WARWICK
THE UNIVERSITY OF WARWICK

Manuscript version: Author's Accepted Manuscript
The version presented in WRAP is the author's accepted manuscript and may differ from the published version or Version of Record.

Persistent WRAP URL:
http://wrap.warwick.ac.uk/160567

How to cite:
Please refer to published version for the most recent bibliographic citation information. If a published version is known of, the repository item page linked to above, will contain details on accessing it.

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions.

Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Publisher's statement:
Please refer to the repository item page, publisher's statement section, for further information.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk.

warwick.ac.uk/lib-publications

Scalable Many-core Algorithms for Tridiagonal Solvers

G.D. Balogh
Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Budapest, Hungary

T.S. Flynn
Department of Computer Science, University of Warwick, Coventry, UK

S. Laizet
Department of Aeronautics, Imperial College London, London, UK

G.R. Mudalige
Department of Computer Science, University of Warwick, Coventry, UK

I.Z. Reguly
Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Budapest, Hungary

Abstract—
We present a novel distributed memory Tridiagonal solver library, targeting large-scale systems based on modern multi-core and many-core processor architectures. The library uses methods based on both approximate and exact algorithms. Performance comparisons with the state-of-the-art, using both a large Cray EX system and a GPU cluster show the algorithmic trade-offs required at increasing machine scale to achieve good performance, particularly considering the advent of exascale systems.

Index Terms: linear solvers, high performance computing.

INTRODUCTION

Tridiagonal systems of equations arise in numerous fields particularly as part of the numerical approximation of multidimensional Partial Differential Equation (PDEs). They frequently appear in Computational Fluid Dynamics (CFD), Computational Electro-Magnetics (CEM), computational finance and image processing. Many industrial and research problems require the solution of a large number of independent tridiag-

 Published by the IEEE Computer Society © 2021 IEEE

onal systems, often in multiple dimensions. They offer significant opportunities for exploiting the massive parallelism available on modern multi-core CPU and many-core GPU devices. With the advent of such hardware, recent work [1] re-examined the choice between different tridiagonal solution algorithms (Thomas [2], PCR [3] and Hybrid). However, many real-world problems require such algorithms to work efficiently over multiple CPU/GPU devices due to the need for compute and memory resources beyond a single node.

A good example is high-fidelity simulations such as the ones performed with the Xcompact3d [4] framework, requiring the solution of up to 150 batches of tridiagonal systems at each time step, to compute derivatives and interpolations using implicit high-order finite-difference schemes. For a production problem with 1024^3 mesh nodes, which represents for instance a wind farm of several kilometers squared with a mesh node every 2 meters, this would require a cluster with more than 80 GPUs to solve 1024^2 systems, each with length of 1024 for a single batched solve (of which there are 150 per timestep). Looking at exascale systems, such simulations will be based on 100-1000 billion mesh nodes, and performed with 10-100 million cores for hundreds of thousands time steps.

Such problems mean that tridiagonal solver algorithms over distributed memory for these multi-core/many-core devices still require careful investigation in terms of performance and especially scalability. This is imperative in the current run-up to exascale systems in high performance computing where the software capabilities of exploiting such systems crucially depend on the scalability of numerical simulation applications and their underpinning algorithms. In this article we investigate the state-of-the-art in multi-core/many-core algorithms for tridiagonal solvers for distributed-memory systems and re-examine the algorithmic trade-offs required at increasing machine scale to achieve good performance. The insights lead to the development of a new, highly scalable implementation extending the single-node work of László et al [1].

This article is divided into three parts. The first presents the core algorithms for the solution of tridiagonal systems. The second investigates their extension to distributed memory-based systems and implications to performance at increasing scale. Third, the best implementations are used to solve a number of large-scale problems, analysing performance on CPU and GPU clusters. We conclude the article with lessons learnt, describing the current release of the solver as an open source software library named Tridsolver[1].

TRIDIAGONAL SYSTEMS SOLVER ALGORITHMS

Tridiagonal systems solvers arise from the need to solve a system of linear equations as given in (1) or its matrix form of $Ax = d$ given in (2), where $a_0 = c_{N-1} = 0$.

$$a_i u_{i-1} + b_i u_i + c_i u_{i+1} = d, \quad i = 0, 1, \ldots, N-1 \tag{1}$$

$$\begin{bmatrix} b_0 & c_0 & 0 & \ldots & 0 \\ a_1 & b_1 & c_1 & \ldots & 0 \\ 0 & a_2 & b_2 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & a_{N-1} & b_{N-1} \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{N-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ \vdots \\ d_{N-1} \end{bmatrix} \tag{2}$$

The solution to such systems is well known. Thomas [2] presented a sequential algorithm while Cyclic Reduction (CR) [5] and Parallel Cyclic Reduction (PCR) [3] are inherently parallel. The latter has been used extensively to implement solvers on GPUs [1], [6]. Additionally, combinations of Thomas and PCR have been used in a hybrid algorithm demonstrating better performance in several cases [1], [7]. In most applications, the tridiagonal systems are scalar where there is only one unknown per grid point. However multiple unknowns leading to block-tridiagonal structures do occur in areas such as CFD. In this article we focus on scalar tridiagonal systems, noting that the same algorithms extend naturally to block-tridiagonal systems.

The *Thomas Algorithm* (Algo. 1), is a specialised form of Gaussian elimination. It consists of a forward pass to eliminate the lower diagonal elements, a_i of the tridiagonal matrix, by adding a multiple of the row above. A backward pass then follows by using the modified c_i values from the last index to the first. This algorithm is inherently serial as each iteration of the loops

[1] https://github.com/OP-DSL/tridsolver/ doi: 10.5281/zenodo.5564551

has a dependency on the previous iteration, taking $2N$ steps.

Algorithm 1 `thomas`(a, b, c, d)

```
 1: d_0^* ← d_0/b_0
 2: c_0^* ← c_0/b_0
 3: for i = 1, 2, ..., N − 1 do
 4:     r ← 1/(b_i − a_i c_{i−1}^*)
 5:     d_i^* ← r(d_i − a_i d_{i−1}^*)
 6:     c_i^* ← r c_i
 7: end for
 8: for i = N − 2, ..., 1, 0 do
 9:     d_i ← d_i^* − c_i^* d_{i+1}
10: end for
11: return d
```

In contrast, the *PCR algorithm* (Algo. 2), assumes the matrix is normalized so that $b_i = 1$ and then for each matrix row i, subtracts multiples of rows $i \pm 2^0, 2^1, 2^2, ..., 2^{P-1}$, where P is the smallest integer such that $2^P \geq N$. Each iteration of the PCR algorithm reduces each of the current systems $(a^{(p-1)}, c^{(p-1)}, d^{(p-1)})$ into two systems of half the size $(a^{(p)}, c^{(p)}, d^{(p)})$. After P steps, all of the modified $a^{(P)}$ and $c^{(P)}$ coefficients become zero, leaving values for the unknowns u_i in $d_i^{(P)}$. In PCR, the iterations of the inner loop do not depend on each other, allowing multiple threads to be used to solve each tridiagonal system. PCR is more computationally expensive than the Thomas algorithm. Nevertheless it is well suited for implementations on modern multi-core/many-core architectures with high computational capabilities. The *CR algorithm* is similar to PCR, but consists of a forward and backward pass. The forward pass of CR is the same as the PCR algorithm but with an additional reverse pass that performs a back solve. This results in fewer operations overall but exhibits less parallelism and requires twice as many passes.

Algorithm 2 `pcr`(a, c, d)

```
 1: for p = 1, 2, ..., P do
 2:     s ← 2^{p−1}
 3:     for i = 0, 1, ..., N − 1 do
 4:         r ← 1/(1 − a_i^{(p−1)} c_{i−s}^{(p−1)} − c_i^{(p−1)} a_{i+s}^{(p−1)})
 5:         a_i^{(p)} ← −r(a_i^{(p−1)} a_{i−s}^{(p−1)})
 6:         c_i^{(p)} ← −r(c_i^{(p−1)} c_{i+s}^{(p−1)})
 7:         d_i^{(p)} ← r(d_i^{(p−1)} − a_i^{(p−1)} d_{i−s}^{(p−1)} −
                    c_i^{(p−1)} d_{i+s}^{(p−1)})
 8:     end for
 9: end for
10: return d^{(P)}
```

Hybrid Algorithms

Combining Thomas with PCR has shown to result in the best performance on GPUs [1]. The tridiagonal system is split into subsystems of size M, each of which is handled by a separate thread. Each subsystem is solved using a modified Thomas algorithm where, in a forward pass (see Algo. 3) each unknown is expressed in terms of two unknowns, u_0 and u_{M-1}:

$$a_i^* u_0 + u_i + c_i^* u_{M-1} = d_i^*, \quad i = 1, 2, ..., M-2.$$

The forward pass results in a reduced tridiagonal system made up of the unknowns at the beginning and end of each subsystem, as can be seen in **Figure 1**. How this reduced system is solved has a significant impact on the overall performance of the solver. Options for solving this include the previously mentioned algorithms. Finally, the result from the reduced system, is substituted back into the individual subsystems, running a backward pass to solve each subsystem (see Algo. 4).

$$\begin{pmatrix}
\mathbf{1} & & & \boldsymbol{c_0^*} & & & & & & & & \\
a_1^* & 1 & & c_1^* & & & & & & & & \\
a_2^* & & 1 & c_2^* & & & & & & & & \\
\boldsymbol{a_3^*} & & & \mathbf{1} & \boldsymbol{c_3^*} & & & & & & & \\
\hline
 & & & \boldsymbol{a_4^*} & \mathbf{1} & & & \boldsymbol{c_4^*} & & & & \\
 & & & & a_5^* & 1 & & c_5^* & & & & \\
 & & & & a_6^* & & 1 & c_6^* & & & & \\
 & & & & \boldsymbol{a_7^*} & & & \mathbf{1} & \boldsymbol{c_7^*} & & & \\
\hline
 & & & & & & & \boldsymbol{a_8^*} & \mathbf{1} & & & \boldsymbol{c_8^*} \\
 & & & & & & & & a_9^* & 1 & & c_9^* \\
 & & & & & & & & a_{10}^* & & 1 & c_{10}^* \\
 & & & & & & & & \boldsymbol{a_{11}^*} & & & \mathbf{1}
\end{pmatrix}
\begin{pmatrix}
\boldsymbol{u_0} \\ u_1 \\ u_2 \\ \boldsymbol{u_3} \\ \hline \boldsymbol{u_4} \\ u_5 \\ u_6 \\ \boldsymbol{u_7} \\ \hline \boldsymbol{u_8} \\ u_9 \\ u_{10} \\ \boldsymbol{u_{11}}
\end{pmatrix}
=
\begin{pmatrix}
\boldsymbol{d_0^*} \\ d_1^* \\ d_2^* \\ \boldsymbol{d_3^*} \\ \hline \boldsymbol{d_4^*} \\ d_5^* \\ d_6^* \\ \boldsymbol{d_7^*} \\ \hline \boldsymbol{d_8^*} \\ d_9^* \\ d_{10}^* \\ \boldsymbol{d_{11}^*}
\end{pmatrix}$$

Figure 1: Tridiagonal matrix split into 3 subsystems after the hybrid Thomas-PCR forward pass [1]. The reduced system is shown in bold and $M = 4$.

The Thomas-PCR implementation on NVIDIA GPUs developed by László et al [1] has a key advantage where up to a certain size of system, the entire subsystem can be stored in the registers of a warp (32 CUDA threads). It can outperform the other previously described algorithms, until the subsystem is too large to fit in a warp's registers. Once this limit is reached, a Thomas algorithm optimized for GPU

Algorithm 3 `hybrid_forward`(a, b, c, d)

1: **for** $i = 0, 1$ **do**
2: $\quad d_i^* \leftarrow d_i/b_i$
3: $\quad a_i^* \leftarrow a_i/b_i$
4: $\quad c_i^* \leftarrow c_i/b_i$
5: **end for**
6: **for** $i = 2, 3, ..., M-1$ **do**
7: $\quad r \leftarrow 1/(b_i - a_i c_{i-1}^*)$
8: $\quad d_i^* \leftarrow r(d_i - aid_{i-1}^*)$
9: $\quad a_i^* \leftarrow -ra_i a_{i-1}^*$
10: $\quad c_i^* \leftarrow rc_i$
11: **end for**
12: **for** $i = M-3, M-4, ..., 1$ **do**
13: $\quad d_i^* \leftarrow d_i^* - c_i^* d_{i+1}^*$
14: $\quad a_i^* \leftarrow a_i^* - c_i^* a_{i+1}^*$
15: $\quad c_i^* \leftarrow -c_i^* c_{i+1}^*$
16: **end for**
17: $r \leftarrow 1/(1 - c_0^* a_1^*)$
18: $d_0^* \leftarrow r(d_0^* - c_0^* d_1^*)$
19: $a_0^* \leftarrow ra_0^*$
20: $c_0^* \leftarrow -rc_0^* c_1^*$
21: **return** a^*, c^*, d^*

Algorithm 4 `hybrid_backward`(a^*, c^*, d^*)

1: $d_0 \leftarrow d_0^*$
2: **for** $i = 1, 2, ..., M-1$ **do**
3: $\quad d_i \leftarrow d_i^* - a_i^* d_0^* - c_i^* d_{M-1}^*$
4: **end for**
5: $d_{M-1} \leftarrow d_{M-1}^*$
6: **return** d

memory accesses was shown to outperform the Thomas-PCR hybrid.

Regardless of the algorithm used, solving tridiagonal systems on parallel systems is considered to be memory bandwidth bound, especially on GPUs [1]. Designing the memory access patterns of the algorithms to give coalesced and aligned memory accesses is a key issue.

Iterative Solutions

We have considered only algorithms providing exact solutions to tridiagonal systems. Iterative methods with approximate solutions are also available, such as the widely used Jacobi method. Such approaches are applicable to solve general systems resulting in matrices that are diagonally dominant, not only tridiagonal systems. The solution to $Ax = d$ is sought starting from an initial guess for the unknowns, iterating until a given convergence criterion is met. Algo. 5 details the Jacobi method applied to a tridiagonal system, where a, b and c are arrays holding the three diagonals and d holds the right hand side of the equation.

Algorithm 5 `jacobi`(a, b, c, d)

1: $p \leftarrow 1$
2: **while** Not Converged **do**
3: $\quad x_0^{(p)} \leftarrow \left(d_0 - c_0 x_1^{(p-1)}\right)/b_0$
4: $\quad$**for** $i = 1, ..., N-2$ **do**
5: $\quad\quad x_i^{(p)} \leftarrow \left(d_i - a_i x_{i-1}^{(p-1)} - c_i x_{i+1}^{(p-1)}\right)/b_i$
6: $\quad$**end for**
7: $\quad x_{N-1}^{(p)} \leftarrow \left(d_{N-1} - a_{N-1} x_{N-2}^{(p-1)}\right)/b_{N-1}$
8: $\quad$`checkIfConverged()`
9: $\quad p \leftarrow p + 1$
10: **end while**
11: **return** $x^{(p)}$

Each iteration requires 5 operations per grid point for tridiagonal matrices, with an additional cost of checking the convergence of the solution within a desired tolerance (approximate solution). So while an iteration of the Jacobi method is cheaper than a direct approach, multiple iterations are needed, with many iterations required for poorly conditioned systems.

DISTRIBUTED MEMORY ALGORITHMS

Several algorithms have been proposed for which the tridiagonal matrices can be split over multiple processes, with TridiagLU [8] providing a state-of-the-art implementation. Many of these algorithms divide the system into partitions and form a smaller decoupled tridiagonal system connecting the partitions (reduced system). In TridiagLU the partitioning is the MPI decomposition and each MPI process holds a single row from the reduced system.

The reduced system is solved iteratively using the Jacobi method. Instead of checking for convergence, an optional estimate of iterations for convergence can be provided to TridiagLU (to avoid the residual calculation with extra global communications). However, this limits its use to domains where it is possible to provide a good es-

timate of the iterations required. For cases where estimating an iteration count for convergence is not practical, TridiagLU can calculate a global norm to check for convergence at the expense of some performance.

TridiagLU also has the option to gather a reduced system, corresponding to one tridiagonal system, onto a single MPI process and solve it on that MPI process. The result is then scattered back to the relevant MPI processes after the reduced solve. Different reduced systems in the batch of tridiagonal systems, will be gathered to different MPI processes so that the load is balanced. Naturally, the use of global collectives degrades performance when using these options.

The new distributed memory tridiagonal solver that we present builds on the hybrid Thomas-PCR algorithm detailed in the previous section. We implemented multiple variations of this hybrid algorithm but the overall structure of the distributed tridiagonal solver can be summarized as follows. Each subsystem of size M belongs to a separate MPI process, which performs the hybrid Thomas-PCR forward pass. This produces a reduced system with two rows per MPI process. The solution to the reduced system is implemented in a number of ways, resulting in different performance characteristics over distributed memory systems. Once the reduced system is solved, the backward pass of the hybrid Thomas-PCR is performed on each MPI process.

The reduced system can be handled using several strategies. It can be gathered onto a single MPI process which then solves it and scatters the results back to the other MPI processes. This *gather-scatter* (GS) implementation can also be slightly modified to obtain an *allgather* (AG) implementation where the reduced system is gathered onto all MPI processes and then solved on each process. AG removes the need for the scattering of the results. Both GS and AG require excessive global communications which naturally leads to poor scaling.

The PCR or Jacobi methods can be used to avoid global collectives. *PCR* would follow the same algorithm as described previously, but with the addition of point to point MPI communications during each iteration of the algorithm. Therefore, it will carry out MPI communications with processes successively further away for the later iterations of the PCR solve. A further alternative is to use the iterative *Jacobi* method on the reduced system similar to the TridiagLU implementation to obtain an approximate solution to the reduced system. Again, there is the option to provide an estimated number of iterations or to check for convergence. The Jacobi solve for the reduced system has the advantage that it only requires MPI processes to communicate with their neighbours. However, if required to check for convergence, then its near neighbour communication advantage gets nullified as a global collective communication is needed for each iteration (or every n number of iterations). Considering the above approaches, the key advantage of PCR for the reduced system is that it avoids the need for collective communications while at the same time providing an exact solution.

An improvement to the forward pass of the hybrid Thomas algorithm (Algo. 3) is to combine it with the the forward of TridiagLU [8]. Algo. 6 normalises each row and forms the a^* column from Figure 1 and c_0^* for a subdomain resulting in a reduced system with one row per subdomain. This relaxes the need to express each unknown in terms of u_0 and u_{M-1}, instead each $u_i, i \in 1, ...M-1$ will be expressed with u_0 and u_{i+1}:

$$a_i^* u_0 + u_i + c_i^* u_{i+1} = d_i^*, \quad i = 1, 2, \ldots, M-1.$$

Although this introduces dependencies inside a partition, the backward substitution pass still requires only a single sweep without extra memory movements. Both the original and modified algorithms are trivially scalable since there is no communication involved. The reduced computational cost of the forward pass and the smaller reduced system size (one row per MPI process instead of two) leads to better overall performance, hence Tridsolver uses Algo. 6 on GPUs when the reduced system is solved with the Jacobi or PCR methods.

TRIDIAGONAL SYSTEMS IN 3D APPLICATIONS

A tridiagonal system by its nature represents a one dimensional problem, however, applications of interest are commonly 2 or 3 dimensional. Tridiagonal systems are formed in these applications by solving along one of the coordinate axes - and there will be as many *independent*

Algorithm 6 `forward_sweep`(a, b, c, d)

1: $d_1^* \leftarrow d_1/b_1$
2: $c_1^* \leftarrow c_1/b_1$
3: $a_1^* \leftarrow a_1/b_1$
4: $b_0^* \leftarrow b_0 - c_0 a_1^*$
5: $d_0^* \leftarrow d_0 - c_0 d_1^*$
6: $c_0^* \leftarrow -c_0 c_1^*$
7: **for** $i = 2, 3, ..., M-2$ **do**
8: $\quad r \leftarrow 1/(b_i - a_i c_{i-1}^*)$
9: $\quad d_i^* \leftarrow r(d_i - a_i d_{i-1}^*)$
10: $\quad a_i^* \leftarrow -r a_i a_{i-1}^*$
11: $\quad c_i^* \leftarrow r c_i$
12: $\quad b_0^* \leftarrow b_0^* - c_0^* a_i^*$
13: $\quad d_0^* \leftarrow d_0^* - c_0^* d_i^*$
14: $\quad c_0^* \leftarrow -c_0^* c_i^*$
15: **end for**
16: $r \leftarrow 1/(b_{M-1} - a_{M-1} c_{M-2}^*)$
17: $d_{M-1}^* \leftarrow r(d_{M-1} - a_{M-1} d_{M-2}^*)$
18: $a_{M-1}^* \leftarrow -r a_{M-1} a_{M-2}^*$
19: $c_{M-1}^* \leftarrow r c_{M-1}$
20: $d_0^* \leftarrow d_0^*/b_0^*$
21: $c_0^* \leftarrow c_0^*/b_0^*$
22: $a_0^* \leftarrow a_0/b_0^*$
23: **return** a^*, c^*, d^*

systems as there are discretization points along the other axes. For example, the Alternating Direction Implicit (ADI) [9] method, preferred in computational finance, works by repeatedly solving tridiagonal systems along different axes. In ADI the a_i, b_i, c_i, d_i coefficients are calculated for each grid point, in a way that matches the underlying data structure of the application; data for the diagonals are stored contiguously in either a row-major (Z is contiguous, Y, X are strided) or more commonly a column-major (X is contiguous, Y and Z are strided) format. This poses a challenge for algorithms that then solve multiple tridiagonal systems simultaneously; coefficients for an individual system will be laid out differently, depending on the direction of the solve. If we solve along the X direction (and use a column major format) for example, then coefficients for the same system are contiguous in memory, followed by the coefficients for the next system, etc. If however, we solve along the Z direction, then coefficients for the same row of different systems are laid out contiguously, followed by the next row, etc.

The TridiagLU library can only handle a data layout corresponding to a Z solve as described above, and so in a 3D application one would have to appropriately transpose data for X and Y solves - an operation notoriously difficult to do efficiently due to poor memory access patterns. On the other hand, the tridiagonal solver library developed in this work, (which we henceforth call the *Tridsolver* library), was designed from the outset to handle higher dimensional applications carrying out 1D solves in different directions - of course as with the approaches of László et al [1] the implementations are still impacted by memory access patterns.

A number of optimizations help improve performance on modern systems. On CPUs the Y and Z dimension solves (see next section) can be vectorized, by splitting the tridiagonal systems into strips of consecutive memory and adding compiler pragma `omp simd` on the appropriate loops. On the GPU a key issue is the uncoalesced memory accesses in the X dimension. Local transposition using vector shuffles [1] provide a solution, where threads of a warp cooperate to read a 32×16 or 32×8 (depending on single or double precision mathematics) block of the YZ plane at once. This corresponds to either 16 or 8 elements of 32 neighbouring tridiagonal systems. After loading this block of data, the elements of the tridiagonal system are not necessarily held by the thread solving that system. The *__shfl_xor()* intrinsic is then used to swap the elements to the correct threads. A similar operation is performed in reverse when storing intermediate values and the solution of the tridiagonal systems.

EVALUATION AND PERFORMANCE

To study the performance and scalability of the Tridsolver library, we designed benchmarks (published with the repository) for two of the UK's HPC systems: ARCHER2, a CrayEX system with AMD Rome CPUs (2×64 cores per node) and 256 GB of RAM, and Cirrus, a HPE/SGI system with 36 GPU nodes, each with 4×NVIDIA V100 16GB GPUs, interconnected with NVLink, and FDR Infiniband between nodes.

As a baseline, we compared against the TridiagLU library on the CPU - which only supports distributed memory parallelism with MPI.

For 3D problems, TridiagLU assumes the same coefficients from different systems are packed together, corresponding to a Z solve. We include an extra copy of the a, b, c coefficient arrays in our timing as these are overwritten by the solve algorithm, but the original values are required by our applications. For the solution of the reduced system, we evaluated both the exact solver approach with GS and the approximate iterative approach (Jacobi) which includes allreduce calls to determine whether the solution has converged.

We evaluated the performance of our library (marked with TridSlv) utilizing, for the reduced system solve, both exact solution approaches (with AG, GS, PCR) as well as the approximate iterative approach (Jacobi), including convergence checks. For the solution of the reduced system we evaluated solves in all directions, and directly compared to TridiagLU in Z. To avoid differences in convergence at increasing scale, we fixed the number of Jacobi iterations at 10 as done by Ghosh et al [8].

Weak Scaling - ARCHER2

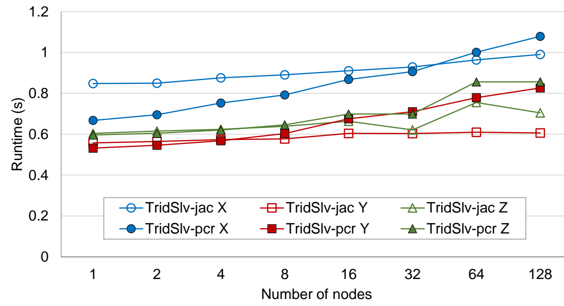
For weak scaling, where problem size increases with machine size, we picked 512^3 grid points per ARCHER2 node, a typical problem size used by frameworks such as Xcompact3D. Currently, ARCHER2 only has 1024 nodes in 4 cabinets, and considering that tridiagonal solves in various directions are completely independent, we tested weak scalability only along one "line" of nodes: for X solve $N \times 1 \times 1$ nodes and $(N*512) \times 512 \times 512$ grid points, for Y solve $1 \times N \times 1$ nodes and $512 \times (N*512) \times 512$ grid points, and for Z solve $1 \times 1 \times N$ nodes and $512 \times 512 \times (N*512)$ grid points. With pure MPI, we have 128 processes per node, which we distribute $4 \times 4 \times 8$ along the X,Y, and Z directions respectively.

Weak scaling performance along different directions varies significantly (see **Figure 2a**) - the X solve is the least amenable to parallelisation and vectorisation with the Thomas algorithm being up to $1.7\times$ slower than the Y solve due to the diagonals for each system is contiguous in memory when solving along the X dimension. The Y and Z solves lend themselves to trivial parallelisation. However, as the algorithm steps from row to row, the corresponding coefficients are separated by larger strides (more so in case of Z) leading to degraded TLB performance, as documented before [1]. The performance on a single node heavily depend on the possible memory bandwidth. On a single ARCHER2 node the hybrid Thomas forward step achieves 270.2 GB/s for X and Y solve and 225 GB/s for Z solve the backward achieves 307.7 GB/s, 275.8 GB6s 268.5 GB/s for X, Y, and Z solves respectively. Although the theoretical maximum for the AMD Rome CPU is 204.8 GB/s per socket the Triad (simple addition kernel) kernel in the BabelStream [10] benchmark achieves 288 GB/s as well. Comparing different solvers, as expected the Tridsolver Allgather (AG) and Gather-Scatter (GS) approaches (**Figure 2b**) have poor scaling efficiency (60%), and while TridiagLU GS scales somewhat better (48-94%) due to the distributed nature of reduced system solves, scaling efficiency remains low due to high communication costs. In contrast, the Jacobi approximate solver (jac) has excellent scalability (90-98%) due to its low volume neighbour-to-neighbour communication patterns. The Tridsolver with PCR for reduced system solve comes very close to Jacobi in terms of scaling efficiency - only falling behind at larger node counts. This is due to PCR having overall worse (long-distance) communication patterns, but communication volume scale logarithmicly with the number of processes along the solve dimension.

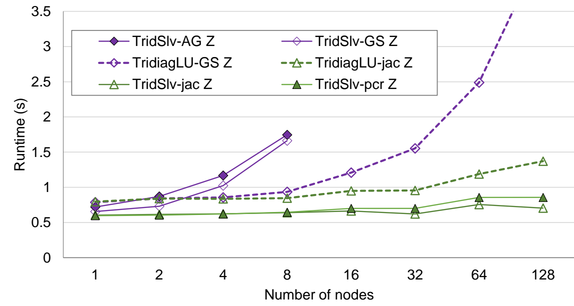
We observed that since forward and backward steps involve no communications, they scale trivially. The reduced system solve with Jacobi also shows good scaling with 92-96% efficiency. However, solving the reduced system with PCR sees a steady fall with a 70-74% scaling efficiency.

Strong scaling - ARCHER2

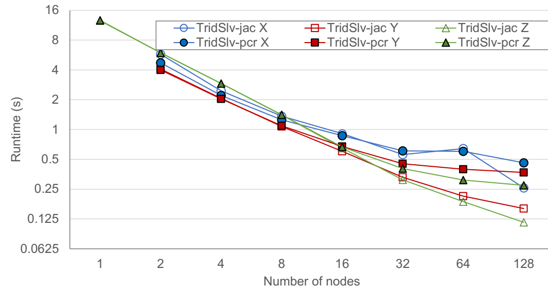
For strong scaling, where a large single global problem is solved at increasing machine scale, we used a grid size of 8192 in the direction of the solve, and 512 in other directions, allowing us to scale from 1 node to 128 nodes. We assigned $4 \times 4 \times 8$ processes per node in the X,Y, and Z directions respectively. **Figure 2c** shows the results in different directions and **Figure 2d** compares different solvers. Here, the logarithmic scale for the y axis reduces the visibility of the difference between the X solve and other solves. Nevertheless it is consistent with the slowdown



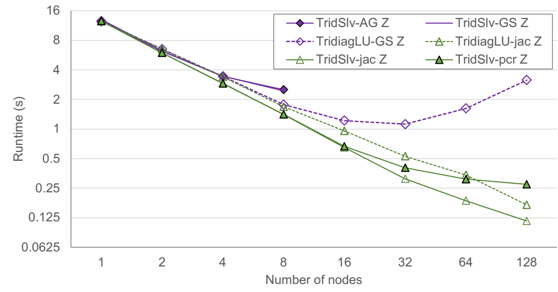
(a) Tridsolver (TridSlv) weak scaling in X,Y and Z-dims



(b) TridSlv vs TridiagLU weak scaling



(c) Tridsolver (TridSlv) strong scaling in X,Y and Z-dims



(d) TridSlv vs TridiagLU strong scaling

Figure 2: ARCHER2 scaling: (a),(b) - Weak-scaling, 512^3 grid points per node. (c),(d) - Strong-scaling, 8192 points in the direction of solve, and 512 in others. AG - AllGather, GS - Gather-Scatter

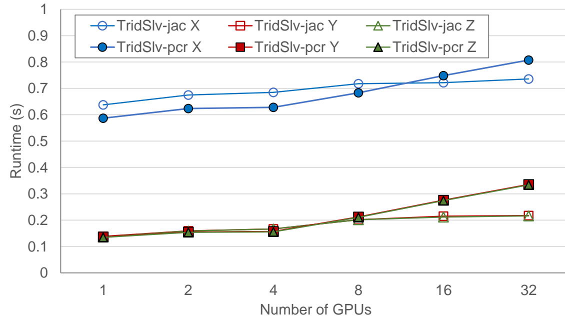
observed when weak scaling. Even superlinear scaling (102-108%) can be observed on up to 8 nodes, with both Jacobi and PCR, owing to the continuously reducing number of TLB and LLC misses. However, at larger scales communication costs dominate; using Jacobi for the reduced solve, efficiency drops to 80% above 32 nodes, and using PCR to 82-56% above 32 nodes. At 128 nodes for the Z solve, the reduced system solve phase accounts for 60% of total time with Jacobi, and 85% with PCR.

Comparing the scalability of different algorithms in Figure 2d, we see that the Tridsolver AG and GS variants slow down early and actually run out of memory due to the large size of the reduced system. The PCR solver shows competitive scaling compared to the approximate Jacobi methods - it is within a factor of 2, and scales further than the TridiagLU library.

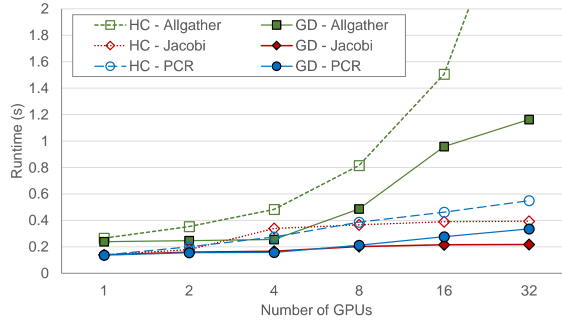
Weak Scaling - Cirrus

For weak scaling on GPUs, we kept the problem size per GPU at 512^3 in order to compare with the CPU results, and scaled the problem in the direction of the solve, only performing MPI decomposition in that direction. We did not compare against other libraries as we are not aware of other MPI-enabled GPU tridiagonal solver libraries. Tridsolver implementations support any MPI distribution by copying data to the host (we used MPT 2.22), then making transfers between CPUs, as well as GPU Direct-enabled MPI distributions (we used OpenMPI 4.1.0), where transfers take place directly between GPUs. Best results were achieved with GPU Direct.

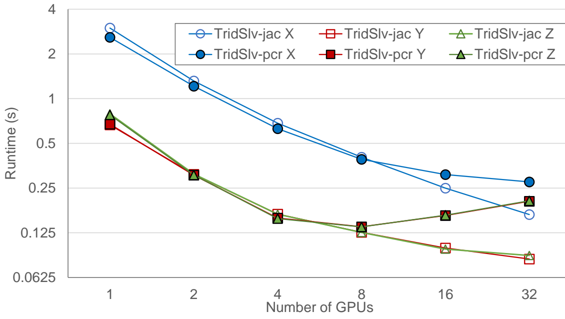
Looking at results from Cirrus, in **Figure 3a** we compare the performance when using Jacobi and PCR variants for solving the reduced system in different directions (note that Y and Z solves are virtually indistinguishable on the plot). As on the CPU, the performance of X solves is degraded by poor memory access patterns. On a single GPU X solve in Tridsolver achieves 458 GB/s bandwidth while the Y and Z solves achieve 731 GB/s and 739 GB/s respectively (the Triad kernel in the BabelStream [10] benchmark achieves 821 GB/s on a single V100 GPU). We compared Tridsolver on a single V100 GPU to the batch tridiagonal solver functions in the cuSPARSE. Currently cuSPARSE has two batch tridiagonal solver funciton: `cusparse<t>gtsv2StridedBatch()` uses the same memory layout as the X solve in Tridsolver and achieves 525.5 GB/s, while



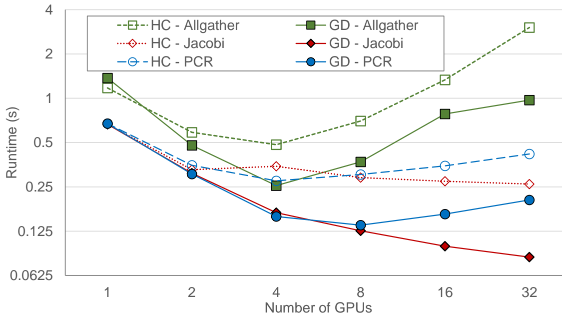
(a) Tridsolver (TridSlv) weak scaling in X,Y and Z-dims



(b) TridSlv, weak scaling, HC vs GD



(c) Tridsolver (TridSlv), strong scaling in X,Y and Z-dims



(d) TridSlv, strong scaling, HC vs GD

Figure 3: Cirrus scaling (MPI+CUDA) :(a),(b) - Weak-scaling, 512^3 points per GPU. (c),(d) - Strong-scaling, 2048 points in the direction of solve, 512 points in others. HC - host copy, GD- GPU direct.

`cusparse<t>gtsvInterleacedBatch()` comparable to the Z solve and achieves 725.6 GB/s. On Cirrus, there is a marked decline in parallel efficiency beyond 4 GPUs. Up to 4 GPUs, communications are done via the high-speed NVLink interconnect, but beyond that, there is inter-node communication through a slower Infiniband connection. Studying performance breakdowns in more detail, we see that in case of the Y and Z solves total time takes between $0.13 - 0.21$ (Jacobi) or $0.13 - 0.33$ (PCR) seconds. The computational part takes 0.13 seconds (forward and backward steps). For up to 4 GPUs the cost of communications is less than 8% and 4% of total runtime for Jacobi and PCR respectively, however beyond 4 GPUs, this increases to 22-27% for Jacobi and 27-53% for PCR.

Figure 3b compares performance of the baseline MPI implementation using explicit host copies (HC) with using GPU Direct (GD) - we also show the scalability of the Allgather (AG) version. AG clearly shows the impact of increasing communication volume as the number of GPUs increase, leading to dramatic slowdowns. On Jacobi and PCR, the GD version is up to $1.69\times$ faster. Note that on a single GPU the PCR and Jacobi versions are $1.8\times$ faster than the AG version. Since on a single node there is no need for a reduced system solve the performance improvement is purely due to the adoption of the new improved Thomas forward and backward pass.

Overall we see that a single GPU is $4.6\times$ faster than a single ARCHER2 node running with a pure MPI parallelization in the Y and Z directions. This difference arise from the overhead of the MPI communication on the CPU node and the bandwidth limitations of the two hardware. At 32 GPUs/nodes, this is reduced to $3\times$ for Jacobi and $2.1\times$ for PCR due to the comparatively worse communications scaling on the Cirrus GPU cluster.

Strong Scaling - Cirrus

The largest problem that can fit in a single GPU has 2048 points in the direction of solve and 512 in others - which then can be strong scaled up to 32 GPUs. Results in **Figure 3c** detail again the Y and Z solves only showing marginal

differences in performance. As before we see a drop in scaling efficiency beyond 4 GPUs, which are in a single node interconnected with NVLink; over 93% up to 4 GPUs, then 55-66% for Jacobi and 39-57% for PCR. As observed during weak scaling, communications become more of a bottleneck for the PCR solver: at 32 GPUs 86% of total time is communications, compared to Jacobi's 72%. The differences between HC and GD versions are even more prominent in strong scaling (see **Figure 3d**). GD is up to 3.25× faster.

CONCLUSION

In this article we investigated the state-of-the-art in multi-core/many-core algorithms for tridiagonal solvers for distributed-memory systems and re-examined the algorithmic trade-offs for obtaining better scaling and runtime performance at increasing machine scale. The exploration led to the development of an improved distributed-memory solver with scalable performance for large number of MPI nodes, based on the hybrid Thomas-PCR algorithm, giving exact solutions to the problem, by extending and augmenting a previous single-node library to execute over clusters of CPUs and GPUs. Further developments led to implementing a new improved Thomas-PCR forward pass and integrating iterative techniques, based on a Jacobi solver, which provided approximate solutions that can be used as an option for the solution of the reduced system resulting on the boundaries of MPI partitions.

Performance evaluation on a CrayEX system showed superior performance on realistic problem sizes specifically for ADI applications. The new solver with the Jacobi solver for the reduced system obtained 90-98% scaling efficiency. However, solving the reduced system with the PCR algorithm provided competitive performance. It achieved almost perfect scaling, tested up to 16 ARCHER2 nodes along the solve dimension, with the added advantage of providing an exact solution.

Execution on a GPU cluster demonstrated that the Jacobi and PCR solvers (for the reduced system solve) scaled with 93% efficiency up to 4 GPUs due to the high bandwidth single node interconnect. However efficiency reduced to 55-66% for Jacobi and 39-57% for PCR beyond this point. Further optimizations with a modified Thomas-PCR forward pass algorithm improved performance with a speedup of 1.8×.

The new tridiagonal solver library is currently being integrated into the OPS domain specific language [11] for the solution of structured-mesh problems. This will extend OPS's capabilities with implicit solutions on top of its existing explicit solvers used in frameworks such as OpenSBLI [12]. The new tridiagonal solver library is currently available as open source software[2].

Acknowledgment

This work was partly supported by the UK ExCALIBUR initiative. Its aim is to deliver research and innovative algorithmic development to harness the power of exascale computing. Gihan Mudalige was supported by the Royal Society Industry Fellowship Scheme (INF/R1/1800 12). The authors would like to thank EPCC for providing access to Cirrus and UKRI/EPSRC for access to ARCHER2 via the UK Turbulence Consortium (EP/R029326/1) and would like to thank EPSRC for its financial support via the CCP Turbulence (EP/T026170/1). This work was supported by the ÚNKP-21-3 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

REFERENCES

1. E. Laszlo, M. Giles, and J. Appleyard, "Manycore algorithms for batch scalar and block tridiagonal solvers," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 4, pp. 1–36, 2016. doi: 10.1145/2830568
2. L. Thomas, "Elliptic problems in linear differential equations over a network: Watson scientific computing laboratory," *Columbia Univ., NY*, 1949.
3. W. Gander and G. H. Golub, "Cyclic reduction—history and applications," *Scientific computing (Hong Kong, 1997)*, vol. 7385, 1997.
4. P. Bartholomew, G. Deskos, R. A. Frantz, F. N. Schuch, E. Lamballais, and S. Laizet, "Xcompact3d: An open-source framework for solving turbulence problems on a cartesian mesh," *SoftwareX*, vol. 12, p. 100550, 2020. doi: 10.1016/j.softx.2020.100550
5. R. A. Sweet, "A cyclic reduction algorithm for solving block tridiagonal systems of arbitrary dimension," *SIAM*

[2] https://github.com/OP-DSL/tridsolver/ doi: 10.5281/zenodo.5564551

Journal on Numerical Analysis, vol. 14, no. 4, pp. 706–720, 1977. doi: 10.1137/0714048

6. H. Kim, S. Wu, L. Chang, and W. W. Hwu, "A scalable tridiagonal solver for gpus," in *2011 International Conference on Parallel Processing*, 2011. doi: 10.1109/ICPP.2011.41 pp. 444–453.
7. Y. Zhang, J. Cohen, A. A. Davidson, and J. D. Owens, "Chapter 11 - a hybrid method for solving tridiagonal systems on the gpu," in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W. mei W. Hwu, Ed. Boston: Morgan Kaufmann, 2012, pp. 117–132. ISBN 978-0-12-385963-1. doi: 10.1016/B978-0-12-385963-1.00011-3
8. D. Ghosh, E. M. Constantinescu, and J. Brown, "Efficient implementation of nonlinear compact schemes on massively parallel platforms," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. C354–C383, 2015. doi: 10.1137/140989261
9. D. W. Peaceman and H. H. Rachford, Jr, "The numerical solution of parabolic and elliptic differential equations," *Journal of the Society for industrial and Applied Mathematics*, vol. 3, no. 1, pp. 28–41, 1955. doi: 10.1137/0103003
10. T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Gpu-stream v2. 0: Benchmarking the achievable memory bandwidth of many-core processors across diverse parallel programming models," in *International Conference on High Performance Computing*. Springer, 2016. doi: 10.1007/978-3-319-46079-6_34 pp. 489–507.
11. I. Z. Reguly, G. R. Mudalige, and M. B. Giles, "Loop Tiling in Large-Scale Stencil Codes at Run-Time with OPS," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 4, pp. 873–886, April 2018. doi: 10.1109/TPDS.2017.2778161
12. G. Mudalige, I. Reguly, S. Jammy, C. Jacobs, M. Giles, and N. Sandham, "Large-scale performance of a DSL-based multi-block structured-mesh application for Direct Numerical Simulation," *Journal of Parallel and Distributed Computing*, vol. 131, pp. 130 – 146, 2019. doi: 10.1016/j.jpdc.2019.04.019

Gábor D. Balogh is currently working toward the Ph.D. degree at the Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Budapest, Hungary. His research focuses on parallel algorithms for domain specific languages. Contact him at balogh.gabor.daniel@itk.ppke.hu.

Tobias S. Flynn, is a PhD Student at the University of Warwick's department of Computer Science. His research focuses on developing high-order computational fluid dynamics methods for multiphase flows targeting current and emerging high performance computing systems. He received his BSc in Computer Science from the University of Warwick. Contact him at t.flynn@warwick.ac.uk.

Gihan R. Mudalige, is Associate Professor (Reader) of High Performance Computing at the University of Warwick. His research focuses on the development and optimization of high-performance and scientific computing applications using high-level abstractions and DSLs. He holds a PhD in Computer Science from the University of Warwick and is a member of the ACM. Contact him at g.mudalige@warwick.ac.uk.

Sylvain Laizet, is a Reader in the Department of Aeronautics at Imperial College London. His research focuses on understanding turbulent flows and how to use them in various engineering applications. He is the main developer of the high-order finite-difference framework `Xcompact3d` dedicated to the study of turbulent flows on supercomputers. Contact him at s.laizet@imperial.ac.uk.

István Z. Reguly, is an Associate Professor at PPCU ITK, Hungary. His research interests include high performance scientific computing on many-core hardware and domain specific libraries. He received his PhD degree in Computer Science from the PPCU, Hungary and is a member of the IEEE. Contact him at reguly@itk.ppke.hu.