Article:
Parker, Joseph, Hill, Peter Alec orcid.org/0000-0003-3092-1858, Dickinson, David orcid.org/0000-0002-0868-211X et al. (1 more author) (2021) Parallel tridiagonal matrix inversion with a hybrid multigrid–Thomas algorithm method. Journal of Computational and Applied Mathematics. pp. 1-15. ISSN 0377-0427

https://doi.org/10.1016/j.cam.2021.113706

Parallel tridiagonal matrix inversion with a hybrid multigrid–Thomas algorithm method

J. T. Parker[a,b,*], P. A. Hill[c], D. Dickinson[c], B. D. Dudson[c]

[a] *United Kingdom Atomic Energy Authority, Culham Centre for Fusion Energy, Culham Science Centre, Abingdon, Oxon, OX14 3DB, UK*
[b] *Science and Technology Facilities Council, Rutherford Appleton Laboratory, Harwell Campus, Didcot OX11 0QX, UK*
[c] *York Plasma Institute, Department of Physics, University of York, Heslington, York YO10 5DD, UK*

Abstract

Tridiagonal matrix inversion is an important operation with many applications. It arises frequently in solving discretized one-dimensional elliptic partial differential equations, and forms the basis for many algorithms for block tridiagonal matrix inversion for discretized PDEs in higher-dimensions. In such systems, this operation is often the scaling bottleneck in parallel computation. In this paper, we derive a hybrid multigrid–Thomas algorithm designed to efficiently invert tridiagonal matrix equations in a highly-scalable fashion in the context of time evolving partial differential equation systems. We decompose the domain between processors, using multigrid to solve on a grid consisting of the boundary points of each processor's local domain. We then reconstruct the solution on each processor using a direct solve with the Thomas algorithm. This algorithm has the same theoretical optimal scaling as cyclic reduction and recursive doubling. We use our algorithm to solve Poisson's equation as part of the spatial discretization of a time-evolving PDE system. Our algorithm is faster than cyclic reduction per inversion and retains good scaling efficiency to twice as many cores.

Keywords: tridiagonal matrix inversion, multigrid, parallel computing

2010 MSC: 65M55, 65Y05

1. Introduction

Tridiagonal matrix inversion is an important operation with many applications, including in computational fluid dynamics [1], plasma physics [2], Poisson solvers [3], preconditioning of iterative solvers [4], cubic spline interpolation [5], and computer graphics [6]. It arises frequently in the discretization of partial differential equation systems on structured grids, particularly those involving the solution of elliptic equations, like Laplace's or Poisson's equation. The discretization of operators in one dimension using centred second-order finite differences leads to tridiagonal systems, while the discretization in two or more dimensions leads to block tridiagonal systems. Developing efficient solvers for tridiagonal matrix inversion is useful beyond one-dimensional systems however, as methods for equations in multiple dimensions are often based on one-dimensional approaches. For example, the Alternating Direction Implicit (ADI) method for implicit time advance inverts a tridiagonal system for each dimension independently. Similarly, Naulin's method [7] for inverting elliptic operators in two or three dimensions is based on iterative corrections to a one-dimensional solver.

Tridiagonal systems may be inverted optimally in serial using the Thomas algorithm, a special case of Gaussian elimination that requires only $\mathcal{O}(N_x)$ operations, for N_x the dimension size. The Thomas algorithm is inherently sequential: it consists of two passes – forwards and backwards through the matrix rows – where each step depends on the

*Corresponding author: joseph.parker@ukaea.uk

previous step. Elimination of unknowns is only possible because the passes reach the boundaries of the global domain, with the boundary rows coupling two unknowns, rather than three unknowns as in the rest of the domain.

Many solvers exist for parallel tridiagonal matrix inversion [see for example 8, §5.5 for a review]. A fast, approximate solution is given by the Parallel Diagonal Dominant (PDD) method [9, 10]. This decomposes the tridiagonal system as a series of subsystems, one for each processor, and treats the coupling between subsystems as small corrections. While this scales ideally, the approximation is not valid unless the coupling terms are indeed negligible.

Exact parallel solutions have a theoretical minimum run time of $\mathcal{O}(\log N_x)$ [8]. The first solvers to achieve this were cyclic reduction [3] and recursive doubling [11, 12]. These are direct solvers based on computing the LU factorization of the tridiagonal system from independent, and therefore parallelizable, components. In these solvers, the $\mathcal{O}(\log N_x)$ scaling arises from the tree-like movement of data.

The SPIKE algorithm [13, 14] is a recursive method for solving tridiagonal, and more general banded and block tridiagonal, systems. It is motivated by using a factorization based on a domain decomposition method which is more amenable to parallelization than the LU factorization used in cyclic reduction and recursive doubling. The SPIKE algorithm has two layers of solvers: an inner and an outer solver. The inner solver reconstructs the solution on local subdomains, given the solution from the outer solver, which solves a reduced system for the coupling of the subdomains. Since the reduced system takes the same form as the original system (in our case, the reduced system for a tridiagonal matrix is also tridiagonal), the inner solver is applied recursively.

In this paper, we introduce an algorithm with the same domain-decomposition factorization as the SPIKE and PDD algorithms. As our local subdomain systems are tridiagonal, we use the Thomas algorithm as an inner direct solver. However, rather than use a recursive direct solve as described in [13], we use multigrid, an iterative method, for the reduced system. The motivation for this is two-fold. Firstly, we wish to minimize data movement, and therefore consider an iterative method that only requires local guard cell swaps, rather than global communications. Secondly, we are interested in inverting tridiagonal systems as part of a larger initial value problem. We therefore have a good initial estimate for the solution – namely, the solution from the previous timestep – and wish to take advantage of this by using an iterative method. Iterative methods also allow control of convergence tolerances, while multigrid in particular has many parameters that can be tuned to optimize a specific simulation, such as number of levels, number of smoothing cycles, and different options for the smoothing, prolongation and refinement methods. Finally, as the operator to be inverted is often constant or slowly-varying throughout a simulation, many quantities in our algorithm can be cached, reducing the overall work. Where not strictly constant, corrections can be applied using an outer solver.

The local Thomas algorithm inversions require $\mathcal{O}(N_x/N_p)$ operations, where N_x and N_p is the total number of grid points and processors respectively. The multigrid method converges to a given tolerance in a fixed number of cycles, so (as we shall see in Sec. 2) the number of operations per processor is $\mathcal{O}(1)$ independent of problem size, while the number of guard cell communications grows slowly as $\mathcal{O}(\log N_p)$. In addition, the cost of convergence checking scales empirically as $\mathcal{O}\left(N_p^{5/4}\right)$. Thus the overall runtime of our algorithm is $T = \mathcal{O}(N_x/N_p) + \mathcal{O}(\log N_p) + \mathcal{O}\left(N_p^{5/4}\right)$. We find experimentally that the ideal scaling region persists across most core counts, with performance only degrading once there are ~ 8 points per processor.

This paper is structured as follows. In Sec. 2 we derive our hybrid method and discuss complexity and communication requirements. In Sec. 3 we present numerical experiments using different solvers in a plasma filament simulation using the BOUT++ package [15]: we compare our hybrid multigrid-Thomas method to parallel cyclic reduction, a pure multigrid

implementation, and a direct solver that replaces the multigrid component of our algorithm with a direct solve on a single core (requiring an additional gather/scatter communication). In Sec. 4 we summarize and discuss future work.

2. Hybrid multigrid–Thomas algorithm method

In this section we derive our hybrid algorithm. We begin by discussing the solution of local tridiagonal systems on subdomains in Sec. 2.1, and then derive the reduced system which couples the subdomains in Sec. 2.2. We describe our implementation of multigrid for the reduced system in Sec. 2.3 and the calculation of error tolerances in Sec. 2.4. In Sec. 2.5 we derive the theoretical runtime of our algorithm. Finally in Sec. 2.6 we discuss techniques for reducing the amount of communication in the algorithm.

2.1. Local solves with the Thomas algorithm

We solve the $n \times n$ tridiagonal linear system $Mx = f$,

$$\begin{pmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-2} & b_{n-2} & c_{n-2} \\ & & & & a_{n-1} & b_{n-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-2} \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{n-2} \\ f_{n-1} \end{pmatrix}. \tag{1}$$

We divide the domain x using p processors such that $n = mp$ and there are m points per processor, and the rows qm to $(q+1)m - 1$ are local to the qth processor,

$$\left(\begin{array}{ccc|ccc|ccc} b_0 & c_0 & & & & & & & \\ \ddots & \ddots & \ddots & & & & & & \\ & a_{qm-1} & b_{qm-1} & c_{qm-1} & & & & & \\ \hline & & a_{qm} & b_{qm} & c_{qm} & & & & \\ & & & \ddots & \ddots & \ddots & & & \\ & & & & a_{(q+1)m-1} & b_{(q+1)m-1} & c_{(q+1)m-1} & & \\ \hline & & & & & a_{(q+1)m} & b_{(q+1)m} & c_{(q+1)m} & \\ & & & & & & \ddots & \ddots & \ddots \\ & & & & & & & a_{mp-1} & b_{mp-1} \end{array}\right) \left(\begin{array}{c} x_0 \\ \vdots \\ x_{qm-1} \\ \hline x_{qm} \\ \vdots \\ x_{(q+1)m-1} \\ \hline x_{(q+1)m} \\ \vdots \\ x_{mp-1} \end{array}\right) = \left(\begin{array}{c} f_0 \\ f_1 \\ \vdots \\ f_{qm-1} \\ \hline f_{qm} \\ \vdots \\ f_{(q+1)m-1} \\ \hline f_{(q+1)m} \\ \vdots \\ f_{mp-1} \end{array}\right). \tag{2}$$

The terms outside the block diagonal, a_{qm} and $c_{(q+1)m-1}$, contribute to the equations in processor q's rows, but depend on terms local to the neighbouring processors, and therefore require communication. To emphasise this, we introduce

halo cells (denoted with superscript h),

$$\left(\begin{array}{ccc|cccc|cc}
\ddots & \ddots & \ddots & & & & & & \\
a_{qm-1} & b_{qm-1} & c_{qm-1} & & & & & & \\
 & & 1 & 0 & -1 & & & & \\
\hline
 & -1 & 0 & 1 & & & & & \\
 & & & a_{qm} & b_{qm} & c_{qm} & & & \\
 & & & & \ddots & \ddots & \ddots & & \\
 & & & & a_{(q+1)m-1} & b_{(q+1)m-1} & c_{(q+1)m-1} & & \\
 & & & & & & 1 & 0 & -1 \\
\hline
 & & & & & & -1 & 0 & 1 \\
 & & & & & & & a_{(q+1)m} & b_{(q+1)m} \\
 & & & & & & & \ddots & \ddots
\end{array}\right)
\left(\begin{array}{c}
\vdots \\
x_{qm-1} \\
x^h_{qm} \\
\hline
x^h_{qm-1} \\
x_{qm} \\
\vdots \\
x_{(q+1)m-1} \\
x^h_{(q+1)m} \\
\hline
x^h_{(q+1)m-1} \\
x_{(q+1)m} \\
\vdots
\end{array}\right)
=
\left(\begin{array}{c}
\vdots \\
f_{qm-1} \\
0 \\
\hline
0 \\
f_{qm} \\
\vdots \\
f_{(q+1)m-1} \\
0 \\
\hline
0 \\
f_{(q+1)m} \\
\vdots
\end{array}\right). \tag{3}$$

Focussing on the equations local to the qth processor, we have

$$M_q x = \begin{pmatrix}
1 & & & & \\
a_{qm} & b_{qm} & c_{qm} & & \\
 & \ddots & \ddots & \ddots & \\
 & & a_{(q+1)m-1} & b_{(q+1)m-1} & c_{(q+1)m-1} \\
 & & & & 1
\end{pmatrix}
\begin{pmatrix} x^h_{qm-1} \\ x_{qm} \\ \vdots \\ x_{(q+1)m-1} \\ x^h_{(q+1)m} \end{pmatrix}
=
\begin{pmatrix} 0 \\ f_{qm} \\ \vdots \\ f_{(q+1)m-1} \\ 0 \end{pmatrix}
+
\begin{pmatrix} x_{qm-1} \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}
+
\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ x_{(q+1)m} \end{pmatrix}. \tag{4}$$

We may solve for x by inverting M_q, which is local to processor q, to obtain

$$\begin{pmatrix} x^h_{qm-1} \\ x_{qm} \\ \vdots \\ x_{(q+1)m-1} \\ x^h_{(q+1)m} \end{pmatrix}
= M_q^{-1}
\begin{pmatrix} 0 \\ f_{qm} \\ \vdots \\ f_{(q+1)m-1} \\ 0 \end{pmatrix}
+ M_q^{-1}
\begin{pmatrix} x_{qm-1} \\ 0 \\ \vdots \\ 0 \\ 0 \end{pmatrix}
+ M_q^{-1}
\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ x_{(q+1)m} \end{pmatrix}. \tag{5}$$

This may also be written

$$x = M_q^{-1} f + \alpha x_{qm-1} + \beta x_{(q+1)m}, \tag{6}$$

where $x = (x^h_{qm-1}, x_{qm}, \ldots, x_{(q+1)m-1}, x^h_{(q+1)m})$, $f = (0, f_{qm}, \ldots, f_{(q+1)m-1}, 0)$, and α and β are the first and final columns of M_q^{-1} respectively. Note also that $M_q^{-1} f$, α and β are all calculated using only data that is local to processor q. Moreover M_q^{-1}, α and β only change when the original operator M changes, and f only changes when the right-hand side changes. That is, $(M_q^{-1} f)$, α, β and f are always constant within a timestep, and, when M is time-independent, M_q^{-1}, α and β are constant throughout the whole simulation.

Given vectors $(M_q^{-1} f)$, α and β, equation (6) allows us to construct the full solution on processor q from the values x_{qm-1} and $x_{(q+1)m}$. These values are respectively the final interior point on the processor below and the first interior point on the processor above. We also note from (5) that processor q's first and last interior points x_{qm} and $x_{(q+1)m-1}$ depend on the neighbouring points x_{qm-1} and $x_{(q+1)m}$ only. Other interior points, shown by ellipses in (5), are never required. We may therefore construct the solution on every processor using a reduced grid that contains only the points either side of processor boundaries, the $2(p-1)$ variables $\{\ldots, x_{qm-1}, x_{qm}, x_{(q+1)m-1}, x_{(q+1)m}, \ldots\}$. Moreover, it is

only necessary to solve for the first interior points in each domain, as we can reconstruct the last interior points using (5). For example, if processor q knows its first interior point x_{qm} and its up-neighbour's first interior point $x_{(q+1)m}$, we may calculate x_{qm-1} from the first (interior) row of (5). We thus solve on the reduced grid

$$\boldsymbol{X} = (X_0, X_1, \ldots, X_q, \ldots, X_{p-1}, X_p)^T \equiv (x_0, x_m, \ldots, x_{qm}, \ldots, x_{(p-1)m}, x_{mp-1})^T. \tag{7}$$

This is a grid of $(p+1)$ points: the p first interior points, plus the final interior point on the final processor. Including the final point on this grid means we do not need to modify the boundary conditions from the original problem, since both x_0 and x_{mp-1} are on both the full and the reduced grid. Moreover, multigrid requires grids of size $2^k + 1$, so this choice allows us to use processor counts of 2^k (rather than the more awkward $2^k - 1$).

2.2. Equations for the reduced system.

To derive equations for the reduced grid $\boldsymbol{X}$, we consider the first and last interior rows of (5),

$$x_{qm} = r_q^l + \alpha_q^l x_{qm-1} + \beta_q^l x_{(q+1)m}, \tag{8a}$$

$$x_{(q+1)m-1} = r_q^u + \alpha_q^u x_{qm-1} + \beta_q^u x_{(q+1)m}, \tag{8b}$$

where l and u denote the lower and upper interface respectively, $r_q^l = (M_q^{-1} f)_{qm}$, $r_q^u = (M_q^{-1} f)_{(q+1)m-1}$, $\alpha_q^l = \alpha_{qm}$, $\alpha_q^u = \alpha_{(q+1)m-1}$, $\beta_q^l = \beta_{qm}$ and $\beta_q^u = \beta_{(q+1)m-1}$. We may substitute the lower interface variables in favour of $\boldsymbol{X}$ elements, to yield

$$X_q = r_q^l + \alpha_q^l x_{qm-1} + \beta_q^l X_{q+1}, \tag{9a}$$

$$x_{(q+1)m-1} = r_q^u + \alpha_q^u x_{qm-1} + \beta_q^u X_{q+1}. \tag{9b}$$

To obtain an equation solely for X_q it is sufficient to eliminate x_{qm-1} from (9a). To do this, we consider the corresponding equations on processor $q-1$,

$$X_{q-1} = r_{q-1}^l + \alpha_{q-1}^l x_{(q-1)m-1} + \beta_{q-1}^l X_q, \tag{10a}$$

$$x_{qm-1} = r_{q-1}^u + \alpha_{q-1}^u x_{(q-1)m-1} + \beta_{q-1}^u X_q. \tag{10b}$$

Eliminating $x_{(q-1)m-1}$ between these, we obtain an expression for x_{qm-1},

$$x_{qm-1} = \left(r_{q-1}^u - \frac{\alpha_{q-1}^u}{\alpha_{q-1}^l} r_{q-1}^l \right) + \frac{\alpha_{q-1}^u}{\alpha_{q-1}^l} X_{q-1} + \left(\beta_{q-1}^u - \frac{\alpha_{q-1}^u}{\alpha_{q-1}^l} \beta_{q-1}^l \right) X_q. \tag{11}$$

Substituting this into (9a), we obtain

$$\begin{aligned} X_q &= \frac{1}{\Delta} \left[r_q^l + \alpha_q^l \left(r_{q-1}^u - \frac{\alpha_{q-1}^u}{\alpha_{q-1}^l} r_{q-1}^l \right) + \alpha_q^l \frac{\alpha_{q-1}^u}{\alpha_{q-1}^l} X_{q-1} + \beta_q^l X_{q+1} \right], \\ \Delta &= 1 - \alpha_q^l \left(\beta_{q-1}^u - \frac{\alpha_{q-1}^u}{\alpha_{q-1}^l} \beta_{q-1}^l \right), \end{aligned} \tag{12}$$

which is simply X_q as a linear combination of X_{q-1}, X_{q+1} and a constant, *i.e.*, it is a tridiagonal system which we write $AX = g$. In (12), terms with subscript $q-1$ are not local to processor q and must be communicated from processor $q-1$. The terms containing α and β depend on the system matrix M, so may be calculated and communicated once by processor $q-1$, and then stored on processor q. Terms containing r depend on f and so must be calculated and communicated every timestep.

Algorithm 1: A sketch of 2-level multigrid

while *residual* $\|g - A\hat{X}\|$ *is too large* **do**

smooth, *i.e.*, perform a few iterations of a method like Jacobi or Gauss–Seidel to improve $\hat{X}$, the approximate solution to $AX = g$;

calculate the residual $r = \|g - A\hat{X}\| = \|A(X - \hat{X})\|$;

restrict the residual r by approximating it onto a coarser grid;

obtain an approximation solution to $A^c e = r^c$, where $e = X - \hat{X}$, and A^c and r^c are approximations to the original operator and the residual on the coarser grid;

prolong the solution e by interpolating it onto the original grid;

update the approximate solution $\hat{X}$ to $\hat{X} + e \approx X$;

end

Note also that α^l is an element from the inverse of a tridiagonal matrix, and is only zero if α^u is also zero. If $\alpha^l_{q-1} = \alpha^u_{q-1} = 0$, we use (10b) directly to eliminate x_{qm-1} from (9a). This yields the same expression (12), but with the ratio $\alpha^u_{q-1}/\alpha^l_{q-1} = 0$.

Equation (12) defines a tridiagonal system for X_q. The size of the system is $(p+1) \times (p+1)$, so grows with the number of processors, even though the underlying system for x_i has fixed size $n \times n$. This means that the reduced system and its properties change as we vary the number of processors. For example, if we were to solve for X_q using an iterative method, we would expect the number of iterations required to reach a tolerance to change as we change the number of processors. This is in contrast to more conventional approach of parallelizing serial algorithms where system properties should not depend on processor count.

2.3. Multigrid

We now consider solving system (12) for $\boldsymbol{X}$ for a fixed problem size $n \times n$. We consider iterative methods as these typically require nearest neighbour halo cell communications, rather than global collectives. At high core counts, we expect the algorithm to be latency-bound – the limit to performance is the rate of passing small amounts of data, rather than the rate at which work is performed. In this regime, the total number of iterations is a better measure of the algorithm's overall cost than complexity.

Let us first consider using a simple iterative method, like Jacobi or Gauss–Seidel. The rate of convergence of these methods depends on the largest eigenvalue of the system matrix A, but the number of iterations required to reach a tolerance typically increases with increasing problem size. Although our original problem is of fixed size, as we increase the number of processors, the size of the reduced problem for $\boldsymbol{X}$ increases. We find the increased number of iterations required for convergence offsets the speed-up from increased parallelism, and the algorithm does not scale with simple iterative approaches. We therefore consider multigrid methods, which typically require many fewer iterations than simple iterative methods for large problem sizes.

Multigrid originated in the 1960's and 1970's with the theoretical work of Fedorenko [16], Bakhvalov [17] and Hackbusch [18], and the numerical work of Brandt [19, 20] (see [21] for a brief overview of multigrid development). Multigrid has since grown into a widely-developed subject [22, 23, 24]. The multigrid method is motivated by the observation that when solving $AX = g$ by iterative methods, the long wavelength contributions to the residual $r = g - A\hat{X}$ (where $\hat{X}$ is

the approximate solution) decay much more slowly than short, grid-scale contributions. We can write r as the right-hand side of an equation for the error $e = X - \hat{X}$,

$$Ae = AX - A\hat{X} = g - A\hat{X} = r. \tag{13}$$

If we solve $Ae = r$ on a grid that is coarser than the original grid, then errors that are long wavelength on the original grid are now shorter wavelength relative to the coarse grid, and therefore decay more quickly. Taking the solution e to (13) for the coarse grid and projecting back onto the original grid, we obtain an updated approximation for $\hat{X} \to \hat{X} + e \approx X$ which is much improved at long wavelengths. An outline of multigrid with two levels is given in Algorithm 1. Iterating this idea, we may solve $Ae = r$ on a hierarchy of nested grids of varying coarseness to obtain an approximation that converges quickly at all wavelengths. The standard grid has $2^{k_l} + 1$ points on each level with grid spacing doubling with each coarsening.

There are many variants of a multigrid method, as one can use different algorithms for each component part. There are three main components: (1) *smoothing*, iterations of a solver like Jacobi or Gauss–Seidel to improve the approximate solution on a given grid; (2) *restriction*, approximating the residual from a fine grid onto a coarser grid; and (3) *prolongation*, interpolating a solution from a coarse grid onto a finer grid. In addition to these, there are different choices of *cycles*, *i.e.*, when to traverse between different grid levels. In the simplest of these, the V cycle, the algorithm starts on the finest grid, smooths and coarsens on each level in turn. On reaching the coarsest grid, the algorithm reverses direction, in turn smoothing and refining on each level. Other common choices are the W cycle and the F cycle [22].

The important property shared by all multigrid variants is that the norm of the residual $\|r\| = \|g - A\hat{X}\|$ decreases by a fixed factor every cycle (excepting cases where the algorithm has failed). This ensures that the solution converges to a fixed tolerance in a finite number of cycles. We show later that this ensures that multigrid requires total work that increases linearly with problem size $\sim \mathcal{O}(n)$, and the number of halo cell communications grows slowly as $\sim \mathcal{O}(\log_2 p)$.

Owing to these properties, any multigrid variant should perform well. We now give details of the implementation we benchmark in Sec. 3, namely smoothing with red-black Gauss–Seidel, restriction with the "full-weighting" operator, and prolongation with linear interpolation. With these choices, the norm of the residual reduces from $\|r\|$ to $0.06\|r\|$ for every V cycle [22, Table 4.2], a convergence rate we observe in our implementation.

2.3.1. Red-black Gauss–Seidel

Red-black Gauss–Seidel is a parallelizable variant of the Gauss–Seidel method for obtaining an approximate solution to $AX = g$. Alternate grid points are labelled red and black, and the approximate solution $\hat{X}$ is updated in two passes, first for red points,

$$\hat{X}^+_{2k} = \frac{1}{b_{2k}} \left(r_{2k} - a_{2k}\hat{X}_{2k-1} - c_{2k}\hat{X}_{2k+1} \right), \tag{14a}$$

for $k = 0, \ldots, (n+1)/2$, and then for black points,

$$\hat{X}^+_{2k+1} = \frac{1}{b_{2k+1}} \left(r_{2k+1} - a_{2k+1}\hat{X}^+_{2k} - c_{2k+1}\hat{X}^+_{2k+2} \right), \tag{14b}$$

for $k = 0, \ldots, (n-1)/2$. Here $\hat{X}^+$ denotes the updated approximation to $\hat{X}$. Importantly, each red update depends only on black points, and *vice versa*, meaning that each update in a pass is independent and can be performed in parallel. This is unlike the original Gauss–Seidel method

$$\hat{X}^+_k = \frac{1}{b_k} \left(r_k - a_k\hat{X}^+_{k-1} - c_k\hat{X}_{k+1} \right), \tag{15}$$

for $k = 0, \ldots, n$, where each update has a serial dependence on the previous update $\hat{X}^+_{k-1}$. In both (14) and (15) half the grid points used in the update are from the current approximation $\hat{X}$ and half are from the update $\hat{X}^+$. In red-black Gauss–Seidel, these are grouped so that all red points are updated using $\hat{X}$, and all black points are updated using $\hat{X}^+$. This means that the residual $r_{2k+1} - A\hat{X}^+_{2k+1} = 0$ by construction for all black points. This allows us to omit some communication when constructing the residual on the coarse grid.

2.3.2. Prolongation

Prolongation, or interpolation, is the procedure for approximating a coarse grid solution on a finer grid. We use linear interpolation; this may be represented by the matrix I in

$$IX^c = \frac{1}{2}\begin{pmatrix} 2 & & \\ 1 & 1 & \\ & 2 & \\ & 1 & 1 \\ & & 2 \end{pmatrix}\begin{pmatrix} X^c_0 \\ X^c_1 \\ X^c_2 \end{pmatrix} = \begin{pmatrix} X^c_0 \\ (X^c_0 + X^c_1)/2 \\ X^c_1 \\ (X^c_1 + X^c_2)/2 \\ X^c_2 \end{pmatrix} = X^f, \tag{16}$$

where X^c and X^f represent matrices on coarse and fine grids respectively.

2.3.3. Restriction

Restriction is the opposite operation to prolongation, namely approximating a finer grid vector on a coarser grid. As multigrid grids are nested, it is tempting to simply take values from corresponding grid points. However, the coarse grid problems better retain the properties of the full problem if the restriction operation R is proportional to the transpose of the interpolation operation, $R = I^T$ [22]. In the case of the linear interpolation, the restriction operator is called the full weighting operator, and is

$$RX^f = \frac{1}{4}\begin{pmatrix} 2 & 1 & & & \\ & 1 & 2 & 1 & \\ & & & 1 & 2 \end{pmatrix}\begin{pmatrix} X^f_0 \\ X^f_1 \\ X^f_2 \\ X^f_3 \\ X^f_4 \end{pmatrix} = \begin{pmatrix} (2X^f_0 + X^f_1)/4 \\ (X^f_1 + 2X^f_2 + X^f_3)/4 \\ (X^f_3 + 2X^f_4)/4 \end{pmatrix} = X^c. \tag{17}$$

2.3.4. Coarse grid problems

Whatever choice is made for restriction R and prolongation I, the matrix of the problem to solve on the coarse grid is found from the following consideration. Writing the coarse and fine grid problems $A^cX^c = g^c$ and $A^fX^f = g^f$ respectively, we have

$$\begin{aligned} A^fX^f &= g^f \\ A^fIX^c &= Ig^c \\ RA^fIX^c &= RIg^c \approx g^f \\ \implies A^c &\equiv RA^fI. \end{aligned} \tag{18}$$

For each level of multigrid, we therefore construct the system matrix A^c for the system matrix of the level above A^f. When A is fixed throughout a simulation, each of the coarse grid matrices may be calculated once during initialization. Note that the coarse grids on every level are tridiagonal.

Coarse grid coefficients. With linear interpolation (16) and full weighting (17), the elements of the coarse matrix A^c are

$$a^c = \frac{1}{4}a^f_- + \frac{1}{8}b^f_- + \frac{1}{4}a^f \tag{19a}$$

$$b^c = \frac{1}{8}b^f_- + \frac{1}{4}c^f_- + \frac{1}{4}a^f + \frac{1}{2}b^f + \frac{1}{4}c^f + \frac{1}{4}a^f_+ + \frac{1}{8}b^f_+ \tag{19b}$$

$$c^c = \frac{1}{4}c^f + \frac{1}{8}b^f_+ + \frac{1}{4}c^f_+, \tag{19c}$$

where a, b and c are the sub-, on-, and super-diagonal elements of either A^c or A^f, depending on superscript. Terms on the right-hand side with no subscript are evaluated at the same grid point as the left-hand side term – the point is shared between the grids. Terms with subscript plus or minus are evaluated at the point above or below respectively *on the finer grid*. These points do not exist on the coarser grid.

2.4. Convergence checking and residual calculation.

To check for convergence, we adopt the weighted error measure used in the ODE solver PVODE [25],

$$\|E\| \equiv \left[\sum_{i=0}^{N_x-1} \frac{1}{N_x}\left(w_i E_i\right)^2\right]^{1/2}, \tag{20}$$

where E_i is the elementwise residual for the original (not reduced) problem,

$$E_i = |(f - Mx)_i|, \tag{21}$$

and w_i is a weight accounting for both absolute and relative error,

$$w_i = \frac{1}{\texttt{rtol}|x_i| + \texttt{atol}}, \tag{22}$$

with `rtol` and `atol` user inputs for the required relative and absolute tolerances respectively. The method is converged when $\|E\| < 1$.

This choice of error measure has a number of advantages. Firstly, it allows direct comparison between the errors in our inversion algorithm and the errors in the iterative PVODE time advance algorithm.

Secondly, by combining relative and absolute tolerances in a single weight, it allows the algorithm to converge when each point is converged in either the absolute or the relative error. This is in contrast to the other commonly-used error conditions

$$E_A \equiv \|f - Mx\| < \texttt{atol}, \qquad E_R \equiv \|f - Mx\|/\|x\| < \texttt{rtol}, \tag{23}$$

where $\|\cdot\|$ is the Euclidean norm. The latter requires all points to converge in one of the absolute or relative measures, rather than allowing some points to converge in one and some in the other. This means that the algorithm is more robust when using the PVODE error measure (20).

Finally, we note that the errors (20) and (23) are written in terms of the original problem $Mx = f$, not the reduced problem $AX = g$ that is solved by our algorithm. This is potentially a problem, as reconstructing x from X for every convergence check would represent a large amount of work relative to the other operations in an iteration – reconstructing the solution is $\mathcal{O}(N_x/N_p)$ operations per processor, while the multigrid work for a processor's single X grid point is $\mathcal{O}(1)$. However, we may write (20) in terms of the reduced problem. Given any boundary values X_q and X_{q+1} for processor

q, our algorithm constructs a solution which satisfies $Mx = f$ on the interior points of q. Thus by construction, the residuals of interior points are zero. Moreover, the values of x in boundary cells correspond to the values on the reduced X grid. Therefore we may write (20) as

$$\|E\| \equiv \left[\sum_{q=0}^{N_p-1} \frac{1}{N_x} \left(\hat{w}_q \hat{E}_q \right)^2 \right]^{1/2}, \tag{24}$$

where the elementwise error $\hat{E}_q$ is now

$$\hat{E}_q = |(g - AX)_q|, \tag{25}$$

and the weight $\hat{w}_q$ is now

$$\hat{w}_q = \frac{1}{\texttt{rtol}|X_q| + \texttt{atol}}. \tag{26}$$

Thus $\|E\|$ may be computed without knowing the full solution x; this is not possible with the error (23), as the relative error $\|E_R\|$ cannot be computed without knowing all of x.

2.5. Complexity and communication

An outline of the full algorithm is given in Algorithm 2. As there is little opportunity for computation/communication overlap, the total runtime is proportional to the runtime for the different components.

The cost of solving the local subsystems with the Thomas algorithm is $\mathcal{O}(N_x/N_p)$. As these are local to processor, there is no communication cost.

The scaling for multigrid component of our algorithm is slightly different from usual multigrid scaling, as the size of the multigrid system varies with processor count. The complexity of a multigrid system of size N_x is calculated as follows (see for example [24], §§2.4.3 and 6.2.1). Consider solving in serial. The error reduction per V cycle is independent of the finest grid size, so reducing the residual from $\mathcal{O}(1)$ to a given tolerance takes a constant number of V cycles. Therefore the total work is proportional to the work in one V cycle. The work on level k of a V cycle is proportional to the grid size $W_k = CN_k$. Summing all levels and noting that the ratio of neighbouring grid sizes is approximately constant (in our case $\rho = N_{k-1}/N_k = (2^{k-1}+1)/(2^k+1) \approx 1/2$), we find the total work is $W = \sum_{k=1}^{l} W_k = CN_l \sum_{k=0}^{l-1} \rho^k$, where l is the number of multigrid levels and $N_l = N_x$ is the resolution of the finest grid. Thus work is $\mathcal{O}(N_x)$, as $\rho < 1$.

In parallel, the work can be distributed over all processors, so the runtime is $\mathcal{O}(N_x/N_p)$. There are however now two sources of communication cost: nearest neighbour communications in smoothing, refining and prolongation, and a global `all reduce` to synchronize the summed residual for convergence checking.

The nearest neighbour communications are performed a fixed number of times per iteration. While the number of V cycles is independent of N_x and N_p, the number of iterations per V cycle increases with the number of multigrid levels, and therefore increases with N_x according to $N_x = 2^l + 1$. With two visits to each level per V cycle, this implies the number of iterations is $N_i = 2l = 2\log_2(N_x - 1)$ so the nearest neighbour communication time scales as $\mathcal{O}(\log_2 N_x)$.

The `all reduce` to synchronize summed residuals in the convergence check is performed once per V cycle, *i.e.* $\mathcal{O}(1)$ calls. The cost of the `all reduce` depends on the implementation, machine (and machine state!) but we observe it to be around $\mathcal{O}\left(N_p^{5/4}\right)$. The total cost of multigrid is therefore $\mathcal{O}(N_x/N_p) + \mathcal{O}(\log_2 N_x) + \mathcal{O}\left(N_p^{5/4}\right)$.

Algorithm 2: Iterative tridiagonal solver

```
Initialize left-hand side. Invert local matrix to calculate coefficients for constructing the solution from halo cells. Use
  these coefficients to calculate coefficients of the matrix of the reduced problem. Calculate coarsened versions of this
  matrix for each multigrid level. When M is constant, cache these values;
Initialize right-hand side. Invert local problem to calculate coefficient for constructing the solution arising from M^{-1} f.
  Calculate this term's contribution to the reduced system's equations. These terms cannot be cached, as f changes every
  timestep;
Set initial guess to solution from previous timestep;
while true do
    smooth solution using Gauss–Seidel with red-black colouring;
    if done enough smoothings at this level then
        calculate the residual;
    if not enough subiterations then
        continue;
    else if residual tolerance met then
        exit;
    else if refining then
        refine the grid;
        update solution using residual calculated on previous level;
        reset smoothing count;
        if now on finest grid then
            stop refining, start coarsening;
    else
        coarsen the grid;
        reset smoothing count;
        if now on coarsest grid then
            stop coarsening, start refining;
Cache reduced solution for use on next timestep;
Construct full solution from halo cell values;
```

Now considering our implementation, our grid size is $N_p + 1$, so that the amount of multigrid work on each processor is $\mathcal{O}((N_p+1)/N_p) \sim \mathcal{O}(1)$ and the number of nearest neighbour communications is $\mathcal{O}(\log_2 N_p)$. The global `all reduce` still scales as $\mathcal{O}\left(N_p^{5/4}\right)$. Our expected runtime is therefore

$$T = A\frac{N_x}{N_p} + B\log_2(N_p) + C + DN_p^{5/4}. \tag{27}$$

While this theoretical runtime has the same scaling as parallel cyclic reduction, we see in Sec. 3 that our method yields better scaling for an initial value problem.

2.6. Minimizing communication

As communication dominates our algorithm's cost at scale, we consider some methods for reducing communication.

2.6.1. Simultaneous solution of subsystems

We often need to invert many independent tridiagonal systems with different coefficients. This arises for example in inverting a two-dimensional Laplacian with a Fourier transform in one of the dimensions, $\nabla^2 = \partial^2/\partial x^2 + \partial^2/\partial z^2 \mapsto \partial^2/\partial x^2 - k_z^2$, where the x dimension is distributed, but the z direction is local to processor. We then have N_z independent tridiagonal systems in x, parameterized by k_z. Each processor holds the data for its x-subdomain for each of the N_z systems. We could invert these systems one at a time. However, doing so requires $\mathcal{O}(\sum_{k_z} N_i(k_z)) \sim \mathcal{O}(N_z \bar{N}_i)$ halo cell communications of a single number, where N_i and $\bar{N}_i$ are the number of iterations as a function of k_z and its average over k_z. This is inefficient due to the overheads of sending many small messages.

Instead, we perform the inversion for each system simultaneously, and communicate vectors of length N_z. While this does not significantly increase the cost of each send, it decreases the number of sends to $\mathcal{O}(\max N_i)$, which is smaller by around a factor of N_z. To prevent unnecessary work, we skip loop iterations for the k_z modes that have converged.

2.6.2. Predicting convergence

At large core counts, the dominant communication cost is from the `all reduce` needed to synchronize the summed error E (24) to ensure all processors exit on the same iteration. To minimize `all reduce` calls we take advantage of the fact that the error reduction per V cycle, $R = E^+/E$, is constant when the algorithm is proceeding normally. The constant R depends on the smoothing, prolongation and refinement methods, and are tabulated in [22, Table 4.2]. We compute R using the errors from the second and third V cycle (the first cycle is often atypical) and use it to predict the number of further V cycles needed for the slowest converging k_z to meet the error tolerance. We then do not calculate the global summed error – thus skipping the `all reduce` – until all modes are expected to have converged.

In our experiments, this reduced the number of `all reduce` calls from $\sim \mathcal{O}(7)$ to 3. Unfortunately, the decrease in communication time is offset by an increased amount of work required: since we no longer know which modes have converged, we can no longer skip the corresponding work. Whether this yields a performance improvement appears to be problem-dependent. We have therefore left this as an option in our implementation, but have not used this method in our results presented in Sec.3.

3. Numerical experiments

We now assess the performance of our algorithm by using it to invert a discretized Laplacian in a time-evolving partial differential equation system. We consider a simple model for plasma filament propagation, the "blob2d" example

in BOUT++ [15],

$$\frac{\partial n}{\partial t} = -\{\phi, n\} + 2\frac{\partial n}{\partial z} + D_n \nabla^2 n, \tag{28a}$$

$$\frac{\partial \Omega}{\partial t} = -\{\phi, \Omega\} + 2\frac{\partial n}{\partial z} + D_\Omega (\nabla^2 \Omega)/n, \tag{28b}$$

$$\nabla^2 \phi = \Omega, \tag{28c}$$

in a two-dimensional box (x, z) with Dirichlet boundary conditions in x and periodic boundary conditions in z. In (28), n is plasma density, Ω is vorticity, ϕ is electrostatic potential, t is time, D_n and D_Ω are dissipation parameters, and $\{A, B\} = (\partial A/\partial x)(\partial B/\partial z) - (\partial A/\partial z)(\partial B/\partial x)$ is a Poisson bracket. We use the two-dimensional Laplacian $\nabla^2 \equiv \partial^2/\partial x^2 + \partial^2/\partial z^2$. Between each timestep, the vorticity equation (28c) must be solved for ϕ so its value can be used in (28a) and (28b) to advance n and Ω. We use ϕ at the current timestep as an initial guess for the iterative method (and $\phi = 0$ as the guess for the first time step). We parallelize only in the x direction, with the z direction remaining local to processor. As the domain is periodic in z, we Fourier decompose in that direction and solve (28c) as a one-dimensional problem in x, solving independent k_z-modes simultaneously as described in Sec. 2.6.1.

This model is implemented in BOUT++ [15], a modular framework for writing fluid and plasma simulations. We implemented our tridiagonal matrix inversion algorithm as a module in BOUT++, and now compare it to BOUT++'s implementations of parallel cyclic reduction, of pure multigrid, and of a domain-partitioned direct solver. The parallel cyclic reduction is based on the implementation by Kang [26]. The multigrid implementation is the same as described in Section 2.3; for fairness it is implemented using the same data structures and communication patterns as the hybrid multigrid-Thomas algorithm. The pure multigrid algorithm is applied on the full system rather than the reduced system. This means there is no longer the need to reconstruct the full solution from the solution on the reduced grid, saving $\mathcal{O}(N_x/N_p)$ work; but instead we must perform more levels of multigrid, requring $\mathcal{O}(N_x/N_p)$ more work from performing Gauss–Seidel iterations on the finer grids, and $\mathcal{O}(\log(N_x/N_p))$ additional communications from the corresponding halo swaps. The partitioned direct solver we use was derived by Austin *et al.* [27]. It is similar to our algorithm in that it uses local solves on each processor to derive the same reduced system. However, instead of using multigrid, it gathers the reduced system onto a single processor, solves directly with the Thomas algorithm, and scatters the results back. That is, it replaces the $\mathcal{O}(\log N_p)$ halo swap communications from multigrid's iterations, with single all-to-one and one-to-all gather/scatter communications of size N_p.

The code is run on the Archer2 HPC system (two 64 core AMD Zen2 7742 processors per node, 2.25 GHz, with HPE Cray Slingshot 2x100 Gbps bi-directional interconnect per node). Timings are taken from BOUT++ internal timers, and are for the evolution of equations (28) including the initialization of our algorithm, but excluding other code initialization and I/O. To aid reproducibility, we have made available an archive containing BOUT++ input and output files, Archer2 job submission scripts, and our scripts for processing and plotting the results [28]. This archive also contains a script to automatically download and build BOUT++ with the same git commit, Archer2 modules and runtime environment as we used to generate the results presented here.

We present run times for evolving (28) with two time advance algorithms, the explicit non-adaptive fourth-order Runge–Kutta (RK4) scheme, and the implicit, adaptive timestep and adaptive order PVODE solver [25]. As RK4 is non-adaptive, it performs a specified number of time steps regardless of the state of physical system (28). Therefore RK4 gives a simple measure of speed of the Laplacian inversion algorithm, in the context of the evolution of a full system. We show results of this numerical study in Sec. 3.1. In contrast, PVODE uses an implicit linear multistep method which

adapts the size of the timestep and the order of method depending on the stiffness of the problem (*i.e.* the current physical state). Rather than specifying a timestep, the user specifies relative and absolute tolerances for the error in the time advance, using the error expression (20). This time advance method is preferred in BOUT++, as it requires minimal user input, and usually provides faster wall-time-to-solution than RK4. However, as the timestep is adaptive, the number of times that the Laplacian inversion is called varies depending on the state of the physical system, and in particular varies between the different algorithms, and the input parameters to these. In Sec. 3.2 we plot run times for the same set of solvers, but now also vary the number of multigrid levels in our multigrid-Thomas solver, and show that this and the number of Laplacian inversion calls strongly influences the overall run time.

3.1. Fourth-order Runge–Kutta

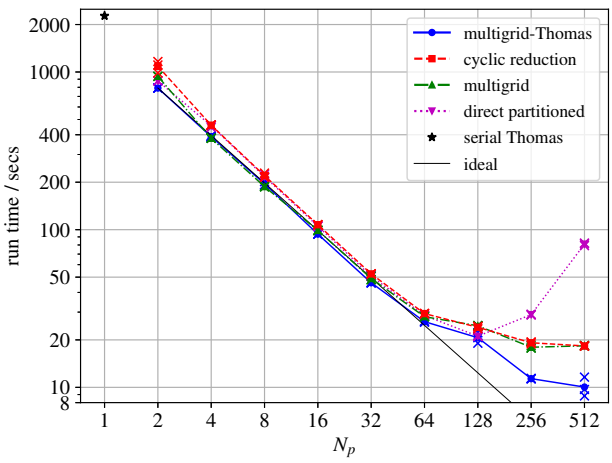
We advance the system (28) with fourth-order Runge–Kutta for two sets of resolutions, $(N_x, N_z) = (1024, 1024)$ and $(N_x, N_z) = (8192, 1024)$. We evolve the smaller case for 2000 timesteps, which requires 8000 inversions of equation (28c), and the larger case for 1000 timesteps, which requires 4000 inversions. We plot run time against processor count for these resolutions for our algorithm (solid blue), cyclic reduction (dashed red), pure multigrid (dot-dashed green) and the direct partitioned algorithm (dotted magenta) in Fig. 1(a) and (b), and corresponding parallel efficiencies in Fig. 1(c) and (d). In both multigrid algorithms, we use the tolerances `rtol` $= 10^{-7}$ and `atol` $= 10^{-6}$, and set the number of multigrid levels to the maximum number possible. For our algorithm, this varies with core count, $\log_2(N_p) - 1$. For pure multigrid, this is constant for fixed problem size, $\log_2(N_x) - 1$.

For the smaller problem, all algorithms scale ideally to 64 cores, before dropping to around 60% to 70% efficiency at 128 cores. This is due to increased contention for memory on one Archer2 node (128 cores): at smaller core counts we have spread ranks even across a single node, so that each doubling in core count halves the available memory to each core. Above 128 cores, efficiency degrades in all algorithms, though our multigrid-Thomas algorithm scales the best, with run time continuing to reduce until reaching the maximum core count, 512 cores (BOUT++ is constrained to require at least 2 x-points per core). For the larger problem, we again see near-ideal scaling for all algorithms at small core counts, with performance degradation due to resource contention as we approach 1 node (128 cores). Above 128 cores, parallel cyclic reduction and the multigrid algorithms scale super-ideally. Again our multigrid-Thomas algorithm scales best, retaining ideal scaling relative to a single node up to 16 nodes (2048 cores), and performing with 70% efficiency at the maximum 32 nodes (4096 cores, 2 x-points per core).

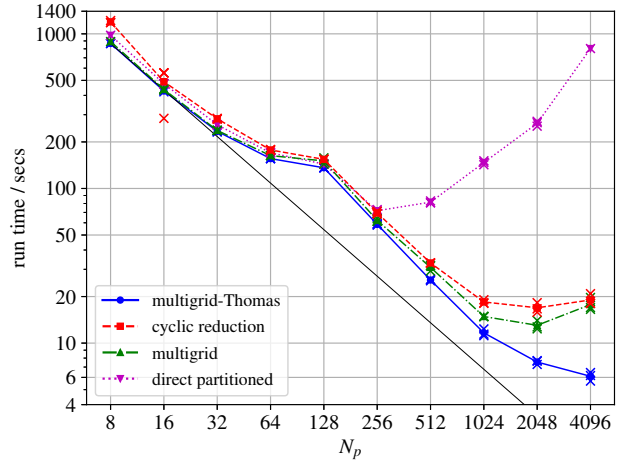
In Fig. 1(e) and (f) we plot the percentage speed up of our algorithm, relative to parallel cyclic reduction (red), pure multigrid (green) and the direct partitioned algorithm (magenta). This shows that our algorithm is around 15% faster than cyclic reduction, around 10% faster than the direct partitioned algorithm and around 5% faster than pure multigrid for low and medium processor counts (except two small processor counts at the lower resolution where pure multigrid is faster). It also shows that our algorithm is significantly faster than all other algorithms at high processor counts as expected from the improved scaling.

3.2. PVODE time advance

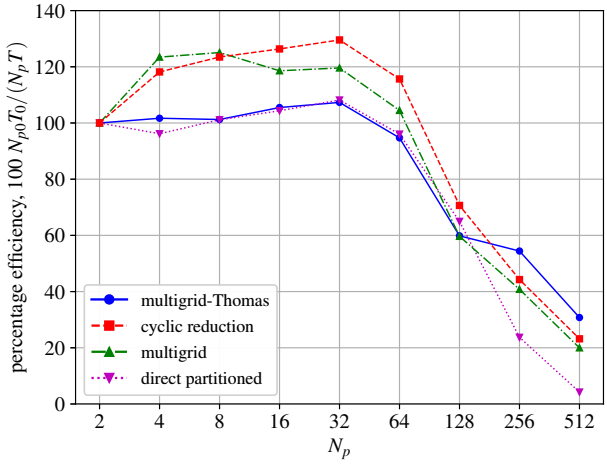
We now show results for the PVODE timestepping algorithm. To use PVODE, we provide that library with a function evaluating the right-hand sides of (28)(a,b) for each iteration of n and Ω; this function call includes inverting (28c) to find ϕ, *i.e.* one call of the Laplacian inversion algorithm. As PVODE continues to iterate for n and Ω until the specified tolerances are met, the number of times the right-hand side is called varies depending on the Laplacian



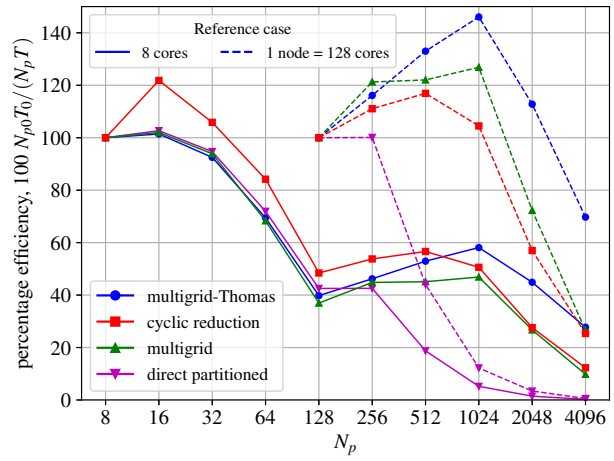
(a) $N_x = 1024$



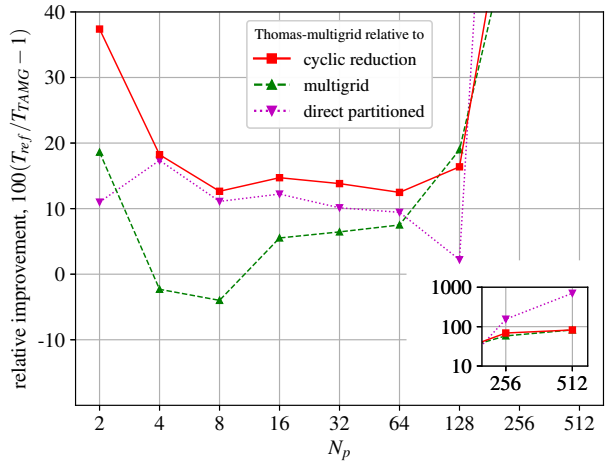
(b) $N_x = 8192$



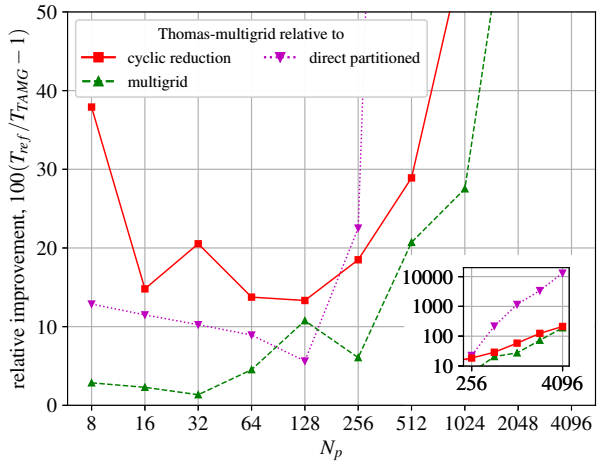
(c) $N_x = 1024$



(d) $N_x = 8192$



(e) $N_x = 1024$



(f) $N_x = 8192$

Figure 1: Plots for the blob2d filament simulation with resolution (left column) $N_x = 1024$ and (right column) $N_x = 8192$. (a,b) Scaling plots for cyclic reduction (dashed red), pure multigrid (dot-dashed green), the direct partitioned algorithm (dotted magenta) and our algorithm (solid blue). We mark the run time for serial Thomas in the smaller case with a star. (c,d) The parallel efficiencies for timings shown in (a,b). In (d) we plot the same data using both 8 cores (solid) and 128 cores = 1 node (dashed) as the reference case. (e,f) The relative performance improvement of our algorithm compared to the other algorithms, expressed as a percentage for the timings shown in (a) and (b) respectively.

inversion algorithm used, and each Laplacian inversion algorithm's parameters. This means that the run time now not only depends on the time per Laplacian inversion, but also the number of Laplacian inversions that are required.

We consider a simulation following a filament for 400 cyclotron times with $(N_x, N_z) = (8192, 1024)$, and with the tolerances $(\texttt{atol}, \texttt{rtol}) = (10^{-7}, 10^{-6})$ for our algorithm and the multigrid algorithm, and $(\texttt{atol}, \texttt{rtol}) = (10^{-6}, 10^{-5})$ for the PVODE time advance. This corresponds to the larger problem in Sec. 3.1, though run for a longer physical time, so that unnormalized run times are not comparable.

In Figure 2(a) we plot the total run time against core count for simulations using the PVODE time advance with the Laplacian inversion performed using parallel cyclic reduction (dashed red), pure multigrid (dot-dashed green), the direct partitioned algorithm (dotted magenta), and the multigrid-Thomas algorithm (solid colours). We also plot the parallel efficiency relative to 128 cores (one Archer2 node) in Figure 2(b) and the speed-up of the multigrid-Thomas algorithm relative to the other algorithms in Figure 2(c). Different line colours correspond to the maximum number of multigrid levels used in the multigrid-Thomas algorithm. Recall that as each core represents a grid point, the maximum possible number of multigrid levels increases with core count as $\log_2(N_p) - 1$. In contrast, the maximum number of levels in pure multigrid is set by the problem size as $\log_2(N_x) - 1$, so in this case is 12. While the overall scaling behaviour is independent of `max_level`, the maximum number of multigrid levels, at a fixed core count there can be significant differences in the run time depending on `max_level`. Moreover, the fastest run times do not correspond to setting `max_level` to the maximum possible number of levels; rather the optimal value is problem-dependent and requires user tuning. In this particular case, setting `max_level` $= 4$ results in ideal scaling up to 1024 cores (8 nodes, 8 x-points per core). The parallel efficiency in Figure 2(b) confirms that our algorithm scales with at least 90% efficiency for all values of `max_level` up to 1024 cores, before dropping to around 60% and 30% efficiency at 2048 and 4096 cores respectively (except for `max_level` $= 6$ which has 80% at 2084 cores). While this is reasonably good scaling efficiency, it is somewhat worse than the efficiency seen in Figure 1(b) for the large RK4 test case (for parallel efficiency relative to a single node, the dashed lines). Indeed both algorithms with a multigrid component have a worse parallel efficiency with PVODE; in contrast, parallel cyclic reduction has an almost identical parallel efficiency for the two time advance methods. Consequently, parallel cyclic reduction is now more competitive with multigrid-Thomas with the two algorithms having similar parallel efficiencies. However, the relative speed-up graph in Figure 2(c) shows that multigrid-Thomas is still 10% to 20% faster in the good scaling region (except at 512 cores), similiar to the speed-up in the good scaling region of the larger RK4 case, Figure 1(f). When the multigrid-Thomas algorithm stops scaling ideally at 1024 cores (with `max_level` $= 4$), it is $\sim 20\%$ faster than parallel cyclic reduction, $\sim 30\%$ than pure multigrid, and $\sim 80\%$ faster than the direct partitioned algorithm. As before, the multigrid-Thomas algorithm attains its fastest run time at the maximum 32 nodes (4096 cores, 2 x-points per core).

We can understand the spread in run times for different values of `max_level` in the multigrid-Thomas algorithm by considering the number of times the Laplacian inversion is called, which we plot in Figure 2(d) against core count. These values are noisy and with no discernible dependence on core count, but typically parallel cyclic reduction requires a similar number of Laplacian inversions or fewer when compared to the multigrid-Thomas algorithm. In Figure 2(e) we plot the run times from Figure 2(a) normalized to the number of Laplacian inversions. This shows that multigrid-Thomas is the fastest algorithm per single inversion. This essentially replicates the result shown in Figure 1(b). Moreover, after normalization the lines for different `max_level` largely coincide, showing that the variation in the run time is accounted for by variation in the number of Laplacian inversions required.

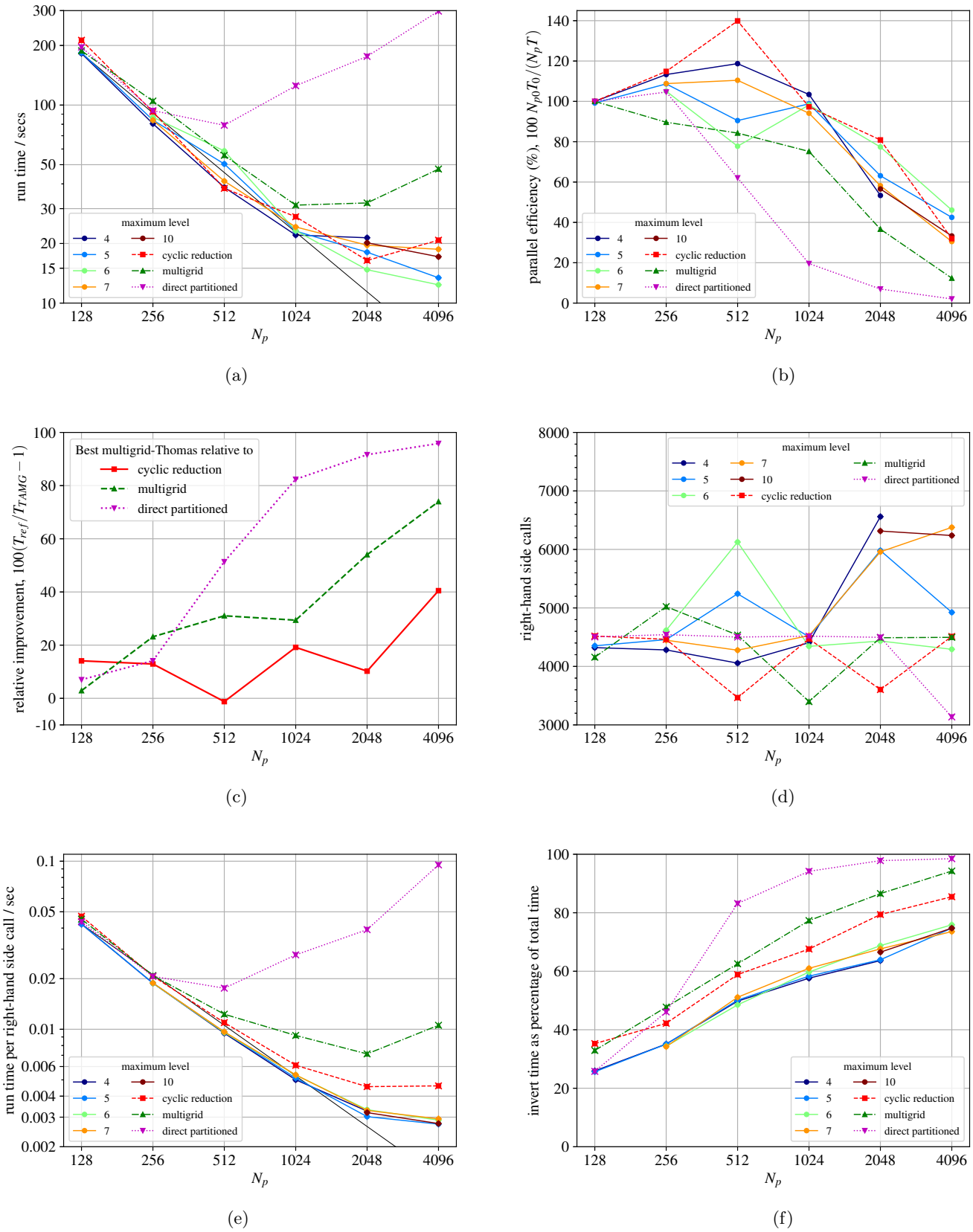


Figure 2: Metrics for PVODE time advance with Laplacian inversion performed by cyclic reduction (dashed red), pure multigrid (dot-dashed green), the direct partitioned algorithm (dotted magenta) and our algorithm varying the maximum number of multigrid levels (solid colours), plotted against core count: (a) total run time; (b) parallel efficiency relative to 128 cores (one Archer2 node); (c) relative performance improvement of multigrid-Thomas compared to other algorithms; (d) number of Laplacian inversions; (e) total run time divided by number of Laplacian inversions; and (f) percentage of total time spent inverting the Laplacian.

Finally in Figure 2(f) we plot the percentage of total run time spent in the Laplacian inversion measured by BOUT++ internal timers. This shows that at a fixed core count, the multigrid-Thomas algorithm spends the smallest proportion of time in Laplacian inversion. For all algorithms, the proportion increases with core count, and dominates the run time at the highest core counts showing that inversion is indeed the scaling bottleneck.

4. Summary

In this paper we have introduced a hybrid multigrid–Thomas algorithm designed to efficiently invert one-dimensional tridiagonal matrix equations in a highly-scalable fashion. We implemented this algorithm as a module in the plasma code BOUT++ [15] and measured its performance in a model problem for plasma filament propagation using our algorithm to solve Poisson's equation as part of the spatial discretization of a time-evolving PDE system. We compared its performance to that of cyclic reduction, pure multigrid and a direct partitioned solver in two cases using different time-advance algorithms, non-adaptive fourth-order Runge–Kutta and an adaptive solver from the PVODE library. While both parallel cyclic reduction and pure multigrid also have the minimum theoretical complexity for parallel algorithms, $\log(N_x)$, the multigrid-Thomas algorithm is fastest per Laplacian inversion and scales best. Thus it is the fastest and most-scalable when using non-adaptive timestepping schemes, like fourth-order Runge–Kutta. When using the adaptive PVODE timestepping scheme, we found that the multigrid-Thomas algorithm required more internal timesteps to achieve a given tolerance which reduces its performance advantage over the other algorithms. However, the multigrid-Thomas algorithm's better scaling performance means that it still outperforms the other algorithms while still retaining good parallel efficiency.

4.1. Further work

There are two areas of further work which may extend the scalability of the hybrid multigrid algorithm.

Firstly, we could consider changes to the multigrid algorithm to reduce the amount of communication required. In this paper, we have used linear interpolation in the multigrid prolongation step. We could replace this with a higher order scheme, which would be more expensive to compute but which would converge at a faster rate. Faster convergence means fewer iterations and thus less communication; at high core counts, this might lead to faster run times, even after accounting for the increased work. As a concrete example, Briggs et al. [22, Table 4.2] studied convergence rates and costs for different smoothing, interpolation and prolongation methods. Our scheme (red-black Gauss–Seidel, linear interpolation, full weighting, and one pre- and post-smoothing, $\nu_1 = 1$, $\nu_2 = 1$) has a computational cost of 1.63 (relative to some baseline) and a convergence factor of 0.06 (*i.e.* one multigrid cycle reduces the residual from r to $0.06r$). For Briggs et al.'s model problem, replacing linear interpolation with cubic interpolation and increasing the number of pre-smoothings to $\nu_1 = 2$ roughly doubles the cost to 3.37 but also halves the convergence factor to 0.03 (*i.e.* doubles the rate). In a communication-bound computing regime, this may well yield reduced run times.

Finally, we could extend the scalability by exploiting the non-uniform memory access (NUMA) region of modern processor architecture [29]. Each each core has fast access to shared memory in the NUMA region, usually a socket or a node. In this paper, we have performed the direct solver on a core and the multigrid solve on a grid of cores. However, shared memory would allows us to perform the direct solve on a NUMA region, and multigrid across NUMA regions. This would extend the good scaling performance of our algorithm by a factor of the number of cores per NUMA region,

which on Archer2 is 8. This approach as been used to extend the scalability of the Fast Fourier Transform in the plasma code GS2 [30] on the original Archer system (which had 12 cores per NUMA region) by a factor of ~ 10 [31].

Acknowledgements

This work made use of computational support by CoSeC, the Computational Science Centre for Research Communities, through CCP Plasma (EP/M022463/1) and HEC Plasma (EP/R029148/1). This work used the ARCHER2 UK National Supercomputing Service (https://www.archer2.ac.uk) and the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk). J.T.P. is grateful for fruitful conversations with H. S. Thorne and F. C. Parker.

References

[1] J. Wendt, Computational Fluid Dynamics: An Introduction, A von Karman Institute book, Springer Berlin Heidelberg, 2008. URL: `https://books.google.co.uk/books?id=IIUkqI-HNbQC`.

[2] S. Jardin, Computational Methods in Plasma Physics, Chapman & Hall/CRC Computational Science, CRC Press, 2010. URL: `https://books.google.co.uk/books?id=gZzf_B56FDcC`.

[3] R. W. Hockney, A fast direct solution of Poisson's equation using Fourier analysis, Journal of the ACM (JACM) 12 (1965) 95–113.

[4] D. Bertaccini, F. Durastante, Iterative Methods and Preconditioning for Large and Sparse Linear Systems with Applications, Chapman & Hall/CRC Monographs and Research Notes in Mathematics, CRC Press, 2018. URL: `https://books.google.co.uk/books?id=YmpQDwAAQBAJ`.

[5] G. D. Knott, Interpolating Cubic Splines, Progress in Computer Science and Applied Logic, Birkhäuser Boston, 2012. URL: `https://books.google.co.uk/books?id=7SPUBwAAQBAJ`.

[6] R. Ferguson, Practical Algorithms for 3D Computer Graphics, Second Edition, Taylor & Francis, 2013. URL: `https://books.google.co.uk/books?id=NKONAgAAQBAJ`.

[7] M. Løiten, 2017. Ph.D. thesis, Technical University of Denmark `https://github.com/CELMA-project/dissertation/releases/download/v1.1.x/17_PhD_Loeiten.pdf`.

[8] E. Gallopoulos, B. Philippe, A. Sameh, Parallelism in Matrix Computations, Scientific Computation, Springer Netherlands, 2015. URL: `https://books.google.co.uk/books?id=q9xECgAAQBAJ`.

[9] X.-H. Sun, Application and accuracy of the parallel diagonal dominant algorithm, Parallel Computing 21 (1995) 1241–1267.

[10] X.-H. Sun, S. Moitra, A fast parallel tridiagonal algorithm for a class of CFD applications, volume 3585, Citeseer, 1996.

[11] H. S. Stone, An efficient parallel algorithm for the solution of a tridiagonal linear system of equations, Journal of the ACM (JACM) 20 (1973) 27–38.

[12] H. S. Stone, Parallel tridiagonal equation solvers, ACM Transactions on Mathematical Software (TOMS) 1 (1975) 289–307.

[13] E. Polizzi, A. H. Sameh, A parallel hybrid banded system solver: the SPIKE algorithm, Parallel computing 32 (2006) 177–194.

[14] B. S. Spring, E. Polizzi, A. H. Sameh, A feature complete SPIKE banded algorithm and solver, arXiv preprint arXiv:1811.03559 (2018).

[15] B. D. Dudson, P. A. Hill, D. Dickinson, J. T. Parker, A. Allen, G. Breyiannia, J. Brown, L. Easy, S. Farley, B. Friedman, E. Grinaker, O. Izacard, I. Joseph, M. Kim, M. Leconte, J. Leddy, M. Løiten, C. Ma, J. Madsen, D. Meyerson, P. Naylor, S. Myers, J. Omotani, T. Rhee, J. Sauppe, K. Savage, H. Seto, D. Schwörer, B. Shanahan, M. Thomas, S. Tiwari, M. Umansky, N. Walkden, L. Wang, Z. Wang, P. Xi, T. Xia, X. Xu, H. Zhang, A. Bokshi, H. Muhammed, M. Estarellas, F. Riva, Bout++ v4.3.1, 2020. URL: https://doi.org/10.5281/zenodo.3727089. doi:10.5281/zenodo.3727089.

[16] R. P. Fedorenko, The speed of convergence of one iterative process, USSR Computational Mathematics and Mathematical Physics 4 (1964) 227–235.

[17] N. S. Bakhvalov, On the convergence of a relaxation method with natural constraints on the elliptic operator, USSR Computational Mathematics and Mathematical Physics 6 (1966) 101–135.

[18] W. Hackbusch, Ein iteratives Verfahren zur schnellen Auflösung elliptischer Randwertprobleme, Math. Inst., Univ., 1976.

[19] A. Brandt, Multi-level adaptive technique (MLAT) for fast numerical solution to boundary value problems, in: Proceedings of the Third International Conference on Numerical Methods in Fluid Mechanics, Springer, 1973, pp. 82–89.

[20] A. Brandt, Multi-level adaptive solutions to boundary-value problems, Mathematics of Computation 31 (1977) 333–390.

[21] P. Wesseling, Introduction to multigrid methods, 1995. NASA technical report.

[22] W. Briggs, V. Henson, S. McCormick, A Multigrid Tutorial, 2nd Edition, SIAM, 2000. https://www.researchgate.net/publication/220690328_A_Multigrid_Tutorial_2nd_Edition.

[23] W. Hackbusch, Multi-Grid Methods and Applications, Springer Series in Computational Mathematics, Springer Berlin Heidelberg, 2013. URL: https://books.google.co.uk/books?id=jJ36CAAAQBAJ.

[24] U. Trottenberg, C. Ulrich Trottenberg, C. Oosterlee, A. Schuller, A. Brandt, P. Oswald, K. Stüben, Multigrid, Elsevier Science, 2001. URL: https://books.google.co.uk/books?id=-og1wD-Nx_wC.

[25] G. D. Byrne, A. C. Hindmarsh, PVODE, an ODE solver for parallel computers, The International Journal of High Performance Computing Applications 13 (1999) 354–365. URL: https://doi.org/10.1177/109434209901300405. doi:10.1177/109434209901300405. arXiv:https://doi.org/10.1177/109434209901300405.

[26] J.-H. Kang, Parallel tri-diagonal matrix solver using cyclic reduction (CR), parallel CR (PCR), and thomas+PCR hybrid algorithm, 2019. URL: https://github.com/jihoonakang/parallel_tdma_cpp.

[27] T. M. Austin, M. Berndt, J. D. Moulton, A memory efficient parallel tridiagonal solver, Preprint LA-VR-03-4149 (2004).

[28] J. T. Parker, P. A. Hill, D. Dickinson, B. D. Dudson, Files and plotting scripts for "Parallel tridiagonal matrix inversion with a hybrid multigrid–Thomas algorithm method", 2020. URL: https://zenodo.org/record/4292047. doi:10.5281/zenodo.4292047.

[29] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, R. Thakur, MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory, Computing 95 (2013) 1121–1136.

[30] M. Barnes, D. Dickinson, W. Dorland, P. A. Hill, J. T. Parker, C. M. Roach, S. Biggs-Fox, N. Christen, R. Numata, G. Wilkie, L. Anton, J. Ball, J. Baumgaertel, G. Colyer, M. Hardman, J. Hein, E. Highcock, G. Howes, A. Jackson, M. T. Kotschenreuther, J. Lee, H. Leggate, N. Mandell, A. Mauriya, T. Tatsuno, F. Van Wyk, GS2 gyrokinetics software, 2020. URL: https://doi.org/10.5281/zenodo.2551066. doi:10.5281/zenodo.2551066.

[31] L. Anton, F. van Wyk, E. Highcock, C. Roach, J. T. Parker, Enhancing scalability of the gyrokinetic code GS2 by using MPI shared memory for FFTs, Proceedings of the Cray User Group (2016). URL: https://cug.org/proceedings/cug2016_proceedings/includes/files/pap124s2-file1.pdf.