiles.

The code in Fig.2 was programmed as a generic Chapel **inline proc** with longhand arithmetic. This was then able to be re-used for the second line in Eq.5 which applies a related (but different) complex compact Givens rotations, namely $\bar{G}^c(\psi)$, to submatrices of V.

6.4 Miscellaneous

In the classical SVD implementation, one loop follows an index from n down to **1** and executes the loop only if the element of an array *e[1..n]* at that index is not negligible when compared to the norm of the bi-diagonal matrix. It relies on a **sentinel value** of *e[1]* always being zero (0.0) so that it never executes (but instead exits) the loop when that index is one (1). The Chapel code does not rely on the sentinel value and explicitly tests the index at each iteration. The impact on the run time was shown to be negligible.

Within the classical SVD, there are two places during the iterative diagonalisation where there is a risk of a **floating point exception** occurring during the computations of the explicitly-calculated initial *shift* or the implicitly-calculated Givens rotation matrices. This is due to some elements of Σ' being almost zero. Some simple scaling techniques that help avoid such problems have been included within this Chapel implementation. It should be noted that the approach used is only an interim measure awaiting a solution using techniques and methods from an updated BLAS approach [7] and the Chapel IEEE754 module [8].

7. Performance Measurement

The ideal parallelisation halves the run-time if the number of central processor unit (CPU) cores C on which to run the algorithm are doubled; and halves that run-time again if the number of CPU cores are doubled for a second time. This direct proportionality *speedup* can be expressed as:

$$\frac{R_1}{R_C} = C \tag{7}$$

where the run-time of the code in parallel mode when C cores are used is R_C, and the run-time of the code in serial mode when only one core is used is R_1.

Any parallelisation is unlikely to be the ideal of Eq.7. The real world relationship is better expressed as a function along the lines of

$$\frac{R_1}{R_C} = S(C) \tag{8}$$

The term $S(C)$ is the *speedup*. When it is a damped variant of a linear function, its most common form, it is a measure of how closely a parallel algorithm delivers on that ideal of direct proportionality.

8. Numerical Experiments

To ensure consistency with the original reference [2], numerical experiments were run on the two most extreme of its three (explicitly chosen difficult) validation cases. Further experiments were run on one of these test cases to provide performance data to measure the effectiveness of the parallelisation. All of the experiments quoted herein were run on a dual Intel Xeon E5-2650v4 system with 8 memory channels and 24 computational cores so that no more than three cores shared one such channel at peak core usage.

The first test case is the specific (small) 8×5 matrix A defined as:

$$A = \begin{bmatrix} +22 & +10 & +2 & +3 & +7 \\ +14 & +7 & +10 & +0 & +8 \\ -1 & +13 & -1 & -11 & +3 \\ -3 & -2 & +13 & -2 & +4 \\ +9 & +8 & +1 & -2 & +4 \\ +9 & +1 & -7 & +5 & -1 \\ +2 & -6 & +6 & +5 & +1 \\ +4 & +5 & +0 & -2 & +2 \end{bmatrix} \tag{9}$$

The second test case is the generic $N \times N$ matrix A whose elements a_{ij} are defined as:

$$a_{ij} = \begin{cases} 0 & \text{if } i > j \\ +1 & \text{if } i \equiv j \\ -1 & \text{if } i < j \end{cases} \tag{10}$$

Performance data from experiments with Eq.10 run for a mix of matrix sizes $N \in [2400, 3200, 4000]$ and active cores C where $C \in [4, 8, \ldots, C_{\max}]$ and $C_{\max}$ is the total number of cores on the system was used to evaluate the speedup. The independent and dependent experimental variables during these performance tests were respectively the number of cores on which each experiment was run and the elapsed time of

each experiment. The value of N was chosen large enough that the run-time start-up overhead was an insignificant part of the elapsed (or run) time of each experiment in the sequence.

To ensure that only C cores would be used for data parallelisation, the **config** constant definition

--dataParTasksPerLocale=C

was given as a command-line option to the Chapel executable to control the number of tasks used for data parallelisation.

The serial performance of the algorithm was needed to provide a baseline elapsed running time for when each test case was not parallelised. Such serialisation was achieved by calling the algorithm from within a **serial** block and using

--dataParTasksPerLocale=1

on the command line for consistency because $C \equiv 1$.

8.1 Software Platform

All numerical experiments were run with version 1.22.0 of the Chapel compiler, **chpl**, with a **gcc** (version 4.8.5) back-end. The Chapel compiler options used were:

chpl --fast --ieee-float ....

The second option wisely demands that the compiler not violate the IEEE 754 Standard [9] in the code it produced.

8.2 Accuracy Verification

The SVD decomposition of the $m \times n$ matrix A of Eq.1a can be verified by comparing that same A against the $m \times n$ matrix product A' of those decomposed parts, i.e.

$$A' = U\ \Sigma\ V^T \tag{11}$$

From Eq.11, the worst relative error in any element of A' is the scaled worst case in the ∞-norm δA of the matrix of relative errors between elements of A' and A, a'_{ij} and a_{ij}, i.e.

$$\delta A = \frac{1}{N} \times \left| 1 - \frac{a'_{ij}}{a_{ij}} \right| \quad \forall i, j \ \in \ 1..N, 1..N \tag{12}$$

Where some a_{ij} was zero, the relative error δA in Eq.12 was replaced by its absolute counterpart. The scaling by the reciprocal of N tries to remove the effect of the error from the computation of Eq.11 itself which has an error of $N \times \varepsilon$ (where ε is the machine precision).

9. Discussion

The real (or elapsed or run) times R_C for the various experimental cases at **real**(64) precision are given in Table 1.

Cores	**N = 2400**	**N = 3200**	**N = 4000**
C	$R_C(secs)$	$R_C(secs)$	$R_C(secs)$
1	50.6480	125.4140	247.4930
4	15.8240	40.3940	79.8760
8	11.3730	31.2230	62.9920
12	10.5180	30.0970	61.2090
16	7.3890	22.5310	47.9370
20	6.9630	18.1110	38.5160
24	7.0750	17.0310	32.2210

TABLE 1. Elapsed Times on a Xeon E5-2650-v4

The speedup *S(C)* seen in those experiments could then be computed from each R_C using Eq.7. The results are plotted in an un-smoothed fashion in Fig.3.

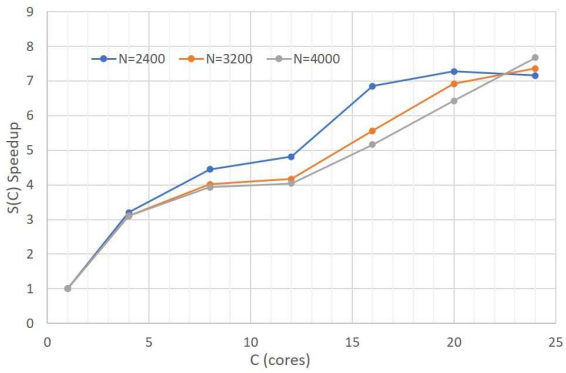

Figure 3. Speedup (S(C)) Against Cores (C) Used

The plot shows that the results parallelised well and that they scaled consistently with the matrix size. The speedup curve is largely a damped linear curve between one and eight cores. Beyond eight cores, the system no longer has one dedicated memory channel for each task (or core) and those tasks have to compete for communication (or memory) bandwidth, the **speedup** curve would be expected to exhibit some change in behaviour at this point. For the Xeon E5-2650v4 on which the experiments have been run, this scenario occurs after eight cores and that explains the kink seen above just after eight cores. Beyond that, the performance goes back to being a damped linear curve again.

A simple and robust verification of the correctness and accuracy of the Chapel implementation looked at the singular values of Eq.9 and Eq.10 from the numerical experiments and compared them against that from the original reference [2]. For Eq.9, these actually have exact values of:

$$\Sigma = [\sqrt{1248}\ ,\ 20\ ,\ \sqrt{384}\ ,\ 0\ ,\ 0]$$

Using absolute errors in the case of the last two values, and relative errors for the rest, the reference case shows calculation errors for these of:

$$\Delta\Sigma = [23\varepsilon\ ,\ 13\varepsilon\ ,13\varepsilon\ ,\ 6\varepsilon\ ,\ 13\varepsilon]$$

For the **real**(32) case in the Chapel implementation:

$$\Delta\Sigma = [1\varepsilon\ ,\ 2\varepsilon\ ,2\varepsilon\ ,\ 12\varepsilon\ ,\ 12\varepsilon]$$

and for **real**(64) case:

$$\Delta\Sigma = [<1\varepsilon\ ,\ <1\varepsilon\ ,3\varepsilon\ ,\ 8\varepsilon\ ,\ 3\varepsilon]$$

The relative errors are even smaller while the absolute errors are comparable. For Eq.10 and the case of $N = 30$ from the original reference, the 29 relative errors (and one absolute error for the smallest singular value which was close to zero) showed a similar to that just seen for Eq.9. So it was accepted that at least in the calculation of singular values, correctness was verified and accuracy goals were met.

A more detailed verification of correctness and accuracy looked at the results of Eq.12 with the numerical experiments done with Eq.9 and Eq.10. In these test cases, the scaled worst error ∞-norm δA of Eq.12 never exceeded $O(kN\varepsilon)$, where k was a small integer, and often fractional. This was better that the theoretical error and on this basis, it was accepted that correctness was verified and accuracy goals were met. For the size of matrices used in the original

reference [2], such accuracy is quite acceptable even for **real**(32) data. But for the size of the matrices involved in the numerical experiments seen earlier, it is only an acceptable accuracy with **real**(64) data, being unreliable for **real**(32) data. That inaccuracy for even medium-sized matrces is a problem with the underlying algorithm, not the Chapel implementation, and is one of the reasons why more accurate alternatives are used today such as those in (say) the LAPACK [4] toolkit.

The curves of Fig.3 used the elapsed run-time, R_C of Fig.1, to calculate the speedup for the SVD, a time which includes some start-up overhead H, where H was 0.3±0.03 seconds depending on the experiment. Recalculating the speedup using $R_C - H$ (to remove that overhead) produced plots which looked identical to those in Fig.3, and on that basis, they are not reproduced here.

10. Comparison Against Fortran

For completeness, the SVD of a 1603×1603 test matrix computed by the serial **dsvdc.f** routine from the LINPACK [3] toolkit was compared against that computed by the Chapel implementation. The numerical results matched and the averaged run times showed that

a. this Fortran LINPACK [3] toolkit implementation took about 14.03 seconds, and

b. the Chapel implementation as discussed herein in serial mode took about 13.36 seconds.

Those numbers clearly show that the un-vectorised Chapel code code is just as fast as the Fortran code, at least to an engineering level of accuracy. These figures highlight that when it gives itself, or is given, a level playing field, Chapel now competes with Fortran in terms of performance. Further, nothing needs to change in the Chapel code to allow it to run many times faster, in parallel.

For the record, the compiler used was **gfortran** at the same revision as that of the back-end **gcc**, i.e. version 4.8.5. It was used as

gfortran -O3 -mfma -msse4.2 -fno-math-errno ...

to ensure that the comparison was as fair as possible. In fact, Fortran was given a slight edge as those options achieve some small level of vectorisation in the code. Chapel on the other hand, has no vectorisation capability currently although some manual loop unrolling has been done in inner products and Givens rotation computations. The numerical experiments were run on the same Intel Xeon E5-2650v4s used for the speedup experiments.

11. Conclusion

This paper has discussed in detail the parallelised Chapel implementation of the classical SVD. Only matrices made up of what Chapel calls **real**(w) floating point numbers were considered, **complex**(w) were ignored. Not a better SVD, but the same SVD except that it was parallelised, albeit with the restriction that it is not a distributed implementation. It runs on a single Chapel **locale**. All numerical experiments were made with the same test cases as the original reference to ensure that the same rigorous validation standards used in the original paper applied to the Chapel implementation that came out of this exercise.

The numerical experiments showed both consistent parallelisation and good performance. Rewriting the classical SVD to suit Chapel's row-major ordering dominated the work load in the redesign of the floating point operations. The original objectives and constraints of the exercise were met by Chapel, especially that of the code being a readable exposition of the underlying computational mathematics. That code contained no obtrusive parallelisation hints or **pragma**s seen in some other parallel programming languages. It also avoided the complexity of having to revert to Fortran-like column-major matrix ordering that Chapel's own **dmapped** feature can provide, or the confusion and overhead of working with matrix transpose temporary arrays and the copying back and forth associated with their use.

Chapel's ease of expression of classical programming concepts made implementation straightforward and its high level of abstraction avoided the need to use obscure low-level features to extract performance. Chapel certainly delivered on its promise of programmer productivity with most of the effort for the implementation being spent on the development of the algorithm mathematics rather than on language or interface issues. The 1-based indexing used in the classical SVD (and every reputable linear algebra text) easily mapped to the same within Chapel. With Chapel storing arrays in row-major ordering, the classical SVD needed some redesign to obtain good data locality access patterns and maximise cache hits during memory access. Once the application of Givens rotations moved to complex space, the performance problems in the iterative diagonalisation resolved themselves.

During any Givens rotation and the more accurate Householder reflection of the LAPACK [4] toolkit, *safe scaling* techniques [7] should in the future be incorporated into this Chapel implementation. They would improve the robustness of the computation across the board, not just in extreme cases. It may even make it more effective for **real**(32) data. The floating point operations that would have to be done as part of such a task would demand the use of something like the Chapel **ieee754** module [8].

As of the time of writing, Chapel is on the brink of being able to vectorise code. Once that situation happens, the vectorisation of the algorithm for the operations in both Eq.2 and Eq.5 should improve both the parallel and serial run-times well beyond the limited super-scalar performance delivered by the loop unrolling done currently. Rather than make the compiler take all the responsibility for vectorisation, that task would be simpler if Chapel has a true short **vector** type that maps closely to Single Instruction Multiple Data (SIMD) instructions. On a more general note, once Chapel supports vectorisation in some shape or form, it may effectively preclude the need for the huge effort that goes into the development of production grade **basic** (or low-level) linear algebra toolkits such as GotoBLAS [10] or BLIS [11].

Aside from the vectorisation and Householder reflection just mentioned, future should look at all those things that were out of scope for this exercise. A native SVD implementation in Chapel could consider alternative algorithms [12] for better performance or accuracy or both. The latest releases of the EIGEN [13] C++ template library, LAPACK [4] toolkit, the GSL (GNU Scientific Library) [14], or **libflame**[15,16] would be ideal sources of inspiration. These alternatives might include the Jacobi SVD for matrices of a small to medium size such as were used in the earlier numerical experiments,

or the more complex divide-and-conquer algorithms that have appeared in the 1990s for larger matrices. Multi-**locale** (or distributed) implementations also need to be considered especially for larger matrices. And then there is also the need to handle sparse matrices which might need a different algorithm altogether!

The use of the 4.8.5 release of the GNU compilers is suboptimal, but conservative. For those chasing superior performance, not really one of the goals of this exercise, newer releases of those compilers or even better, the use of the LLVM back-end in Chapel would be more likely to help achieve their goal.

The experience of this exercise, and that of others who re-factor numerical algorithms with Chapel with a view to their parallelisation should be collated into a single document. Having such a reference might help to ensure that others will not have to go through the same, sometimes painful and time-consuming, learning process.

12. Acknowledgments

The authors are highly appreciative of the assistance from the entire Chapel development team, especially the support from Brad Chamberlain and Michael Ferguson.

The first author is particularly grateful for some insightful emails on zero-based array subscripts from Dr Martin Richards, several invaluable discussions on Chapel with Emeritus Professor Jan Hext, and a few probing and perceptive comments from Jeff Hammond that were crucial to exposing the final performance hurdle within the iterative diagonalisation stage.

Finally, the authors thank the anonymous reviewers who picked up on some areas where tighter language or better explanations were needed.

13. REFERENCES

[1] Chamberlain BL (2015), *Chapel*, In: Programming Models for Parallel Computing, eds. Pavan Balaji, MIT Press, ISBN 978-0-262-52881-8.

[2] Golub GH, Reinsch C (1970), *Singular Value Decomposition and Least Squares Solutions*, Numer. Math. **14**, pp403-420.

[3] Dongarra JJ, Bunch JR, Moler CB, Stewart GW (1979), *LINPACK User's Guide*, SIAM: Philadelphia, PA. ISBN 0-89871-172-X.

[4] Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999), *LAPACK Users' Guide (3rd ed.)*, SIAM: Philadelphia, PA. ISBN 0-89871-447-8.

[5] Beyls K, D'Hollander E (2009), *Refactoring for Data Locality*, IEEE Computer **42**(2), pp62-71.

[6] Van Zee FG, van de Geijn RA, Quintana-Orti G (2014), *Restructuring the Tridiagonal and Bidiagonal QR Algorithms for Performance*, ACM Transactions on Mathematical Software **40**(3), pp1-34.

[7] Anderson E (2017), *Algorithm 978: Safe Scaling in the Level 1 BLAS*, ACM Transactions on Mathematical Software **44**1, pp1-28.

[8] McGuckin D, Harding P (2019), *I3 - an IEEE754 Introspection Toolkit Overview*, Internal Report 19-CH-002, Pacific ESI 2019.

[9] IEEE Standard for Floating Point Arithmetic: ANSI/IEEE Std 754-2019 (2019), The Latest Revision of the IEEE Std 754.

[10] Goto KJ, van de Geijn RA (2008), *High Performance Implementation of the Level3 BLAS*, ACM Transactions on Mathematical Software **35**(1), pp1-14.

[11] Low TM, Igual F, Smith T, Quintana E (2016), *Analytical Modeling is Enough for High-Performance BLIS*, ACM Transactions on Mathematical Software **43**(2), pp1-18.

[12] Berry MW, Mezher D, Philippe B, Sameh A (2005), *Parallel Algorithms for the Singular Value Decomposition*, In: Handbook of Parallel Computing and Statistics, eds. Errisos John Kontoghiorghes. Chapman and Hall (CRC Press) 2005. ISBN 978-0824740672.

[13] Jacob B, Guennebaud G (2020), *EIGEN - a C++ Library*, See *http://eigen.tuxfamily.org*.

[14] Galassi M, Davies J, Theiler J, Gough B, Jungman G, Alken P, Booth M Rossi F, Ulerich R (2019), *The GNU Scientific Library Reference Manual*, See *https://www.gnu.org/software/gsl/doc/latex/gsl-ref.pdf* .

[15] Van Zee FG, Chan E, van de Geijn RA, Quintana-Orti ES, Quintana-Orti G (2009), *The libflame Library for Dense Matrix Computations*, IEEE Computing in Science and Engineering **11**(6), pp56-63.

[16] Van Zee FG, Chan E, van de Geijn RA (2011), *libflame*, In: Encyclopedia of Parallel Computing, eds. David Padua. Springer, Boston, MA. ISBN 978-0-387-09765-7.